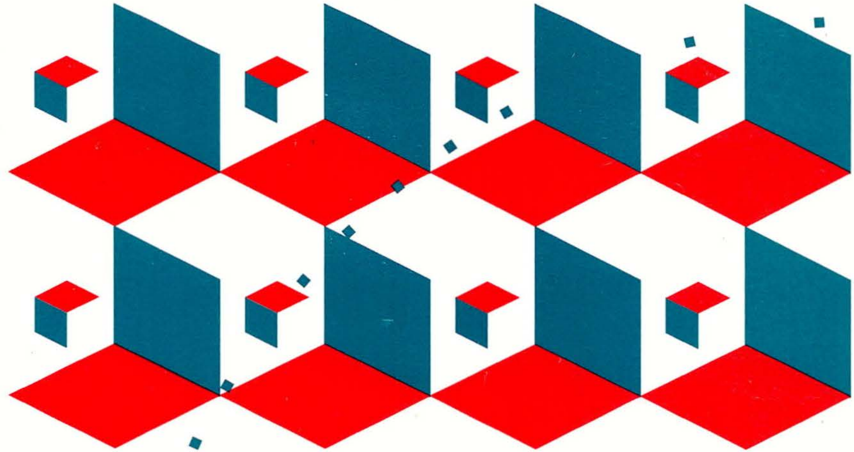
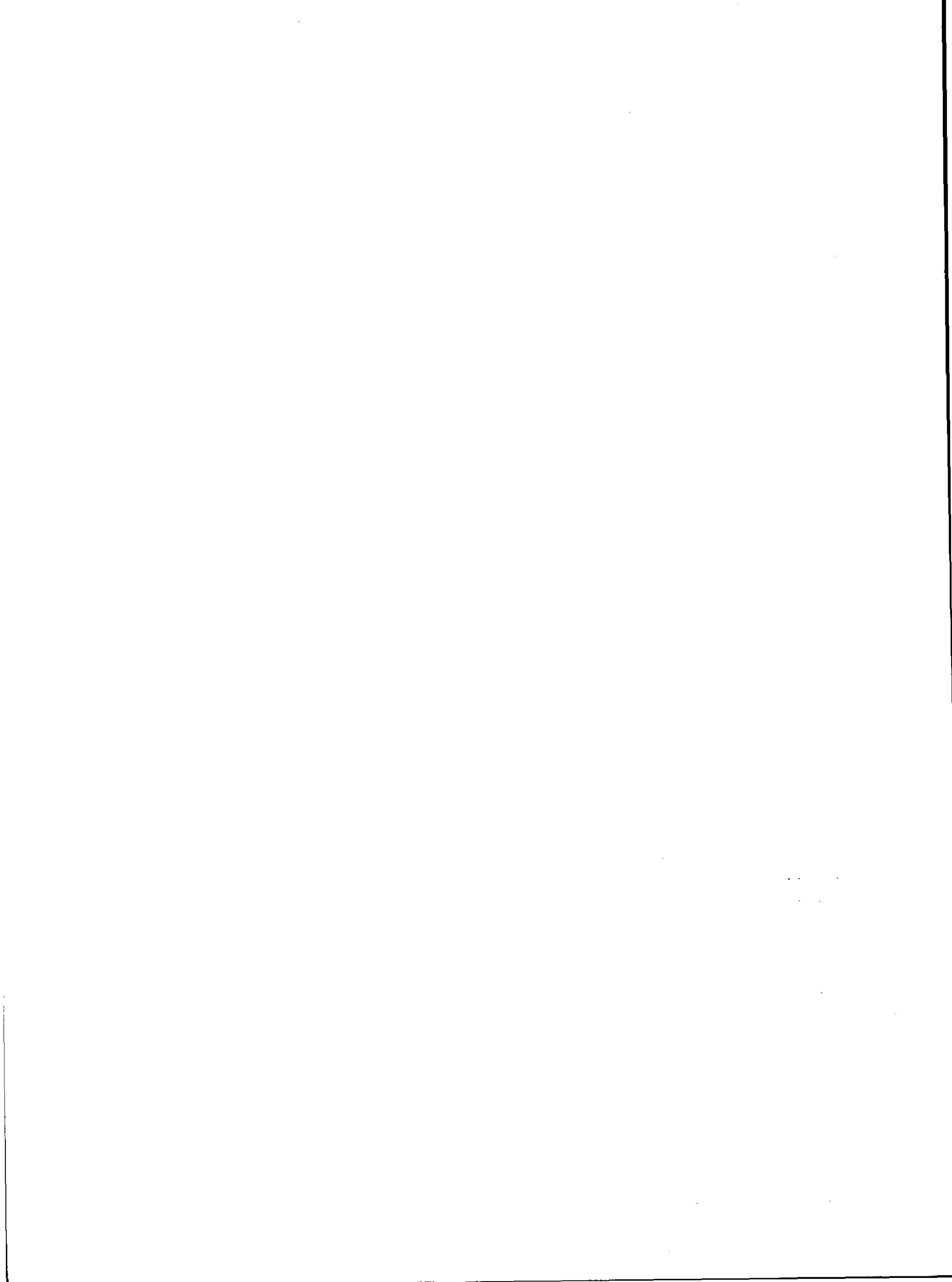


CONVEX

Exemplar Programming Guide

Third Edition

Hewlett-Packard Company
Convex Technology Center
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America



Exemplar Programming Guide

Order No. DSW-067

Third Edition

June 1996

Hewlett-Packard Company
Convex Technology Center
Richardson, Texas
United States of America

Exemplar Programming Guide

Order No. DSW-067

© Copyright Hewlett-Packard Company 1996. All Rights Reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

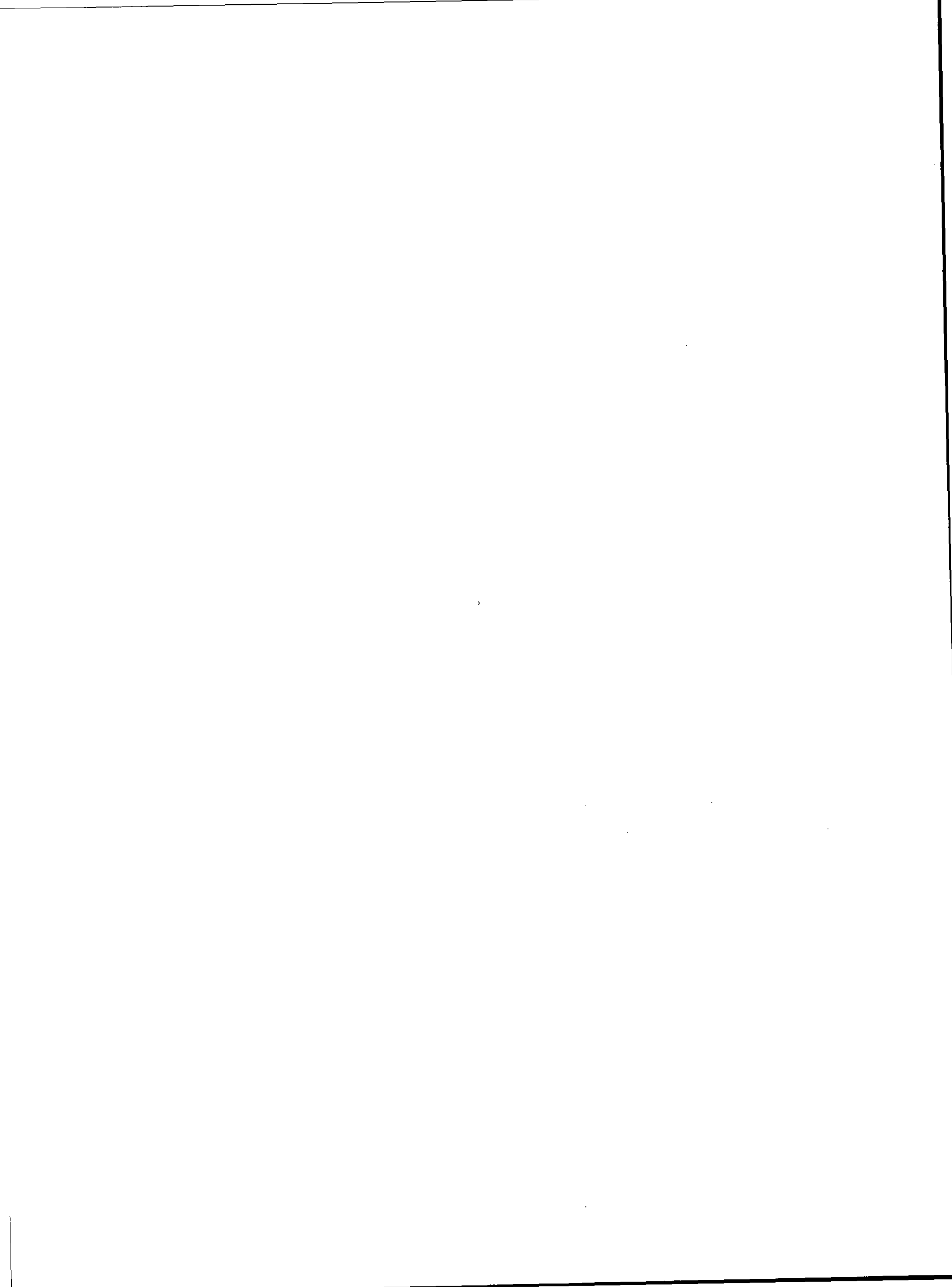
This entire book is recyclable.

Printed in the United States of America

Revision Information for

Exemplar Programming Guide

Edition	Document No.	Description
Third	710-030230-004	Updated to include SPP1600 enhancements, bug fixes. Released with Convex Fortran Version 9.5 and Convex C Version 6.5.
Second	710-030230-003	Updated to include hypernode parallelism, SPP1200 enhancements, bug fixes. Released with Convex Fortran Version 9.3 and Convex C Version 6.3.
First	710-030230-002	First production version of the Exemplar Programming Guide, released with Convex Fortran Version 9.1 and Convex C Version 6.1.



Contents

How to use this guide xxi

Purpose and audience xxi
Scope xxi
Organization xxii
  Suggested reading order xxiii
Notational conventions xxiv
  General conventions xxiv
  Command syntax xxv
Associated documents xxv
Ordering documentation xxvi
Technical assistance xxvi

1 Introduction 1

Scalable parallel processing 1
  SPP vs. traditional vector/parallel architectures 2
    Architectural differences 2
    Memory 3
    Optimizing compilers 4
  SPP vs. clustered workstations 4
    Memory 4
    Optimizing compilers 4
    Interprocess communication 5
    Peripherals 6
    Configurability 6
Exemplar programming model 7
  The shared-memory paradigm 7
  The message-passing paradigm 7
  Message-passing/shared-memory hybrids 8
Overview of Exemplar optimizations 8
  Basic scalar optimizations 8
  Advanced scalar optimizations 9
  Parallelization 10

2 Architecture overview. 11

System organization 11
Memory 14
Physical memory 14
Virtual memory 15
Data caches 16
Cache lines 16
SPP1000 Series cache 17
SPP1200 Series cache 17
SPP1600 Series cache 20
Cache use analysis 21
Data alignment 21
Cache thrashing 22
Cache thrashing: SPP1000 Series 23
Cache thrashing: SPP1200 Series/SPP1600 Series . 27
Interleaving 29
Interleaving example 30
Subcomplexes 33
Physical configuration 33
Subcomplex memory 36

3 Compiler optimizations. 37

Optimization options 37
-no level optimizations 38
Instruction scheduling 38
Span-dependent instructions 38
Register allocation 39
Tree-height reduction 39
Short-circuit evaluation of conditionals in Fortran . . . 41
Data alignment on natural boundaries 42
-O0 Level optimizations 43
Instruction scheduling 43
Redundant-assignment elimination 43
Assignment substitution 44
Common subexpression elimination 44
Redundant-use elimination 45
Constant propagation and folding 45
Algebraic and trigonometric simplification 47

-O1 Level optimizations 48
Constant propagation and folding 48
Redundant-assignment elimination 49
Original code 49
Optimized code 50
Dead-code elimination 52
Copy propagation 52
Common subexpression elimination 53
Code motion 55
Strength reduction 57
Explicit arithmetic reductions 57
Induction variables and constants 57
Global register allocation 59
-O2 Level optimizations 63
Why localize? 64
Strip mining 66
Loop distribution 67
Loop interchange 69
Loop blocking 70
Data reuse 70
Reuse example 71
Blocking example: simple loop 72
Blocking example: matrix multiply 75
Blocking directives, pragmas and options 76
Loop fusion 78
Loop unrolling 80
Loop unroll and jam 83
IF-DO and if-for optimizations 87
Redundant-test elimination 88
Loop boundary-value peeling 89
Test promotion 91
Scalar replacement 93
Inhibitors of localization 94
Loop-carried dependences 94
Aliasing 99
Multiple loop entries or exits 102
RETURN or STOP statements in Fortran 103
Computed or assigned GOTO statements
in Fortran 103
Procedure calls 103
I/O statements 103
Preventing data localization 104

-O3 Level optimizations 105
Basic operation 105
Idle thread states 110
Node-parallelism vs. thread-parallelism 111
Fortran 90 constructs 114
Parallel optimizations 115
Dynamic selection 115
Inhibitors of parallelization 118
Loop-carried dependences 118
Reductions 121
Preventing parallelization 122
Other parallelization directives and pragmas 123

4 Basic shared-memory programming 125

Simple manual loop, task, and region parallelization ... 125
Loop parallelization 126
Combining the attributes 128
Using the attributes 129
prefer_parallel 134
loop_parallel 135
Task parallelization 137
Examples 140
Region parallelization 142
Critical sections 146
-noautopar compiler option 148
-nonodepar compiler option 148
Reentrant compilation 148
Default stack size 149
Loop-specific, task-specific, and
region-specific data privatization 150
loop_private 150
Using loop_private with loop_parallel .. 152
Denoting induction variables in parallel loops ... 153
Privatizing induction variables in nested loops .. 156
task_private 158
parallel_private 160
save_last 163
Performance analysis 165

5 Memory classes 167

Private versus shared memory 168
Memory class addressing 168
thread_private 170
node_private 171
near_shared 171
far_shared 172
block_shared 172
Memory class assignments 173
Static assignments 174
thread_private 175
node_private 177
far_shared 181
block_shared 182
Dynamic assignments 182
Memory class pointers 183
Default classes for dynamic memory 186
thread_private 187
node_private 188
near_shared 191
far_shared 197
block_shared 199

6 Advanced shared-memory programming 205

Parallel information functions 206
Number of processors 206
Number of threads 207
Number of hypernodes 207
Number of threads on current hypernode 208
Thread ID 208
Hypernode ID 209
Level of parallelism 210
Stack memory type 211
Thread IDs and nested parallelism 212
Thread ID assignments 212

Synchronization tools 213
Gates and barriers 213
Synchronization functions 215
Allocation functions 215
Deallocation functions 216
Locking functions 216
Unlocking functions 217
Wait functions 217
`sync_routine` directive and pragma 218
`loop_parallel (ordered)` 221
Critical and ordered sections 223
Synchronizing code 224
Critical sections 224
Ordered sections 228
Limitations 230
Manual synchronization 236
Advanced shared-memory example 246

7 Message-passing programming . . . 251

Overview 251
Approaches to parallelism 252
Message passing on Exemplar systems 252
Convex MPICH 252
PVM/GSM 253
Message-passing programming vs. shared-memory programming 254

8 Limits of optimization 255

Aliasing in Fortran 256
Aliasing in C 257
Worst-case algorithm 257
ANSI algorithm 258
Specifying aliasing modes 259
Iteration and stop values 260
Using potential aliases as addresses of variables . 260
Using hidden aliases as pointers 261
Using a pointer as a loop counter 261
Aliasing stop variables 262
Global variables 263

False cache line sharing 264
Aligning data to avoid false sharing 267
Aligning arrays on cache line boundaries 267
Aligning multidimensional arrays on cache line boundaries 268
Distributing iterations on cache line boundaries 271
Thread-specific array elements 272
Scalars sharing a cache line 274
Working with unaligned arrays 275
Working with dependences 276
Floating-point imprecision 277
Disabling underflow traps 280
Invalid subscripts 280
Misused directives, pragmas, and options 281
Misused memory classes 283
Improper dynamic allocations 283
Incorrect array pointers 286
Hidden dependences 288
Triangular loops 290
Parallelizing the outer loop 292
Parallelizing the inner loop 293
Examining the code 296
Compiler limitations 298
Reductions 298
Evaluation order 300
Incrementing by zero 301
Nondeterminism of parallel execution 303
Large trip counts at -O1 and above 304
Hidden ordered sections 304

9 Potentially unsafe optimizations. . . . 309

Simple strength reduction 309
Code motion 310
Conversion elimination 310

Appendix A: Compiler directives and pragmas 313

Overview 313
Directives and pragmas 314
`align_cti` (*namelist*) 314
`barrier` (*namelist*) 314
`begin_tasks` [(*attribute_list*)] 315
`block_loop[(block_factor=`*n*`)]` 316
`block_shared` (*allocatable_array_namelist*) 316
`critical_section` [(*gate_var*)] 316
`dynsel` [(*trip_count=n*)] 317

end_critical_section 317
end_ordered_section 317
end_parallel 317
end_tasks 318
far_shared(*namelist*) 318
far_shared_pointer(*alloc_var_name*) 318
gate(*namelist*) 318
loop_parallel[(*attribute_list*)] 319
loop_private(*namelist*) 320
near_shared(*namelist*) 320
near_shared_pointer(*alloc_var_name*) 321
next_task 321
no_block_loop 321
no_distribute 321
no_dynsel 322
no_fuse 322
no_loop_dependence(*namelist*) 322
no_parallel 322
no_peel 322
no_promote_test 323
no_side_effects(*funclist*) 323
no_unroll 323
no_unroll_and_jam 323
node_private(*namelist*) 323
node_private_pointer(*alloc_var_name*) 324
opt_level(*level*) 324
ordered_section(*gate_var*) 324
parallel[(*attribute_list*)] 325
parallel_private(*namelist*) 325
peel_all 325
peel 325
prefer_fuse 326
prefer_parallel[(*attribute_list*)] 326
promote_test_all 327
promote_test 327
row_wise(*array_namelist*) 327
save_last 327
scalar 327
sync_routine(*routinelist*) 328
task_private(*namelist*) 328
thread_private(*namelist*) 328
thread_private_pointer(*alloc_var_name*) 328
unroll[(unroll_factor=*n*)] 329
unroll_and_jam[(unroll_factor=*n*)] 329

Appendix B: Optimization options . . . 331

Optimization level and related options 331
Cross compilation options 332
Loop replication options 333
Loop blocking options 334
`IF-DO` and `if-for` optimization options 335
Register use options 336
Data alignment options 337
Using `-align cache` and `-align cache_check`. 338
`-align cache` (Available only in Fortran) 338
`-align cache_check` (Available only in Fortran) 339
Combining `-align` options 340
C aliasing options 341
Other optimization options 344

Appendix C: Optimization report 347

Loop table 348
Supplemental tables 352
Analysis table 352
Test table 353
Privatization table 353
Variable name footnote table 354
Array table 354
Examples 355
Example 1 355
Loop table 357
Analysis table 358
Privatization table 358
Array reference table 358
Example 2 359
Loop table 361
Analysis table 362
Test table 363
Privatization table 363
Array reference table 363

Appendix D: Compiler Parallel Support Library 365

Introduction 365
Symmetric parallelism 365
Asymmetric Parallelism 367
Accessing CPSlib 369
CPS library functions 369
Thread-management functions 370
Symmetric thread functions 370
Asymmetric thread functions 373
Thread information and attribute functions 376
High-level synchronization functions 382
Low-level synchronization functions 385
`sync_routine` directive and pragma 391
Examples 393
Symmetric parallelism 393
Block parallelism 393
Cyclic parallelism 396
Asymmetric parallelism 397
Synchronization using high-level functions 399
Barriers 399
Mutexes 401
Synchronization using low-level functions 403
Critical sections 403
Ordered sections 404

Glossary 409

Index 445

Figures

Figure 1 Exemplar system overview 12
Figure 2 SPP Series functional block 13
Figure 3 SPP1200 Series/SPP1600 Series cache architecture 18
Figure 4 Array layouts—cache-thrashing 24
Figure 5 Array layouts—non-thrashing 26
Figure 6 SPP Series memory interleaving 29
Figure 7 Interleaving of arrays A, B, and C 31
Figure 8 Hypothetical subcomplex configurations 34
Figure 9 Unbalanced tree representation 39
Figure 10 Balanced tree representation 40
Figure 11 Array storage in memory 65
Figure 12 Blocked array access 73
Figure 13 Spatial reuse of A and B 74
Figure 14 LCDs in original and interchanged loops 96
Figure 15 Values read into array A 104
Figure 16 Thread activity: one-dimensional parallelism . . 107
Figure 17 Conceptual strip mine for parallelization 108
Figure 18 Parallelized loop 109
Figure 19 Thread activity: two-dimensional parallelism . . 113
Figure 20 Stride-parallelized loop 130
Figure 21 Virtual addresses for various memory classes . . 168
Figure 22 Physical addresses for various memory classes 169
Figure 23 Ordered parallelization 222
Figure 24 LOOP_PARALLEL(ORDERED) synchronization 230
Figure 25 Data ownership by CHUNK and NTCHUNK blocks 295
Figure 26 Optimization report for Example 1 356
Figure 26 Optimization report for Example 1 (continued). 357
Figure 27 Optimization report for Example 2 360
Figure 27 Optimization report for Example 2 (continued). 361
Figure 28 Symmetric parallelism 366
Figure 29 Asymmetric parallelism 368

Tables

Table 1 Available system configurations 3
Table 2 SPP Series C and Fortran optimization options. . . 37
Table 3 Computation sequence of A(I,J): original loop. 95
Table 4 Computation sequence of A(I,J): interchanged loop 96
Table 5 Pointer class/data class combinations 186
Table 6 Levels of parallelism 210
Table 7 Stack type return values 211
Table 8 Initial mapping of array to cache lines. 265
Table 9 Default distribution of the I loop 266
Table 10 Restructured mapping of array to cache lines . . . 270
Table 11 Optimization report contents. 332
Table 12 Combinations of -align options 340
Table 13 Optimization report contents. 347
Table 14 Reordering transformations reported in opt. report 349
Table 15 Optimizing/special transformations in opt. report 351
Table 16 params->node/PARAMS(1) values 371
Table 17 params->threadscope/PARAMS(4) values. . 371
Table 18 errno values for cps_ppcall and cps_ppcalln. 372
Table 19 errno values for cps_thread_create[n] . . . 374
Table 20 cps_plevel return values 377
Table 21 Accepted CMD/cmd values 381
Table 22 WAIT(2)/wait->wait_attr values. 382

How to use this guide

Purpose and audience

This guide describes efficient methods for shared-memory programming in Convex C and Fortran on Exemplar (also known as SPP Series) computers. The first four chapters cover basic concepts, including automatic optimizations and simple manual optimizations that require minimal programmer intervention. In the following chapters, more progressive topics are covered, including advanced manual optimizations, and the Compiler Parallel Support Library.

The *Exemplar Programming Guide* is for experienced Fortran and C programmers. Readers need not be familiar with the Convex scalable parallel architecture, programming model, or optimization concepts; this book addresses these topics in the necessary detail.

Scope

This guide covers programming methods for Convex Fortran Version 9.5 and Convex C Version 6.5, which run under SPP-UX Version 4.0 or higher. These releases of Convex Fortran and C also require the SPP Series Assembler, Loader and Libraries (ALL) Version 3.8 or higher.

This guide is concerned with producing programs that efficiently exploit the features of both the Exemplar architecture and the SPP Series compilers that run on it. Producing an efficient program requires efficient algorithms and implementation. The techniques of writing an efficient algorithm are beyond the scope of this guide. This guide assumes you have chosen the best possible algorithm for your problem and helps you obtain the best possible performance from that algorithm.

Organization

This document consists of the following chapters:

- Chapter 1, "Introduction," introduces the Exemplar computer, discusses how it differs from other parallel computers, presents its programming model, and provides an overview of the optimizations available using Exemplar compilers.
- Chapter 2, "Architecture overview," presents a programmer's overview of the Exemplar architecture.
- Chapter 3, "Compiler optimizations," discusses optimization levels, options used to specify them, automatic optimizations performed at each level, and directives used to control these optimizations.
- Chapter 4, "Basic shared-memory programming," discusses basic manual optimizations and programming constructs you can use to increase efficiency using shared memory.
- Chapter 5, "Memory classes," discusses the memory classes available for partitioning data on SPP Series machines, and how to use them.
- Chapter 6, "Advanced shared-memory programming," discusses advanced manual optimizations and programming constructs you can use to increase efficiency using shared memory.
- Chapter 7, "Message-passing programming," discusses message-passing programming using PVM/GSM and Convex MPICH on Exemplar machines.
- Chapter 8, "Limits of optimization," discusses common optimization constraints you may encounter and provides suggestions for how to deal with them.
- Chapter 9, "Potentially unsafe optimizations," discusses the aggressive optimizations performed when the `-uo` option is specified.
- Appendix A, "Compiler directives and pragmas," lists and describes compiler directives and pragmas available for use with the SPP Series Fortran and C compilers.
- Appendix B, "Optimization options," lists and describes the optimization options available for use on SPP Series machines.
- Appendix C, "Optimization report," explains the optimization report.
- Appendix D, "Compiler Parallel Support Library," discusses the Compiler Parallel Support library.
- The glossary provides definitions of SPP-related terms.
- An index is included at the end of the document.

Suggested reading order

This book takes a bottom-up approach to presenting information.

- Chapters 1, 2 and 3 provide background information that will help you understand Exemplar architecture and how Exemplar compilers optimize your code.
- Chapter 4 tells you how to derive performance gains with minimal intervention.
- Chapters 5 and 6 explain how to use more advanced programming techniques to further improve performance.
- Chapter 7 discusses message passing on Exemplar machines, which requires even more manual intervention.
- Chapters 8 and 9 tell you about problems you may encounter when using the techniques of the previous chapters and how to enable even more aggressive optimizations.
- The appendixes contain mostly reference information, including a discussion of the Compiler Parallel Support library (CPSlib), which is the most programmer-dependent optimization tool available for SPP Series computers.

If you are interested in a general, comprehensive overview of programming for the Exemplar computer, read the chapters in order.

If you are interested in simply compiling existing programs and getting them to run with minimal effort, start with chapters 3 and 8. Following the cross references that interest you will probably expose you to as much of the rest of the book as is necessary.

If you are interested in getting maximum performance gains for minimum programming effort, read chapters 3 and 4, then proceed if necessary.

If you are willing to spend some time adding directives and rewriting some of your code to realize significant performance benefits (especially if your Exemplar system is equipped with multiple hypernodes), read at least chapters 2 through 6.

If you are interested in running message-passing codes on your Exemplar system, refer to Chapter 7 "Message-passing programming." You may also want to read chapters 2 through 6 of this book to see how the compilers can help you with automatic optimizations.

If you are interested in very low-level control over parallelism using the Compiler Parallel Support library, start with Appendix D. Again, you may want to refer to the other chapters to see how the compiler can help with automatic optimizations.

Notational conventions

This section discusses notational conventions used in this book.

General conventions

In general, the following conventions are used in this guide:

- *Italic*
  - Designates user-supplied variables in a command line or code example
  - Introduces new and important terms
  - Identifies variables in mathematical equations
  - Indicates document titles
- `Constant-width font` designates input and output, including
  - Command names and options
  - System calls
  - Data structures and types
  - Variables and arrays
  - Function and subroutine names
  - Directives, program statements, display examples, printout examples, and error messages returned

  Note that except where noted, the directives and pragmas described in this book can be used with both the C and Fortran compilers. In general discussion, these directives and pragmas are presented in lower case type, but either compiler will recognize them regardless of their case.
- **`Bold constant-width font`** designates text that must be input by the user.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines of code have been left out of an example.

References to man pages appear in the form mnpgname(1), where "mnpgname" is the name of the man page and is followed by its section number enclosed in parentheses. To view this man page, you would type:

```
man 1 mnpgname
```

Note

A Note highlights important supplemental information.

Caution

A Caution highlights information necessary to avoid damage to the system.

Command syntax

Consider this example:

`COMMAND` *input_file* `[...] {a | b}` `[`*output_file*`]`

1. `COMMAND` must be typed as it appears.
2. *input_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional, optional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. [*output_file*] indicates an optional file name.

Associated documents

The Convex Technology Center of Hewlett-Packard provides the following documents to help you use the Fortran and C compilers and associated tools:

- For more information about the Convex Fortran compiler, refer to the *Fortran User's Guide* (DSW-038), *Fortran Language Reference Manual* (DSW-037), and *Release Notice, Fortran Compiler V9.5.*
- For more information about the Convex C compiler, refer to the *C User's Guide* (DSW-086) and *Release Notice, Convex C V6.5.*
- For more information about CXpa, refer to the *CXpa Reference* (DSW-605).
- For more information on the CXdb debugger, refer to the cxdb(1) man page or to the online help available in CXdb.
- For more information about message passing, refer to the *Convex MPICH User's Guide for Exemplar Systems* (DSW-493) and the *PVM/GSM User's Guide for Exemplar Systems* (DSW-501).
- For more information on the Exemplar architecture, refer to the *Exemplar SPP1000 Series Architecture* manual (DHW-014).
- For more information on administering SPP-UX, refer to the *SPP-UX System Administration Guide* (DSW-853).
- For more information on using SPP-UX, refer to the *HP-UX Reference* (Hewlett-Packard order number B2355-90004).

Ordering documentation

To order this document or any other Convex document, send requests to:

Hewlett-Packard Company
Convex Technology Center
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Order documents by title, requesting the most recent edition. In some situations, you may not want the current edition. To receive a specific edition of a manual, contact the local sales office or call the Hewlett-Packard Convex Technical Assistance Center (TAC).

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC):

- Within the continental U.S., use 1-800-952-0379.
- Outside continental U.S., contact the local Convex Technology Center office.

Introduction 1

The distinguishing characteristic of Convex computers has always been their ease-of-use. The Exemplar family of computers brings this ease-of-use legacy to a massively parallel architecture. With the Exemplar family, the full power of massive parallelism can finally be realized without the need to employ complicated and highly platform-dependent programming paradigms. Exemplar compilers generate efficient parallel code with very little intervention on your part; however, you can increase this efficiency by using the techniques discussed in this book.

This chapter provides a general overview of the:

- Exemplar architecture as compared to other parallel architectures
- Exemplar compiler optimizations
- Applicable programming models

Scalable parallel processing

Exemplar systems implement massively parallel processing (MPP) using scalable parallel processing (SPP) technology. Exemplar systems are available in a number of configurations. The SPP1000 Series consists of the compact SPP1000/CD system and the full-size SPP1000/XA system. The SPP1200 Series consists of the compact SPP1200/CD system and the full-size SPP1200/XA system. The SPP1600 Series consists of the compact SPP1600/CD system and the full-size SPP1600/XA system.

As the term SPP implies, scalable parallel machines can be scaled to meet your specific needs. Some MPP systems can only be expanded by doubling the number of processors, or require a large number of processors in their base configurations. Exemplar SPP Series systems can contain as few as 2 processors and as many as 64, and can be scaled by as few as 2 processors. These processors are arranged in 1 to 8 hypernodes, with each hypernode containing 2 to 8 processors.

Processors communicate with each other, with memory and with I/O devices via the Coherent Toroidal Interconnect (CTI). The CTI consists of a nonblocking crossbar on each hypernode for intrahypernode communication, and four high-speed CTI rings that link the hypernodes together for interhypernode communication. The CTI ring design is derived from the IEEE standard 1596-1992, SCI (Scalable Coherent Interface), but this Exemplar implementation sacrifices complete SCI compatibility to provide lower latencies.

SPP Series memory is also easily scalable; each hypernode can support 128 Mbytes, 256Mbytes, 512 Mbytes, 1 Gbyte or 2 Gbytes of physical memory. This allows for a maximum of 16 Gbytes on a fully-configured 8-hypernode system. Each process can access its full 32-bit (4-Gbyte) virtual address space, and, if necessary, programs can be written in a manner that allows them to surpass the 4-Gbyte limit and access all the memory on a system.

SPP vs. traditional vector/parallel architectures

Scalable parallel processing represents a departure from traditional vector/parallel supercomputers like the Convex C Series. The C Series architecture is used to illustrate the difference between traditional and SPP architectures below, but the same differences apply in principle to all vector/parallel machines.

Architectural differences

Convex C Series machines contain a limited number (1-8) of custom processors connected by a high-speed crossbar to a large, shared memory. For connecting small numbers of processors such as these to memory, crossbars are cost-effective and fast, allowing all processors to access all memory with equally high speed. Each processor is equipped with one or more vector units that speed loop computations involving arrays by performing array arithmetic on up to 128 elements per vector instruction. Machines containing multiple CPUs can further reduce time-to-solution by adding parallelism at the process, loop, and task level. The SPP Series architecture takes a different approach. Rather than using vector units to exploit fine-grain parallelism, the SPP processors speed scalar processing by using a reduced set of high-speed instructions coupled with pipelining, high-speed instruction and data caches, and a large register set. See Table 1 for processor configurations in SPP Series machines.

Table 1 Available system configurations

System	Microprocessor	Number of microprocessors	Number of hypernodes
SPP1000/CD	HP PA-RISC 7100	2, 4, 8, 12, or 16	1 or 2
SPP1000/XA	HP PA-RISC 7100	4 to 64 (by multiples of 4)	1 to 8
SPP1200/CD	HP PA-RISC 7200	2, 4, 8, 12, or 16	1 or 2
SPP1200/XA	HP PA-RISC 7200	4 to 64 (by multiples of 4)	1 to 8
SPP1600/CD	HP PA-RISC 7200 with enhanced cache*	2, 4, 8, 12, or 16	1 or 2
SPP1600/XA	HP PA-RISC 7200 with enhanced cache*	4 to 64 (by multiples of 4)	1 to 8

*The 7200 microprocessor with enhanced cache differs from the 7200 microprocessor in that the main cache is 1 Mbyte rather than 256 kbyte.

Two-dimensional parallelism, which can benefit nested parallel structures, is also possible on multihypernode SPP Series machines. Rather than implementing the first dimension in the vector unit and the second across processors (as in C Series), the SPP Series can implement the first level within a hypernode and the second across hypernodes. Single-dimensional parallelism that spans hypernodes can also be implemented.

Memory

Because of the potentially large number of processors available on a multihypernode SPP Series system, memory access via a system-wide crossbar is not practical. Instead, low latency, high-bandwidth memory access is provided by globally shared memory (GSM). In this model, physical memory is distributed among all hypernodes, and the entire virtual address space of a process is accessible by every processor. Processors within a hypernode can access hypernode-local memory via the crossbar regardless of whether the address space is on one or more hypernodes; memory in another hypernode can be accessed via the CTI rings. Of course, interhypernode accesses take longer than intrahypernode accesses. However, part of every hypernode's memory is dedicated to act as a CTIcache, which holds copies of commonly used data from other hypernodes. These CTIcaches and the processor caches are *coherent*, meaning that when a thread references a data item via its virtual address, the value it receives will be the most recently-assigned value. By holding frequently-referenced data close to its referencing processes, regardless of the actual memory location of the data, these caches provide excellent data distribution.

Optimizing compilers

Programs that optimize well on traditional vector/parallel machines will optimize well on the SPP Series with little manual intervention. SPP Series compilers automatically exploit opportunities for parallelism and data localization in programs written for shared-memory machines. Chapters 3 through 6 discuss manual optimizations that can yield even more performance from such programs.

SPP vs. clustered workstations

While the SPP Series architectures use the same processors found in HP workstations, the architectures' low-latency GSM, automatic optimizing compilers, high-speed interconnections, shared peripherals, and user-configurability sharply distinguish SPP computers from clustered workstations. The following subsections discuss each distinguishing feature in detail.

Memory

Each workstation in a cluster has its own private memory; there is no shared memory. This means that any data that must be shared among processors must be passed over the low-performance network that connects them. While an Exemplar computer can support this method of programming, it offers the many advantages of GSM, as described in the "SPP vs. traditional vector/parallel architectures" section.

Many workstation operating systems reserve a large amount of memory for system use, restricting user processes to what is left. SPP-UX requires only a small fraction of each processor's memory, leaving a large majority of it for user processes, whether they are using GSM or message passing.

Optimizing compilers

Programs for clustered workstations are compiled using the workstations' compilers. If the cluster contains workstations that require different executables (i.e., if it is a *heterogeneous* cluster), the programmer must generate the executables using the proper compiler. Homogeneous clusters eliminate this requirement, but automatic parallelization is nevertheless unavailable on any type of cluster. The compilers used may generate efficient code for each processor, but any parallelism or process coordination must be explicitly implemented by the programmer via message passing.

Convex compilers have long been highly regarded for their ability to automatically optimize code. Exemplar compilers continue this

legacy and add to it fully automatic parallelism and several new data localization optimizations designed to improve memory usage and aid parallelization. Additionally, a rich set of directives allows you to further enhance the automatic optimizations performed on your shared-memory program.

Exemplar compilers give the highest performance—with little or no programmer intervention—from generic programs that exploit GSM. Message-passing programs, with their parallelism explicitly coded, also benefit from all Exemplar compiler optimizations.

Interprocess communication

To communicate among themselves or access each other's data, the processors in a cluster of workstations must communicate over low-performance networks and access distributed memory. Communication can be handled only by passing explicit messages between workstations over the network; because of the distributed memory and absence of parallelizing compilers, programmers must explicitly code parallelism. Parallel tasks running on clusters, then, must be fairly autonomous to avoid wasting time waiting for data or synchronization instructions to travel over the network. Clusters are best suited to coarse-grained parallelism, such as that possible at the process level, or to manually-parallelizable algorithms that contain a large ratio of computation to communication. In these cases, task chunks or processes and their data are parcelled out to underused workstations, run to completion, and the results are sent back to the parent.

Fine-grained, loop-level parallelism is difficult to efficiently perform on clusters because of the need for frequent data accesses and synchronization.

Exemplar systems are suitable for both coarse- and fine-grained parallelism. Programs containing potential parallelism, when compiled with Exemplar compilers, automatically exploit the parallelism, spawning threads to run on as many processors as are available and rejoining these threads upon completion. This fine-grained parallelism takes full advantage of the fully coherent memory caches and high-speed interconnects available on an Exemplar system.

While message passing is supported and can be used to speed certain applications (refer to Chapter 7, "Message-passing programming"), with GSM, it is not necessary for most programs. When message passing is used on an Exemplar machine, the high-speed interconnects can give a substantial performance increase over traditional networks. This makes message-passing programs that exploit finer-grained parallelism practical.

SPP-UX automatically schedules threads within a hypernode to execute on idle and underused processors as necessary, ensuring a balanced machine load and exploiting both thread- and process-level parallelism.

Peripherals

Peripheral devices connected to an Exemplar system can be accessed from any processor on the machine. On clustered workstations, peripherals are processor-dependent. Programs running on Exemplar systems therefore have access to potentially greater mass storage space.

Configurability

Exemplar systems can be configured by system administrators into one or more *subcomplexes*. A subcomplex is a collection of a specified number of processors on specified hypernodes with global memory. Subcomplexes allow Exemplar systems to be configured into several logical entities that make the most effective use of available resources given the computational tasks at hand.

In terms of configuring hardware, adding processors to a cluster can actually degrade performance because of the low-performance network and private memory. The network can present a bottleneck when parallelism increases to exploit the new processors; to overcome this, coarser granularity can be used—and this can require more private memory than the processors can address. The absolute performance of an Exemplar system, on the other hand, increases unhindered by a traditional network or private-memory limits. Adding peripherals and memory to an Exemplar system can also provide improved absolute performance, because all processors can access both, whereas memory and peripherals are processor-specific on clusters.

Exemplar programming model

The Exemplar programming model provides three perspectives from which a programmer can write (or adapt) code to run on an Exemplar system. Those perspectives are the shared-memory, message-passing, and shared-memory/message-passing hybrid paradigms. This book focuses on using the shared-memory paradigm but also provides some information on the other two paradigms.

The shared-memory paradigm

In the shared-memory paradigm, the compilers handle optimizations, and, if requested, parallelization. Data distribution is taken care of by hardware via the CTIcache. Numerous compiler directives and pragmas (discussed in detail in Chapter 4, "Basic shared-memory programming," and Chapter 5, "Memory classes," and listed in Appendix A, "Compiler directives and pragmas") are available to further increase optimization opportunities. Because the shared-memory approach is used on most non-MPP machines, the support of it on SPP Series machines allows programs from such machines to be easily ported to the Exemplar architecture. Most programmers also use the familiar shared-memory paradigm for programs written from scratch for Exemplar machines, because it automates many of the tasks for which the message-passing paradigm requires explicit coding.

Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," cover shared-memory programming in detail.

The message-passing paradigm

The Exemplar message-passing paradigm supports Convex implementations of the MPI and PVM standards. These versions are referred to as Convex MPICH and PVM/GSM (Parallel Virtual Machine for Globally Shared Memory).

Under the message-passing paradigm, functions are used to explicitly spawn parallel processes, share data among them, and coordinate their activities. There is no shared memory; each process has its own private 4-Gbyte address space (this is determined by the HP 7x00's 32-bit address space) and any data that must be shared must be explicitly passed between processes.

Support of message passing allows programs written under this paradigm for distributed-memory machines to be easily ported to Exemplar computers. Programmers familiar with message

passing may choose to write new Exemplar programs using this paradigm rather than shared memory and can realize a substantial performance boost over conventional message-passing machines, even when coding finer-grain parallelism. The few programs that require more per-process memory than possible using shared memory will benefit from the manually-tuned message-passing style.

For more information, see Chapter 7, "Message-passing programming" or the books *Convex MPICH User's Guide for Exemplar Systems* and *PVM/GSM User's Guide for Exemplar Systems.*

Message-passing/shared-memory hybrids

Some programs may benefit from combining the paradigms to allow several shared-memory processes to coordinate their activities via message passing. This model allows the majority of the program to be written in the familiar shared-memory style while the process-private memory benefits of message passing are exploited.

Overview of Exemplar optimizations

Exemplar compilers perform a broad range of user-selectable optimizations. These optimizations, which are specified via compiler command-line options, are briefly introduced here. A more thorough discussion, including the options associated with each, is given in Chapter 3, "Compiler optimizations."

Basic scalar optimizations

Basic scalar optimizations improve performance at the basic block and program unit level.

A basic block is a sequence of statements that has a single entry point and a single exit. Branches do not exist within the body of a basic block. A program unit is a subroutine, function, or `main` program in Fortran or a function (including `main`) in C; program units are also often generically referred to as procedures. Basic blocks are contained within program units; program unit-level optimizations span basic blocks.

To improve performance, basic scalar optimizations:

- Fully exploit the processor's functional units and registers
- Reduce the number of times memory is accessed
- Simplify expressions
- Eliminate redundant operations
- Replace variables with constants
- Replace slow operations with faster equivalents

Advanced scalar optimizations

Advanced scalar optimizations are primarily intended to maximize processor data cache usage. This is referred to as data localization. Concentrating on loops, these optimizations strive to encache the data most frequently used by the loop and keep it encached so as to avoid costly main memory accesses.

The compilers can perform a number of transformations on loops to facilitate more efficient data localization. The most fundamental of these transformations is *blocking*. Blocking breaks a loop whose data is too large to fit entirely in the processor data cache into a loop nest; the inner loop can then make efficient use of the cache while the outer loop runs the inner loop as many times as necessary to cover the entire original trip count.

Advanced scalar optimizations include several other loop transformations; many of them either facilitate more efficient strip mining or are performed on strip mined loops to optimize processor data cache usage. All of these optimizations are covered in detail in Chapter 3, "Compiler optimizations."

Advanced scalar optimizations implicitly include all basic scalar optimizations.

Parallelization

It is through parallelization that you can realize the full power of a scalable parallel computer like the Exemplar computer. Parallelization allows a program to be executed by as many processors as are available within its subcomplex, in most cases dramatically reducing time-to-solution. Exemplar compilers can automatically locate and exploit loop-level parallelism in most programs, and, using the techniques described in Chapter 5, "Memory classes," you can assist the compilers in finding even more parallelism in your programs.

Loops that have been data-localized are prime candidates for parallelization; individual iterations of inner loops that contain strips of localizable data can be parcelled out among several processors and run simultaneously. The maximum number of processors that can be used is limited by the number of iterations of the outer loop, and, of course, by processor availability.

While most parallelization is done on nested, data-localized loops, other code can also be parallelized. For example, through the use of manually-inserted compiler directives, sections of code outside of loops can also be parallelized.

Parallelization optimizations implicitly include all scalar optimizations.

Architecture overview

2

This chapter provides an overview of the Exemplar system architectures for programmers. The SPP1000 Series, SPP1200 Series, and the SPP1600 Series computers are covered. For more detailed information on the topics discussed in this chapter, refer to the *Exemplar SPP1000-Series Architecture* manual, order number DHW-014.

System organization

Think of an Exemplar system as a shared-memory computer with two levels of memory latency. Memory available on the current hypernode (accessed through the crossbar) constitutes the first level, and all other memory (accessed through the CTI rings) constitutes the second. Exemplar SPP1000/XA, SPP1200/XA, and SPP1600/XA systems consist of 1 to 8 hypernodes. Each hypernode contains 4 or 8 processors and 256 Mbytes, 512 Mbytes, 1 Gbyte or 2 Gbytes of physical memory. Processors are arranged in functional blocks, each containing two processors, 128 Mbytes to 512 Mbytes of memory, and some control devices. Functional blocks within a hypernode communicate with each other, with memory, and with peripherals via a 5 port nonblocking crossbar. Functional blocks communicate across hypernodes via four CTI rings.

Exemplar SPP1000/CD, SPP1200/CD, and SPP1600/CD systems consist of 1 or 2 hypernodes, each containing 2, 4 or 8 processors, but are otherwise identical to their respective /XA systems.

Figure 1 shows an overview of a multihypernode Exemplar system containing 8 processors per hypernode.

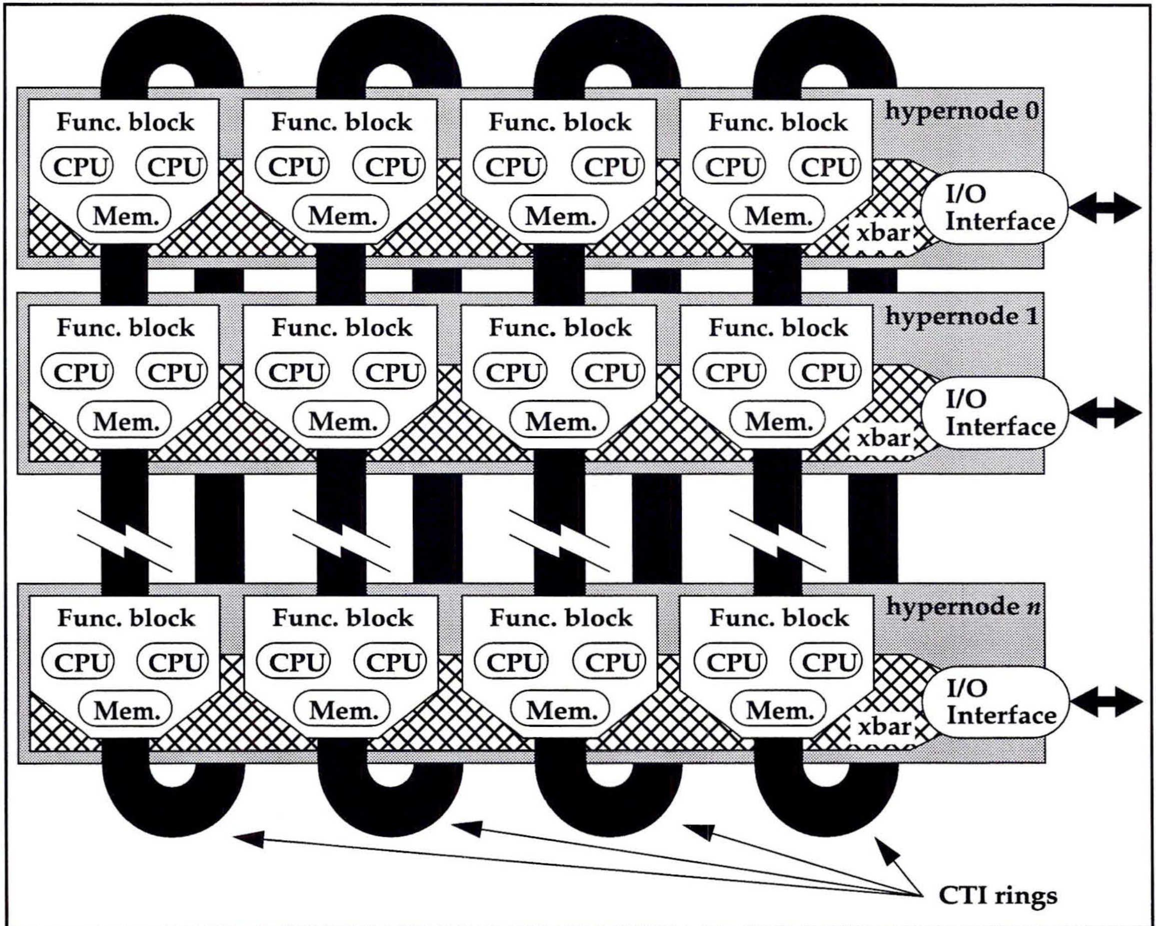


Figure 1 Exemplar system overview

Though it is difficult to illustrate in two dimensions, hypernodes are distributed evenly around the CTI rings. While the rings attach specific functional blocks within the hypernode as illustrated, any functional block can access memory on any other functional block by routing its request through its own crossbar to the functional block that is attached to the correct ring. Data is returned via the same ring and then routed via the crossbar back to the requesting functional block.

Figure 2 shows a single functional block in more detail.

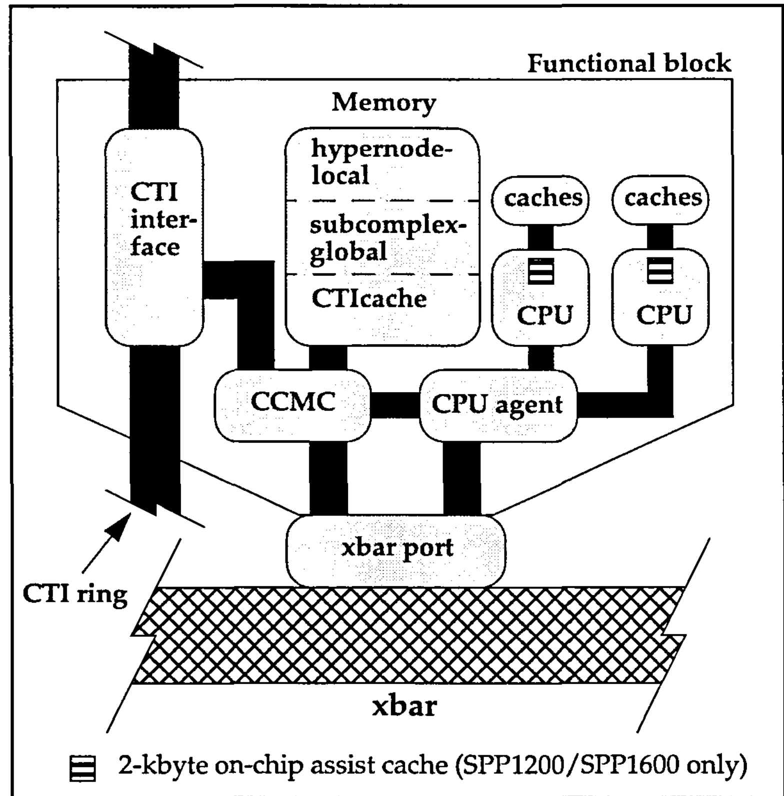


Figure 2 SPP Series functional block

CPUs communicate directly with their own instruction and data caches, which can be accessed by the CPU in one clock (assuming a full pipeline). Caches vary according to the list below:

- SPP1000 Series machines use 1-Mbyte off-chip instruction and data caches
- SPP1200 Series machines use 256-kbyte off-chip instruction and data caches, and a 2-kbyte on-chip assist data cache (both SPP1200 Series caches can be accessed in one clock).
- SPP1600 Series machines use 1-Mbyte off-chip instruction and data caches, and a 2-kbyte on-chip assist data cache (both SPP1600 Series caches can be accessed in one clock).

The functional block's two CPUs communicate with the rest of the machine through the CPU agent. The Convex Coherent Memory Controller (CCMC) provides the interface between the functional block's 2 memory banks and the rest of the machine. All intrahypernode memory accesses take approximately 550-600 nanoseconds, even if they can be fulfilled from the functional block's own memory block. This is because they must traverse the crossbar, which gives equal access to all hypernode memory from all functional blocks.

Memory

Each shared memory process running on an SPP Series system accesses its own 4-Gbyte virtual address space. Of this space, approximately 3.7 Gbytes are available to hold program text, data, and the stack; the remaining space is used by SPP-UX. The stack size is tunable; refer to the section "Default stack size" on page 149 for more information.

Processes cannot access each other's virtual address space. This virtual memory maps to the physical memory of the subcomplex on which the process is running. Subcomplexes cannot access each other's physical memory, just as processes cannot access each other's virtual memory.

Physical memory

All memory (excluding processor caches) on SPP Series systems is implemented in the memory banks of the functional blocks. Each functional block contains 2 memory banks, for a total of 8 banks per 8-CPU hypernode. As shown in Figure 2, this memory is typically partitioned (by the system administrator) into hypernode-local, subcomplex-global, and CTIcache. It is also interleaved as described in the "Interleaving" section later in this chapter.

Hypernode-local memory, as its name implies, is local to its hypernode, and cannot be accessed by other hypernodes. This is where application and SPP-UX executables, as well as user process data that has been explicitly declared private, reside.

Subcomplex-global memory is accessible by all CPUs in a given subcomplex. This memory can be allocated as discussed in the section "Subcomplexes" on page 33.

The CTIcache is used to store copies of global data fetched from other hypernodes.

Virtual memory

Virtual memory is divided into five classes. The compilers choose default classes to provide your programs with normal SMP memory-transaction semantics; programmers can also manually assign data to memory classes to improve data distribution and further increase performance. However, doing so also requires some other aspects of optimization, particularly loop parallelization, to be handled manually.

Brief descriptions of the virtual memory classes and their physical memory mappings follow:

`thread_private`

This memory is private to each thread of a process. A `thread_private` data object has a unique virtual address for each thread within its hypernode. These addresses map to unique physical addresses in hypernode-local physical memory on each hypernode. Threads access the physical copies of `thread_private` data residing on their own hypernode when they access `thread_private` virtual addresses.

`node_private`

This memory is shared among the threads running on a given hypernode but is inaccessible from other hypernodes. A `node_private` data object has a unique virtual address by which all threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode.

`near_shared`

Data objects of the `near_shared` class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. All data of a `near_shared` object maps to physical addresses on a particular hypernode.

`far_shared`

Data objects of the `far_shared` class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. Physically, `far_shared` data is distributed by pages, in a manner that is approximately round-robin, to all the hypernodes in the subcomplex, so the virtual address maps to a single physical address located on one of the hypernodes.

block_shared

Data objects of the block_shared class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. Physically, block_shared data is distributed in blocks equally among the hypernodes on which the process is executing (which could be a subset of the subcomplex), one block per hypernode. block_shared memory must be dynamically allocated; the programmer can then easily ensure that threads on a hypernode make most of their accesses to the block residing on their hypernode.

Using these memory classes is discussed in detail in Chapter 5, "Memory classes."

Data caches

SPP1000 Series, SPP1200 Series, and SPP1600 Series machines each use high-speed data caches to improve performance, but the architectures differ in their implementations of the cache.

Cache lines

Before examining the specifics of each architecture's caches, you must understand how data is moved between the cache and memory. A *cache line* describes the size of a chunk of contiguous data that must be copied into or out of a cache in one operation. When a processor experiences a cache miss—that is, requests data that is not already encached—the cache line containing the address of the requested data is moved to the cache. This cache line also contains some number of other data objects which were not specifically requested; this number varies according to the object size and the type of cache line in question.

A *CTIcache line* is 64 bytes long, so 64 bytes of data move from globally shared memory to the CTIcache when a CTIcache miss occurs. A CTIcache line consists of two contiguous *processor cache lines*, which are 32 bytes long. When a processor cache miss occurs, the requested data is fetched as part of a contiguous 32-byte cache line. If this data resides in any memory on the processor's hypernode, it need not traverse the CTIcache; if it resides in the memory of another hypernode, it will be fetched through the CTIcache.

All processor-encached data not residing on the processor's hypernode must pass through the CTIcache, so if this data is contained in processor cache, it is replicated in the CTIcache.

A primary reason cache lines are employed is to allow for *data reuse*. Data in a cache line is subject to reuse if, while the line is encached, any of the data elements contained in the line besides the requested element are referenced by the program, or if the requested element is referenced more than once.

Because data can only be moved to and from memory as part of a cache line, both load and store operations cause their operands to be encached. Cache coherency hardware invalidates cache lines when they are stored to by a particular processor. This indicates to other processors, which need the data the cache lines contain, that they must load the cache line from the processor that most recently wrote to that particular cache line.

SPP1000 Series cache

SPP1000 Series machines use 1-Mbyte write-back direct-mapped data caches. In a direct-mapped cache, the cache address for a given data object is a function of the object's full virtual address. On the SPP1000 Series, cache addresses are computed within a process using the following formula:

cache_address = `MOD` (*virtual_address*, 2^{20})

Where the `MOD` function yields the remainder when *virtual_address* is divided by 2^{20}. The value of 2^{20} is 1,048,576, or 1 Mbyte. Thus, a data object's cache address is the least-significant 20 bits of its virtual address.

This addressing scheme can result in *cache thrashing*, which is discussed in the "Cache thrashing: SPP1000 Series" section.

SPP1200 Series cache

The SPP1200 Series architecture uses two data caches—a 256-kbyte off-chip main cache, and a 2-kbyte on-chip assist cache.

The 256-kbyte off-chip write-back direct-mapped data cache is, except for size, identical to the off-chip data cache used in the SPP1000 Series architecture. Cache addresses in the SPP1200 Series main cache are determined in the same manner as for the SPP1000 Series except that the least-significant 18 bits of the data's virtual address (2^{18} = 262,144 or 256 kbytes) are used rather than the least-significant 20 bits.

The 2-kbyte on-chip assist cache is fully associative. In a fully-associative cache, there is no correlation between a data item's address in memory and its address in the cache, so an item can never be displaced due to cache thrashing. A data item's address is determined by a hashing algorithm, and all cache addresses are used to store data. No data is displaced from the assist cache until it is full, and in that case, the algorithm will displace the data item that was fetched least recently (i.e., the algorithm implements a FIFO queue). Data displaced from the assist cache is moved to the main cache; all data fetched from memory enters the assist cache first and only enters the main cache if it is displaced from the assist cache.

The assist cache holds 64 processor cache lines (64x32 bytes = 2048 bytes, or 2 kbytes).

The SPP1200 Series/SPP1600 Series cache architecture is illustrated in Figure 3.

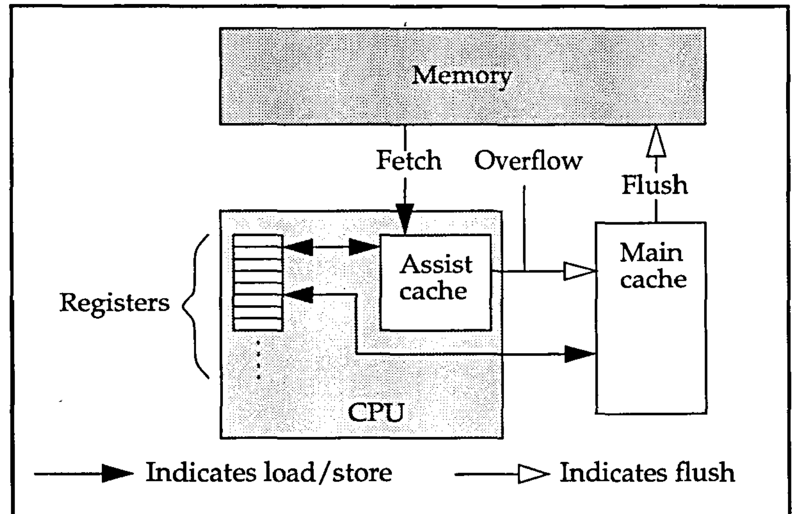


Figure 3 SPP1200 Series/SPP1600 Series cache architecture

Keep in mind that both load and store operations require the cache line containing the data being loaded or stored to be encached in either the assist cache or the main cache. The term *reference*, as used in the following discussion, refers to either a load or store.

When a data item is referenced, all 64 assist cache lines plus one main cache line (the line containing the appropriate virtual address) are checked simultaneously. For a load, if the data resides in either cache, it is loaded directly from the cache to a register in one clock cycle (assuming a full pipeline). Data from the main cache cannot pass through the assist cache in this case; the same data item can never exist in both the assist cache and the main cache. However, for any reference, if the data does not reside in either cache, it must be fetched from main memory (or the CTIcache). In this case, data satisfying a load is always copied into the assist cache on its way to a register.

There is no path from memory directly to the main cache. All entries in the main cache pass through the assist cache when initially fetched; the main cache only receives data displaced from a full assist cache.

When a store executes, the data is written to whichever cache its cache line resides on. This does not necessarily imply that data is stored to the cache from which it was loaded; it is possible that between the load and store operations a cache line residing in the assist cache may be flushed to the main cache, or that a line in the main cache may be invalidated by another thread and reloaded from memory into the assist cache.

Data is stored to memory only when it is flushed from the main cache due to a collision or due to another processor loading it. This happens when the least-significant 18 bits of the address of a data item being written to the main cache is identical to the least-significant 18 bits of a data item already resident in the main cache. In this case the resident item is replaced by the new item in the cache. If the resident cache line has been modified, it is written to memory as part of the replacement process.

If an item being stored does not reside in either cache, the appropriate cache line is reloaded into the assist cache (as with a load), and the data is written there.

Prefetching

When a cache line is fetched or accessed on an SPP1200 Series/SPP1600 Series machine, hardware in the CPU can predict which cache line is likely to be needed next based on loop stride and direction and, if the predicted line is within the same page, *prefetch* it. This means that on an initial fetch from memory, two cache lines are actually fetched—one containing the data needed for the current load or store, and one containing data likely to be fetched next, even if that data is not residing in an adjoining cache line.

A *prefetched cache line* is, by definition, an as-yet-unused cache line residing in the cache as a result of another line being fetched. It is possible that a cache line will be prefetched into the assist cache and eventually be displaced to the main cache without ever being used; this is the only way that unused data can make it into the main cache.

Prefetching takes place every time data is fetched from memory and every time a prefetched cache line is accessed in the assist cache. When an encached prefetched line is accessed, if the cache line likely to be needed next is not already encached, it is prefetched. If the CPU cannot predict which cache line will be needed next but knows the direction in which data is being accessed, it generally prefetches the cache line that is contiguous to the fetched line in the appropriate direction. The only time no prefetching occurs is when both the requested cache line and the predicted next cache line are already encached or the prefetch crosses a page boundary.

The actual prefetching operation does not execute simultaneously with the fetch, but immediately thereafter, and it effectively executes simultaneously with the instructions executing on the CPU. Up to four prefetches can be outstanding.

SPP1600 Series cache

The SPP1600 Series architecture uses two data caches—a 1-Mbyte write-back direct-mapped off-chip main cache, and a 2-kbyte on-chip assist cache. This cache scheme is the same as the SPP1200 Series except that the main cache is 1 Mbyte instead of 256 kbyte. Any discussion of the SPP1200 Series architecture applies to the SPP1600 Series architecture except for the size of the main cache. For example, cache addresses in the SPP1600 Series main cache are determined in the same manner as for the SPP1000 Series (which is different from the SPP1200 Series), using the least-significant 20 bits of the data's virtual address. Refer to the "SPP1200 Series cache" section for more information.

Cache use analysis

CXpa, the Convex visual profiler, can be used to analyze cache performance. See the section "Performance analysis" on page 165 for more information on the types of factors that CXpa tracks. You can also refer to the *CXpa Reference* (DSW-605), the cxpa(1) man page, or contact your sales representative. CXpa is an optional product.

Data alignment

Aligning data addresses on cache line boundaries allows for efficient data reuse in loops, especially when the compiler performs the blocking optimization (refer to Chapter 3, "Compiler optimizations"). You can align data on CTIcache boundaries automatically by:

- Using uninitialized Fortran `COMMON` blocks (blocks with no `DATA` statements) that are at least 64 bytes.
- Using Fortran `ALLOCATE` statements. (Applies only to parallel executables.)
- Using the C functions `malloc` or `memory_class_malloc`. (Applies only to parallel executables.)
- Using uninitialized global arrays or structs in C that are at least 32 bytes.
- Using uninitialized data of the C `external` storage class that is at least 32 bytes.

Only the first item in a list of data objects appearing in any of these statements is aligned. To make most efficient use of available memory, the total size, in bytes, of any array appearing in one of these statements should be an integral multiple of 64 (the size of a CTIcache line, in bytes). Sizing your arrays this way prevents data following the first array from becoming misaligned. Scalar variables should be listed after arrays, and ordered from longest data type to shortest (e.g., `REAL*8` scalars should precede `REAL*4` scalars).

Note

Aliases can inhibit data alignment. Be especially careful when equivalencing arrays in Fortran.

You can force CTIcache boundary alignment for specific scalar variables or arrays by using the `align_cti` directive or pragma. The Fortran directive has the following form:

```
C$DIR ALIGN_CTI(namelist)
```

In C it has the following form:

```
#pragma _CNX align_cti(namelist)
```

Where *namelist* is a list of arrays and/or scalars that will be aligned on CTIcache boundaries.

You can specify CTIcache boundary alignment for all arrays (but not scalars) in your program by using the `-align cti` compiler option.

You can specify processor cache boundary alignment in Fortran for all arrays in your program by the using the `-align cache` compiler option.

See the section "Data alignment options" on page 337 for more information on available `-align` options.

Cache thrashing

Cache thrashing occurs when two or more data items that are needed by the program both map to the same cache address. Each time one of the items is encached, it overwrites another needed item, causing cache misses and impairing data reuse. The sections "Cache thrashing: SPP1000 Series" on page 23 and "Cache thrashing: SPP1200 Series/SPP1600 Series" on page 27 explain how thrashing happens on SPP Series machines.

A type of thrashing known as *false cache line sharing* is discussed in the section "False cache line sharing" on page 264.

Cache thrashing: SPP1000 Series

SPP1000 Series machines use a 1-Mbyte direct-mapped data cache. In a direct-mapped cache, unique data objects can have identical cache addresses. On SPP1000 Series machines, objects whose virtual addresses are identical in their least-significant 20 bits have identical cache addresses. If such data objects are referenced from within the same loop, they will overwrite each other in the cache, generally resulting in constant cache misses, and causing unnecessary and costly main-memory accesses. This condition is known as *cache thrashing*.

Cache thrashing can become a problem on SPP1000 Series machines when two encachable data objects are exactly a multiple of 1 Mbyte apart in memory. To eliminate the problem, you must ensure that your data is not spaced this way.

Consider the following Fortran example:

```
REAL*8 ORIG(65536), NEW(65536), DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
DO I = 1, N
  NEW(I) = ORIG(I) + DISP(I)
ENDDO
```

In this example, the arrays `ORIG` and `DISP` overwrite each other in a 1 Mbyte cache. Because the arrays are in a `COMMON` block, we know that they will be allocated in contiguous memory in the order shown. Each array element occupies 8 bytes, so each array occupies 0.5 Mbyte ($8 \times 65536 = 524288$ bytes); therefore arrays `ORIG` and `DISP` are exactly 1 Mbyte apart in memory, and all their elements have identical cache addresses. The layout of the arrays in memory and in the data cache is shown in Figure 4.

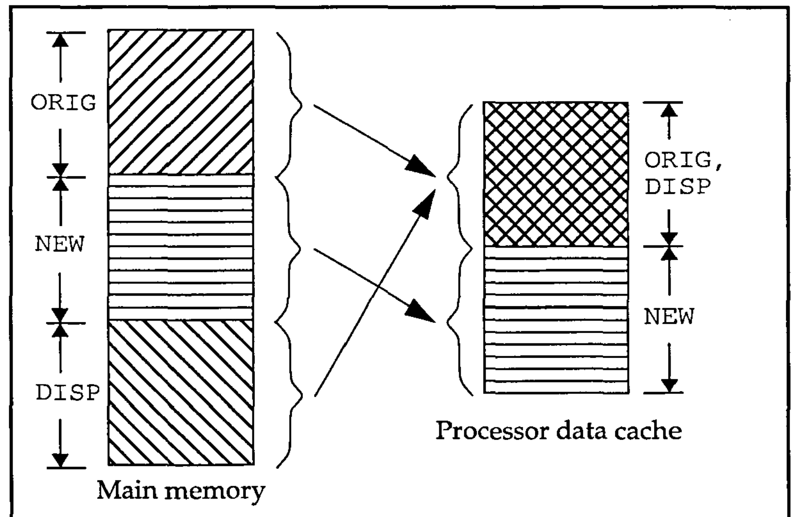


Figure 4 Array layouts—cache-thrashing

When the addition in the body of the loop executes, the current elements of both `ORIG` and `DISP` must be fetched from main memory into the cache. Because these elements have identical cache addresses, whichever is fetched last will overwrite the first. Remember that processor cache data is fetched 32 bytes at a time; to efficiently execute a loop such as this, the unused elements in the fetched cache line (3 extra `REAL*8` elements are fetched in this case) must remain encached until they can be used in subsequent iterations of the loop. Because `ORIG` and `DISP` thrash each other, this reuse is never possible; every cache line of `ORIG` that is fetched is overwritten by the cache line of `DISP` that is subsequently fetched, and vice versa. The cache line is overwritten on every iteration; typically in a loop like this, it would not be overwritten until all of its elements were used.

Because main memory accesses take substantially longer than cache accesses, this severely degrades performance. Even if the overwriting involved the `NEW` array, which is stored rather than loaded on each iteration, thrashing would occur, because stores overwrite entire cache lines the same way loads do.

But the problem is easily fixed by increasing the distance between the arrays. This can be done by either increasing the array sizes or inserting a padding array.

The following example illustrates the padding approach:

```
REAL*8 ORIG(65536), NEW(65536), P(8),DISP(65536)
COMMON /BLK1/ ORIG, NEW, P, DISP
.
.
.
```

Here, the array P(8) moves DISP 64 bytes further from ORIG in memory. Now no two elements of the same index share an SPP1000 Series cache address, and for the given loop, this postpones cache overwriting until the entire current cache line is completely exploited. P is 8 elements, or 64 bytes, which prevents CTIcache as well as processor cache thrashing; a size of 32 bytes would work to prevent processor cache thrashing, but there is a slim chance CTIcache thrashing could still occur because a CTIcache line is 64 bytes.

The alternate approach involves increasing the size of ORIG or NEW by 8 elements (64 bytes), as shown in the following example:

```
REAL*8 ORIG(65536), NEW(65544), DISP(65536)
COMMON /BLK1/ ORIG, NEW, DISP
.
.
.
```

Here, NEW has been increased by 8 elements, providing the padding necessary to prevent ORIG from sharing cache addresses with DISP. Figure 5 shows how both solutions prevent thrashing. Note that the pad never gets encached, whether it consists of the P array or an extended NEW array, because it is never referenced. If it was referenced, it would not solve the thrashing problem.

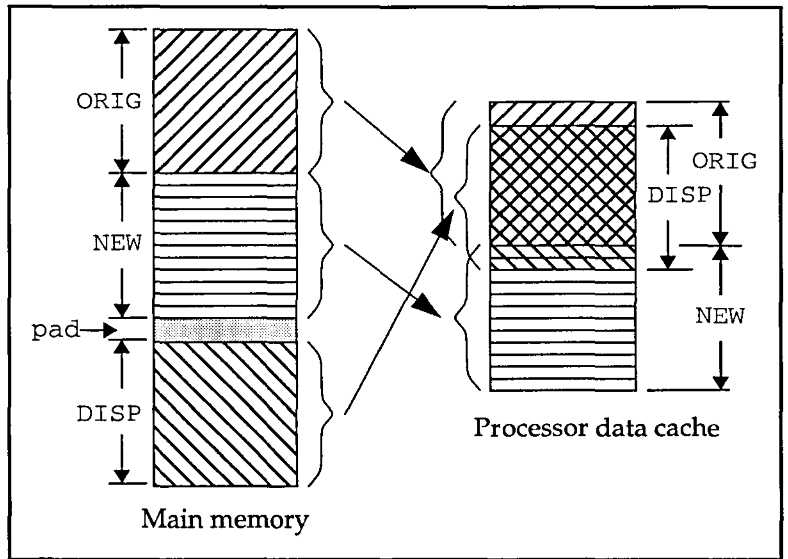


Figure 5 Array layouts—non-thrashing

It is important to note that this is a highly simplified, worst-case example. On SPP1000 Series machines, thrashing can happen any time two data items that are referenced in the same loop are an integral multiple of 1 Mbyte apart in virtual memory. This can happen with data that is not stored in COMMON, in which case it is much more difficult to see, as such data can be stored noncontiguously and may be intermixed with completely unrelated data items. The loop blocking optimization (described in Chapter 3, "Compiler optimizations") will eliminate thrashing from certain nested loops, but not from all loops. Declaring arrays with dimensions that are not powers of two can help, but will not necessarily eliminate the problem completely. Using COMMON blocks in Fortran can also help because it allows you to accurately measure distances between data items, making thrashing problems easier to spot before they happen.

Cache thrashing: SPP1200 Series/SPP1600 Series

The thrashing problem is lessened by the SPP1200 Series/SPP1600 Series' assist cache. Still, there are two sets of conditions that may cause cache thrashing on SPP1200 Series/SPP1600 Series machines. In the first, two thrashing data items, *x* and *y*, must be accessed in the following way:

- *x* and *y* are encached into the assist cache consecutively; *y* therefore immediately follows *x* into the cache.
- *x* and *y*, having entered the cache consecutively, move through the queue consecutively, so that when *x* is displaced from the main cache, *y* is the next cache entry to be displaced.
- *y* displaces *x* from the main cache. To do this, the addresses of *x* and *y* must be identical in their lowest-order 18 bits (20 bits on SPP1600 Series). When *y* is displaced from the assist cache to the main cache immediately after *x* is displaced, *y* displaces *x* from the main cache. If this condition is not met, both items will always be available in the main cache after being displaced from the assist cache.
- In order to adversely affect performance, no reuse of *x* or *y*'s cache line can occur while it resides in either cache, and the program must reference some element of *x*'s cache line after *y* displaces it from the main cache, causing the cycle to repeat.

Architecture

The other conditions that may cause cache thrashing on SPP1200 Series/SPP1600 Series machines require *x* and *y* to be accessed in the following way:

- They cyclically displace each other from the assist cache, with no reuse of either item's cache line between displacements. To do this, the loop must be written so that *y* is referenced at the exact instant when *x* is the next item to be displaced from the assist cache. This condition is difficult to envision because the algorithm that determines when an assist-cache entry must be displaced depends on many variable runtime conditions.
- They displace each other from the main cache. To do this, the addresses of *x* and *y* must be identical in their lowest-order 18 bits (20 bits on SPP1600). If this condition is not met, both items will always be available in the main cache after being displaced from the assist cache.
- In order to adversely affect performance, no reuse on *x*'s cache line can be exploited before *x* is displaced by *y* in either cache. This means that during all the references that must occur between the reference of *x* and the reference of *y* in order for *y* to overwrite *x* in the assist cache, no reuse on *x*'s cache line can occur. Similarly, if *x* resides in the main cache, no reuse on its cache line can occur before it is overwritten when *y* is displaced from the assist cache by some other data item.

If you experience performance problems in a loop and find that it is accessing data as discussed in the preceding conditions for thrashing, try inserting a 64-byte pad between the arrays you suspect are thrashing using the method described in the "Cache thrashing: SPP1000 Series" section. This should prevent main cache thrashing and alleviate the problem.

Interleaving

Physical pages are interleaved across the 8 banks of a hypernode by CTIcache lines. Contiguous CTIcache lines are assigned in round-robin fashion, first to the even banks, then to the odd, as shown in Figure 6.

Each memory block shown in Figure 6 is located on a separate functional block within the hypernode.

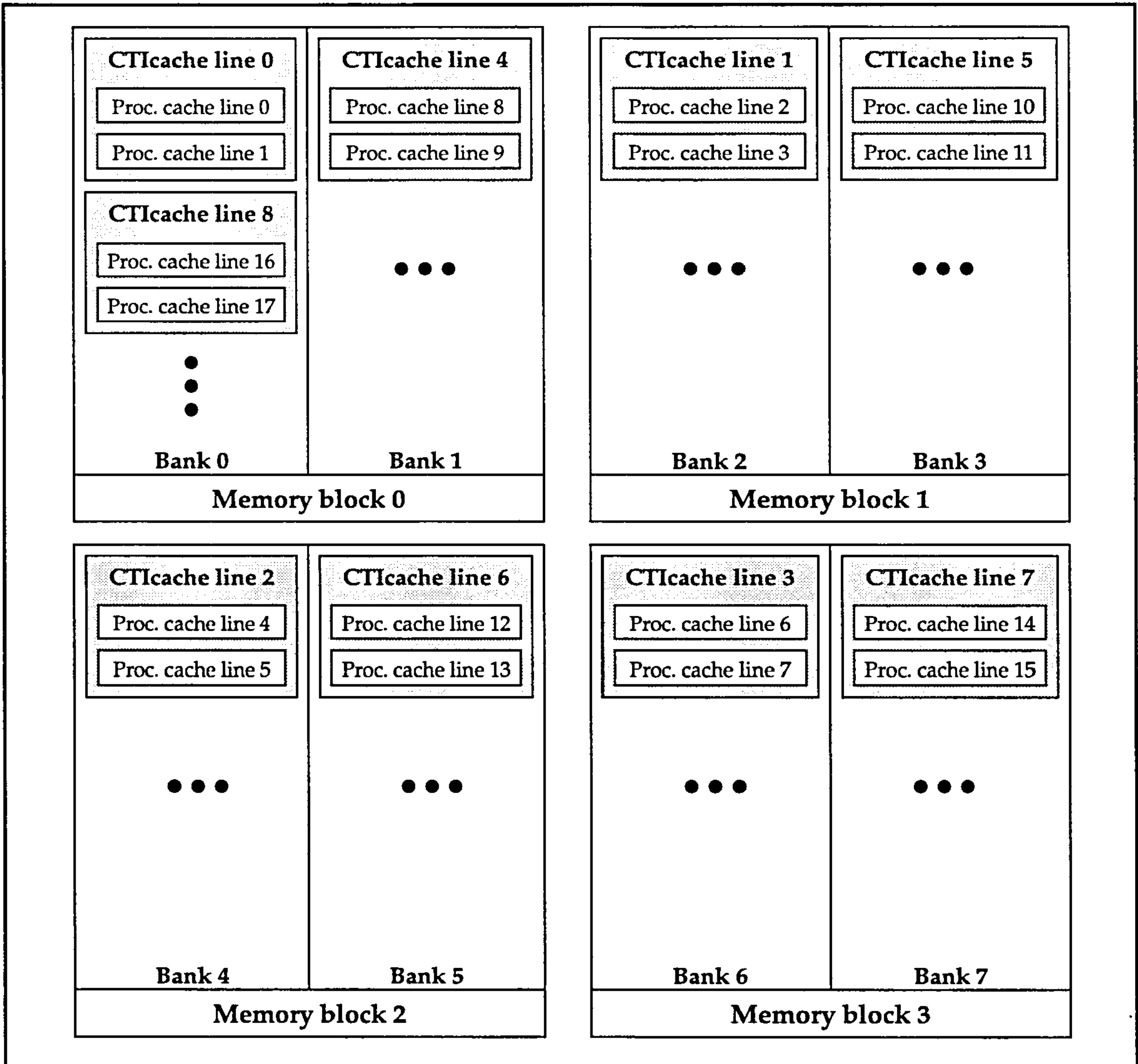


Figure 6 SPP Series memory interleaving

Interleaving speeds memory accesses by allowing several processors to access contiguous data simultaneously. This is extremely beneficial when a loop that manipulates arrays is split among many processors; in the best case, threads will access data in patterns with no bank contention. Even in the worst case, where each thread initially needs the same data from the same bank, after the initial contention delay the accesses will be spread out among the banks.

Interleaving example

The following example illustrates a nested loop that accesses memory with very little contention. This example is greatly simplified for illustrative purposes, but the concepts apply to arrays of any size.

```
REAL*8 A(12,12), B(12,12)
...
DO J = 1, N
  DO I = 1, N
    A(I,J) = B(I,J)
  ENDDO
ENDDO
```

Assume that arrays `A` and `B` are stored contiguously in main memory, with `A` starting in bank 0, CTIcache line 0, processor cache line 0, as shown in Figure 7.

Assume the SPP Series Fortran compiler parallelizes the `J` loop to run on as many processors as are available in the subcomplex (up to `N`). Assuming `N=12` and there are four processors available when the program is run, the `J` loop could be divided into four new loops, each with 3 iterations. Each new loop would run to completion on a separate processor. We'll refer to these four processors as CPU0 through CPU3.

Note

This example is designed to simplify illustration. In reality, the dynamic selection optimization (discussed in Chapter 3, "Compiler optimizations") would, given the iteration count and available number of processors described, cause this loop to run serially. The overhead of going parallel would outweigh the benefits.

In order to execute the body of the `I` loop, `A` and `B` must be fetched from memory and encached. Each of the four processors running the `J` loop will attempt to fetch its portion of the arrays, most likely simultaneously.

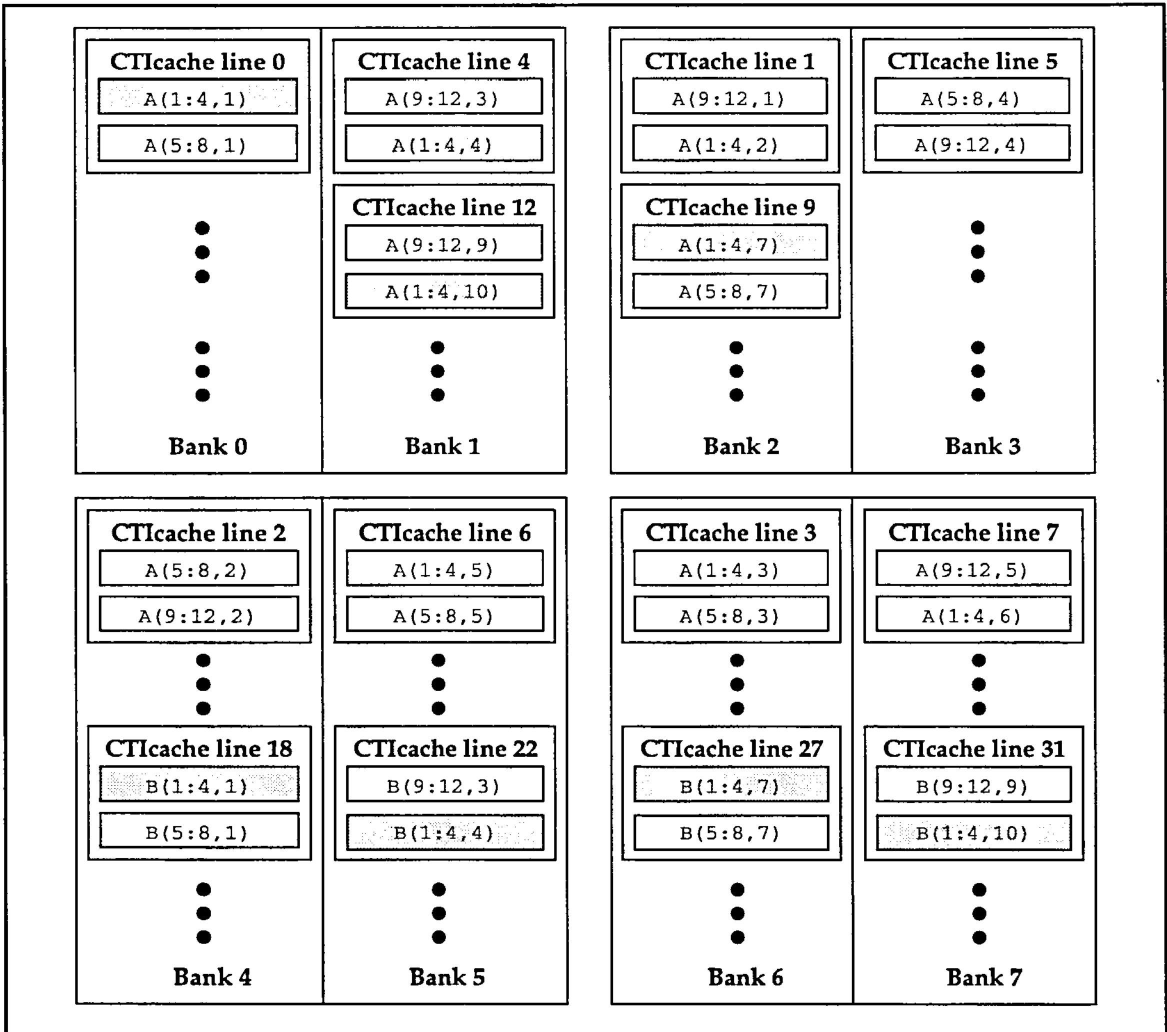


Figure 7 Interleaving of arrays `A`, `B`, and `C`

CPU0 needs `A(1:12,1:3)` and `B(1:12,1:3)`

CPU1 needs `A(1:12,4:6)` and `B(1:12,4:6)`

CPU2 needs `A(1:12,7:9)` and `B(1:12,7:9)`

CPU3 needs `A(1:12,10:12)` and `B(1:12,10:12)`

This means CPU0 will attempt to read arrays `A` and `B` starting at elements `(1,1)`, CPU1 will attempt to start at elements `(1,4)` and so on. For clarity, Figure 7 shows the first 8 CTIcache lines consecutively; after these, only the initial cache lines for each processor are shown. Each processor's initial cache line is shaded.

Without interleaving, all four CPUs would request their arrays simultaneously from the same bank. Arbitration logic would order the CPUs; whichever ended up going first would get its requested cache line while the second experienced a ~255 ns delay (about 26 processor clocks on an SPP1000 system, and 31 processor clocks on an SPP1200 system or an SPP1600 system), the third a ~510 ns delay, and the fourth a ~765 ns delay. Before the second CPU had even received its cache line, the first would likely be back in the queue waiting for its next cache line.

Interleaving helps to eliminate such contention. `A(1,1)`, which is the first element of the chunk needed by CPU0, is on cache line 0 in bank 0; `A(1,4)`, the first element needed by CPU1, is on cache line 4 in bank 1; `A(1,7)`, the first element needed by CPU2, is on cache line 9 in bank 2, and `A(1,10)`, the first element needed by CPU3, is on cache line 12 in bank 1. Contention exists only between CPU1 and CPU3, and after one of these CPUs gets its cache line and moves on, it will proceed at full speed with the other CPU 28 clocks behind it. In other words, after the initial access, the contention will cease. Contention may resurface occasionally as the processors make their way through the data, but the resulting delays are minimal compared to what could be expected without interleaving.

- The initial chunks of `B` are even more favorably distributed, in banks 4, 5, 6, and 7, respectively. No initial contention exists for `B`.

Subcomplexes

On Exemplar systems, processes run on subcomplexes, which are arbitrary collections of processors and global memory. Subcomplexes are highly configurable, and configuration is done using the Subcomplex Manager. For more information on the Subcomplex Manager, refer to the *SPP-UX System Administrator's Guide.*

Subcomplexes allow the system administrator to tailor processors and memory to your specific application needs, making the most efficient use of your Exemplar system resources. For example, a nonparallel or non-time-critical application can be allocated a single processor; an application containing a lot of fine-grain parallelism can be allocated many processors, and an application requiring large amounts of memory can be allocated processors on several hypernodes.

Physical configuration

Subcomplexes can consist of from one processor to the total number installed on the machine. Hypernodes can be split among as many subcomplexes as there are processors in the hypernode, and subcomplexes can be subsets or supersets of hypernodes. Processors can only belong to one subcomplex at a time, and every processor must belong to a subcomplex.

Figure 8 shows an 8-hypernode, 64-processor system split into four subcomplexes.

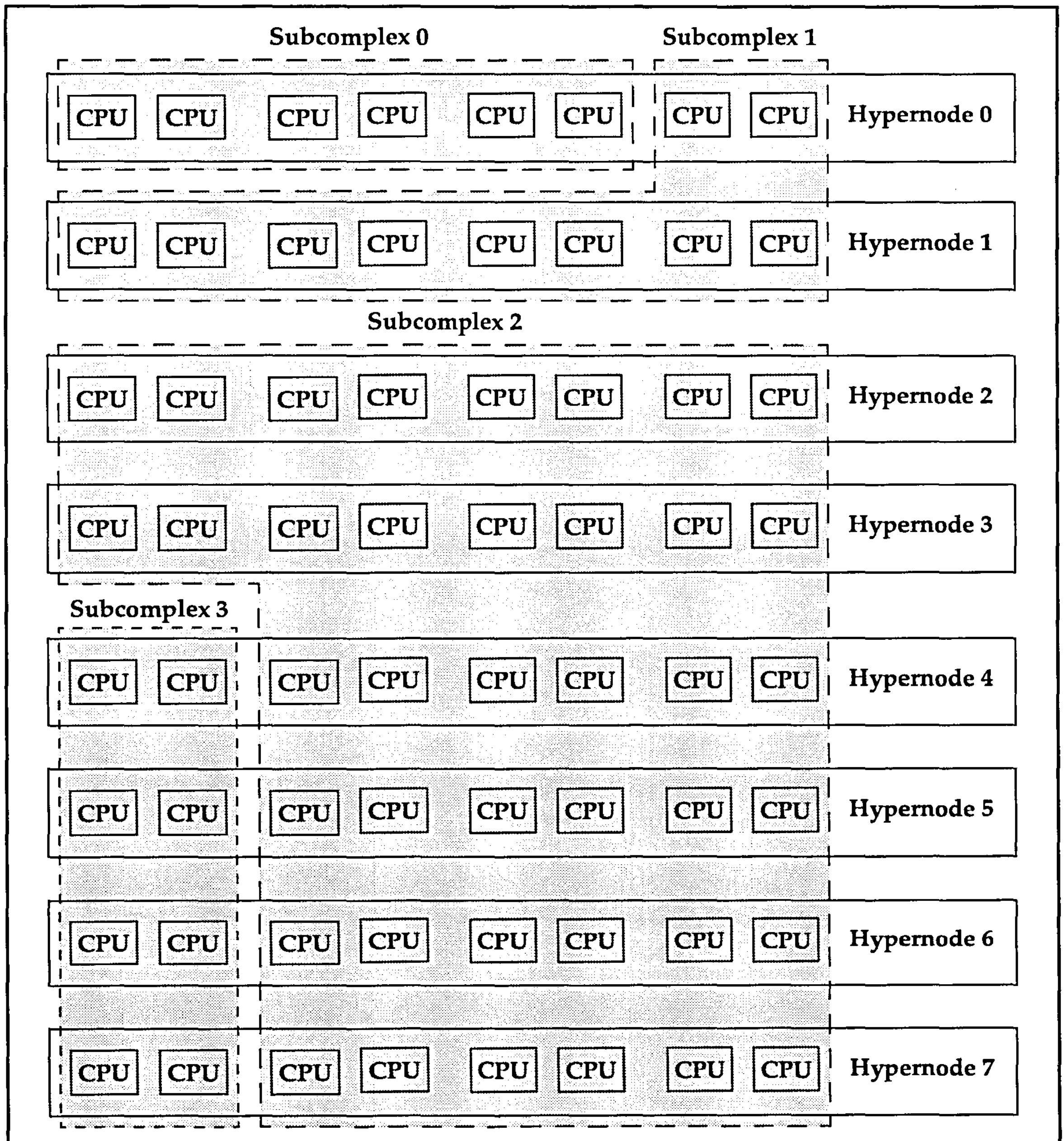


Figure 8 Hypothetical subcomplex configurations

Processors can be added to or removed from a subcomplex at any time; depending on the global memory requirements of the hypernodes making up the subcomplex, the subcomplex may or may not need to go idle. Idling a subcomplex on which processes are running will cause the processes to go idle; activating the subcomplex will automatically activate the idle threads.

Subcomplex memory can also be configured using the Subcomplex Manager. Global memory can only be reliably allocated once, shortly after booting. Global memory is allocated, up to the total amount available, in 8 Mbyte increments for machines that have two memory controllers and two CTI rings and in 16 Mbyte increments for machines having four memory controllers and four CTI rings. The CTIcache memory must be configured at system startup, and the CTIcache size must be uniform across all hypernodes. Valid CTIcache sizes are 64 Mbytes, 128 Mbytes, 256 Mbytes, and so forth.

Also configurable are subcomplex permissions, which are similar to file permissions and can be set at the user, group, and world level.

- Read permissions on a subcomplex allow you to read the subcomplex configuration
- Write permissions allow you to change the subcomplex's job scheduling policies, such as process execution priority
- Execute permissions allow you to run a process on the subcomplex

Only the system administrator can configure or reconfigure a subcomplex.

Subcomplex memory

GSM is global to a subcomplex, and processes run entirely within their assigned subcomplex. For a program to run on more than one subcomplex, message passing via sockets (not the CTI rings) must be used to communicate between the subcomplexes; this is much slower than interhypernode communication within a subcomplex. Message passing between subcomplexes is further discussed in the *Convex MPICH User's Guide for Exemplar Systems* and the *PVM/GSM User's Guide for Exemplar Systems*.

The virtual address spaces accessible from within a subcomplex are unique; this prevents processes running on one subcomplex from accessing memory on another subcomplex. Furthermore, the virtual address spaces accessible to each process running on a subcomplex are unique; each process is assigned a 4-Gbyte virtual space, and processes cannot access each other's memory.

When two or more subcomplexes share a single hypernode, the physical memory pages assigned as global memory for each subcomplex are unique, preventing contention over global memory. However, the physical pages associated with hypernode-local and CTIcache memory are shared among multiple subcomplexes running on one hypernode; this can result in contention over these resources.

Compiler optimizations

3

This chapter discusses the various optimization options available for use with the SPP Series Fortran and C compilers and provides detailed explanations of the optimizations performed under each option.

Optimization options

Five optimization options are available for use with the SPP Series C and Fortran compilers. These options have identical names and perform identical optimizations regardless of which compiler you are using. They are specified on the compiler command line along with any other options you wish to use. SPP Series compiler optimization levels are summarized in Table 2.

Table 2 SPP Series C and Fortran optimization options

Option	Description
`-no`	Machine instruction level scalar optimizations
`-O0`	Basic block level scalar optimizations
`-O1`	Program unit level scalar optimizations and global register allocation
`-O2`	Global instruction scheduling and data localization optimizations; this option is the default
`-O3`	Parallel optimizations

These options are cumulative; each option retains the optimizations of the previous option. For example, entering the following command line compiles the Fortran program `foo.f` with all `-O2`, `-O1`, `-O0`, and `-no` level optimizations shown in Table 2.

```
% fc -O2 foo.f
```

Optimizations

In addition to these options, the `-noautopar` option is available for use with the `-O3` option. `-noautopar` causes the compiler to parallelize only those loops that are immediately preceded by `loop_parallel` or `prefer_parallel` directives or pragmas; for more information, refer to Chapter 4, "Basic shared-memory programming." The `-nonodepar` option, also available for use with the `-O3` option, prevents the compiler from implementing node-parallelism; thread-parallelism—both automatic and directive-specified—is still implemented.

`-no` level optimizations

At optimization level `-no`, the compiler performs scalar optimizations that span no more than a single source statement. These optimizations create object code that fully uses the scalar features of the PA-RISC architecture.

Instruction scheduling

Instruction scheduling rearranges machine instructions to use the computer's functional units most effectively. Each CPU on an SPP Series system has multiple functional units on which operations execute simultaneously.

At optimization level `-no`, the compiler rearranges the machine instructions generated by no more than a single source statement to maximize use of the functional units.

Concurrent execution of machine instructions on multiple functional units, within a single processor, is distinct from parallel processing, which occurs on multiple processors.

Span-dependent instructions

There are several ways to specify a branch in the PA-RISC machine language; short branches can be more efficiently executed than long branches. The SPP Series compilers automatically generate branches using the most efficient and appropriate branching techniques.

For more information on branch instructions, see the Hewlett-Packard *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.*

Register allocation

The SPP Series compilers use a technique for allocating registers that fully exploits the PA-RISC register set. This allows grouping of register loads and concurrent execution of instructions (*pipelining*), and reduces register conflicts.

Tree-height reduction

The compiler represents expressions internally as trees. When they involve floating point numbers, these trees are optimized by *tree-height reduction* or *balancing*. For example, consider this Fortran expression involving `REAL` variables:

```
A + B + C + D + E + F + G + H
```

The expression can be evaluated as follows:

```
(((((((A + B) + C) + D) + E) + F) + G) + H)
```

`(A+B)` is evaluated first, the result is added to `C`, and so on until the expression is fully evaluated. Figure 9 shows how the compiler represents this order internally.

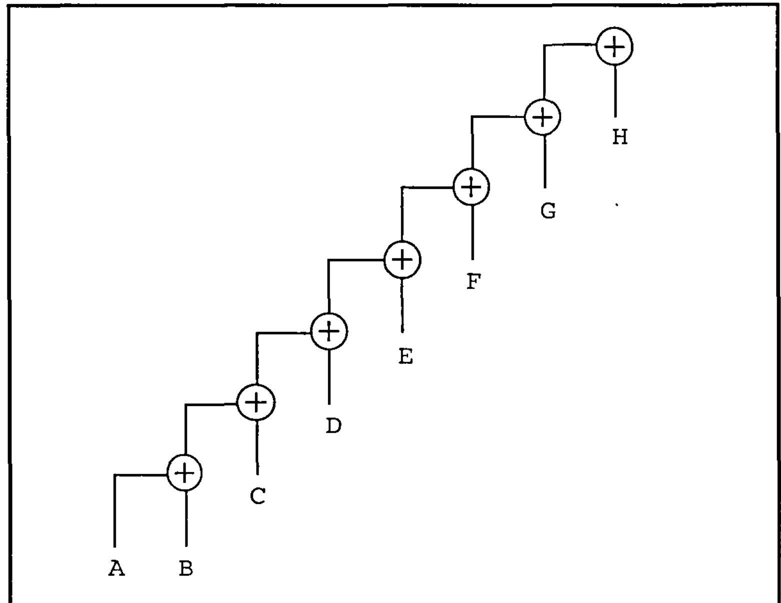


Figure 9 Unbalanced tree representation

The PA-RISC processor's add functional unit takes two clocks to complete an add, but pipelining allows it to issue one add result per clock if the pipeline is kept full. The unit does not chain, so the result of an add in progress cannot be used in the next add.

Because each addition in this example depends on the result of the addition to the left, pipelining in the add functional unit cannot be exploited; it can then only issue one result every two clocks.

Another way to evaluate the expression is

```
(((A + B) + (C + D)) + ((E + F) + (G + H)))
```

This is represented internally as shown in Figure 10.

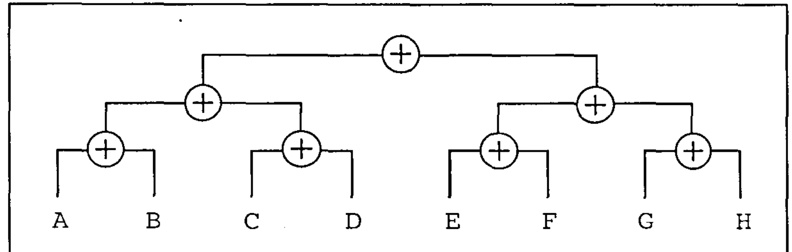


Figure 10 Balanced tree representation

In Figure 9, the depth of the tree is seven; in Figure 10, the depth of the tree is three. The code generated for the tree in Figure 9 executes more slowly than that generated for the tree in Figure 10.

This is because none of the four additions in the innermost parentheses (represented as leaf nodes on the tree in Figure 10) requires the result of another addition. The additions can therefore be fed to the add functional unit so that they fill its pipeline, allowing it to issue one result per clock after the first add and up until the last. `(A+B)` is evaluated on the first clock, followed by `(C+D)` on the second; on the third, (`A+B`) is done and the next add is starting. This pattern continues until the expression is fully evaluated.

The deeper the tree representing the expression, the more time is required to evaluate the expression. If a particular evaluation order is not specified with parentheses, the compilers choose one that minimizes the depth of the expression and maximizes instruction pipelining, while maintaining the arithmetically correct evaluation order. Because the compilers choose evaluation order to ensure the most efficient execution, you can write expressions in any order; if your expression requires a certain order, be sure to indicate it with parentheses.

Short-circuit evaluation of conditionals in Fortran

Short-circuiting the evaluation of conditionals is ANSI standard behavior in C, but Convex Fortran also performs this optimization.

Short circuiting increases the efficiency of IF statements by skipping irrelevant tests when logical operators are involved in the conditional. Convex Fortran short-circuits evaluation of IF statements that contain .AND. and .OR. operators that have logical operands and are used in a logical context. Take, for example, the following Fortran IF statement:

```
IF ((A .EQ. B) .OR. F(G)) THEN
```

If (A .EQ. B) evaluates to true, the evaluation of F(G) is skipped, and the THEN portion of the statement is evaluated.

Similarly, given the Fortran code

```
IF ((A .EQ. B) .AND. F(G)) THEN
```

if (A .EQ. B) evaluates to false, the evaluation of F(G) and the THEN portion of the statement is skipped.

Short-circuit evaluation works with all types of IF statements (arithmetic, logical, and block). Performing arithmetic (+, -, *, /) on a logical expression disables short circuiting within that expression. Logical-valued expressions used as arguments to function calls within an IF statement's conditional expression may not be short circuited. Note that the binary operators .EQ., .NE., .LT., .LE., .GT., and .GE. always produce a logical result.

The compiler short-circuits the evaluation of conditionals by default. You can disable short-circuiting in Fortran by specifying the -nosc flag on the compiler command line.

Data alignment on natural boundaries

To provide more efficient access to data, the compiler automatically aligns data objects to their natural boundaries in memory. This means that a data object's address is integrally divisible by the length of its data type; e.g. REAL*8 objects have addresses integrally divisible by 8 bytes.

Note

Aliases can inhibit data alignment. Be especially careful when equivalencing arrays in Fortran.

To ensure the efficient layout of such aligned data in memory, you should declare scalar variables in order from longest to shortest data length. This minimizes the amount of padding the compiler has to do to get the data onto its natural boundary. Consider the following Fortran example:

```
C      CAUTION: POORLY ORDERED DATA FOLLOWS:
       LOGICAL*2 BOOL
       INTEGER*8 A, B
       REAL*4 C
       REAL*8 D
```

Here, the compiler must insert 6 blank bytes after BOOL in order to correctly align A, and it must insert 4 blank bytes after C to correctly align D.

The same data is more efficiently ordered as shown in the following example:

```
C      PROPERLY ORDERED DATA FOLLOWS:
       INTEGER*8 A, B
       REAL*8 D
       REAL*4 C
       LOGICAL*2 BOOL
```

Natural boundary alignment is the default on SPP Series Convex compilers. Traditional C Series-like tight packing of variables can be specified using the `-align cseries` option as described in the section "Data alignment options" on page 337.

Natural boundary alignment is performed on all data. It should not be confused with CTIcache line boundary alignment, which is performed as described in the section "Data alignment" on page 21. Also discussed in Chapter 2 are the `align_cti` directive and pragma and the `-align cti` compiler option, all of which facilitate CTIcache line boundary alignment.

-O0 Level optimizations

At optimization level -O0, the compiler performs scalar optimizations within a basic block; a basic block is a sequence of statements with only one entry point and one exit. Basic blocks may end with a branch, but they cannot contain branches. The compiler also continues to perform the optimizations performed at -no.

Instruction scheduling

At optimization level -O0 and above, instructions from *multiple* statements, as well as those from single statements, are scheduled as a group.

Redundant-assignment elimination

Redundant-assignment elimination removes unnecessary assignments to a variable. When a variable is not used between two assignments, the first assignment is eliminated. The code in the following Fortran example contains a redundant assignment, X=Y+C, which the compiler removes:

Original code	Optimized code
`X = Y + C`	`!(statement eliminated)`
`!(X not used)`	.
.	.
.	.
`X = 3.1416`	`X = 3.1416`
.	.
.	.
.	.
`Y = (X + 7) * 2.15`	`Y = (X + 7) * 2.15`

Optimizations

Assignment substitution

Assignment substitution eliminates redundant loads. The compiler stores the value assigned to a variable in a register and references the register in subsequent references to the variable. A Fortran example appears below.

Original code	Optimized code
`X = Y + C`	`REG = Y + C`
`X = X * 4.4`	`REG = REG * 4.4`
`T = X * B + 12.4`	`T   = REG * B + 12.4`
`X = 4.179`	`X   = 4.179`

After the machine instructions for the first statement execute, the value of `Y+C` remains in a register. The compiler replaces subsequent references to `X` with references to this register until the value of `X` changes or until the end of the basic block is reached. This optimization eliminates repeated loading and storing of `X` into a register, thereby increasing performance and providing opportunities for further optimization. In this example, assignment substitution makes the first assignment to `X` redundant, so the compiler eliminates the assignment.

Because the compiler substitutes assignments, you rarely need to optimize a program by replacing a variable reference with a constant in the source code.

Common subexpression elimination

The compiler recognizes subexpressions that repeat within a basic block. The compiler retains the value of the subexpression in a register, which eliminates redundant computations and register loads. For example, the compiler recognizes `B+C` as a common subexpression of `A+B+C+D` and `B+E+C` and calculates the subexpression only once.

The compiler also eliminates redundant array address calculations. As with assignment substitution, you do not need to manually create a temporary variable in which to store the value of a common subexpression. The compiler performs that function automatically.

Redundant-use elimination

This optimization is a special case of common subexpression elimination where the subexpression is a variable. The compiler detects multiple references to a variable between assignments and retains the value of the variable in a register. This action helps eliminate redundant register loads.

Constant propagation and folding

After assigning a constant to a variable, the compiler replaces subsequent references to the variable with the constant. For example, if you write X=5, the compiler replaces X with 5 within that basic block or until a new value is assigned to the variable. This is known as *constant propagation*, which is a form of assignment substitution.

A Fortran example of constant propagation and folding follows:

Original code	**Optimized code**
`I = 5`	`I = 5`
`J = 0`	`!(assignment eliminated)`
.	.
.	.
.	.
`J = J + 2`	`J = 2`
.	.
.	.
.	.
`K = K + I * J`	`K = K + 10`

The compiler also replaces operations on constants with the result of the operation. This is known as *constant folding*. For example, it replaces Y=5+7 with Y=12. It then propagates the constant value to replace future references to Y within the basic block. The Fortran compiler also propagates and folds values assigned to names in PARAMETER statements.

The compiler folds the most frequently used Fortran intrinsics and C library routines when they are applied to constant arguments. For example, SIN(0.0) becomes 0.0. The compiler

Optimizations

also folds exponentiation involving constants. For example, 3**3 becomes 27.

The compiler type-converts constants, if necessary, before propagating and folding them. If a Fortran program contains the expression X=1, where X is REAL, the compiler converts 1 to 1.0 before propagating it.

If an integer overflow occurs as a result of constant folding in Fortran, the compiler reports "Integer constant truncation." To see such a message in C, you must supply the -d integer_overflow=e option on the cc command line. If a floating-point overflow occurs, both compilers report "Real constant either too large or too small." Floating-point underflow always results in zero. If any of these messages or conditions occur, eliminate the offending operation or bring the value of the constant within acceptable bounds.

Algebraic and trigonometric simplification

The compiler simplifies algebraic and trigonometric expressions, as shown in the following Fortran examples:

Original expression	Optimized expression
`X + 0`	`X`
`X * 1`	`X`
`X * 0`	`0`
`K .AND. -1`	`K`
`K .AND. 0`	`0`
`K .OR. -1`	`-1`
`K .OR. 0`	`K`
`-1 * X`	`-X`
`X - X`	`0`
`X / -1`	`-X`
`(-1) ** K`	`1-((K .AND. 1) * 2)`
`X ** 0.5`	`SQRT(X)`
`X ** 0`	`1`
`1 ** X`	`1`
`X / X`	`1`
`0 - X`	`-X`
`0 / X`	`0`
`SIN(X) * COS(X)`	`0.5 * SIN(2X)`
`SIN(X) / COS(X)`	`TAN(X)`

The compiler performs obvious variations of these operations for the commutative operators. For example, in Fortran the compiler converts `X+(0+Y)` to `X+Y`.

Optimizations

-O1 Level optimizations

At optimization level -O1, global optimization is done across a group of basic blocks but within a single procedure. The -O1 option performs global, basic-block, and machine-instruction level optimizations.

Constant propagation and folding

Propagating and folding constants at the procedure level is analogous to performing the same operations at the basic-block level. The scope of the optimization is now an entire procedure.

A Fortran example of constant propagation and folding follows:

Original code

```
      INTEGER A,B,C
      A = 5
      B = 15
      READ *, I
      IF (I) 10,10,15
10    A = 6
      C = A
      GOTO 20
15    C = A + B
      GOTO 25
20    B = A + C
      GOTO 30
25    B = A + B + C

30    PRINT *,A,B,C
      END
```

Optimized code

```
      INTEGER A,B,C
      A = 5
      B = 15
      READ *, I
      IF (I) 10,10,15
10    A = 6
      C = 6    !A=6
      GOTO 20
15    C = 20 !A=5,B=15
      GOTO 25
20    B = 12 !A=6,C=6
      GOTO 30
25    B = 40
     !A=5,B=15,C=20
30    PRINT *,A,B,C
      END
```

The compiler propagates and folds constants globally at optimization level -O1 and higher, which eliminates the need to propagate constants by hand in programs compiled at these levels.

Redundant-assignment elimination

At optimization level -O1, the compiler eliminates assignments to variables that do not have subsequent references within the program unit. The following Fortran example shows how the compiler eliminates redundant assignments to the variable A:

Original code

```
    SUBROUTINE FOO
    INTEGER A
    .
    .
    .
    X = Y * Z
    A = Y**3
    ASSIGN 10 TO B
    .
    .
    .
    IF (A .GT. 0) THEN
      .
      .
      .
      A = X * Y + 3.1416
      GOTO B
    ELSE
      .
      .
      .
      X = (X + 7) * Z + 3.1416 ! X USED LATER
    ENDIF
    .
    .
    .
C   A IS NOT USED LATER IN THIS ROUTINE
    END
```

Optimizations

As shown in the optimized code that follows, the Fortran compiler does not eliminate ASSIGN statements and assignments to dummy arguments, function names, and common variables.

Optimized code

```
SUBROUTINE FOO
INTEGER A
.
.
.
X = Y * Z
A = Y**3
ASSIGN 10 TO B
.
.
.
IF (A .GT. 0)THEN
   .
   .  ! ASSIGNMENT TO A REMOVED
   .
  GOTO B
ELSE
   .
   .
   .
  X = (X + 7) * Z + 3.1416 ! X USED LATER
ENDIF
.
.
.
END
```

If the right side of a redundant assignment statement contains a procedure call, the compiler eliminates the assignment and retains the call, as in the following Fortran example:

Original code	**Optimized code**
`SUBROUTINE FOO`	`SUBROUTINE FOO`
.	.
.	.
.	.
`I = INTFUN(X)`	`<NULL> = INTFUN(X)`
.	.
.	.
.	.

```
Comment: I not used
```

If a C or Fortran function appearing in an assignment has no side effects, the compiler can eliminate the function call, as well as the assignment, improving efficiency. A procedure is free of side effects if it:

- Does not modify the value of an argument
- Does not modify the value of a global variable or a variable that appears in a Fortran COMMON block
- Does not modify the value of local static variables used on subsequent calls
- Does not perform input or output
- Does not call another procedure that has side effects

Existing procedure compilers cannot automatically determine whether a side effect exists. The Convex Fortran and C compilers eliminate procedure calls only if you explicitly request this behavior with the NO_SIDE_EFFECTS directive or pragma.

The form of this Fortran directive is

C$DIR NO_SIDE_EFFECTS (*func_list*)

The form of the C pragma is

```
#pragma _CNX no_side_effects(func_list)
```

where *func_list* is a list of procedure names separated by commas. The directive must precede the procedure call that does not contain side effects.

Caution

Do not use the NO_SIDE_EFFECTS directive or pragma on a call to a procedure that

- **Changes the value of an argument**
- **Changes the value of a COMMON or global variable**
- **Performs input or output**
- **Changes the value of a static local variable in C**
- **Changes the value of a SAVEd local variable in Fortran**
- **Calls another procedure that performs one of these operations**

For more information about the NO_SIDE_EFFECTS directive, see Appendix A, "Compiler directives."

Dead-code elimination

If, as a result of constant propagation and folding, the compiler can fold an arithmetic or logical expression in a Fortran IF statement to .TRUE. or .FALSE., or if the expression can be folded to a constant in C, the compiler eliminates any unreachable code that results.

Copy propagation

The compiler can replace a variable with another variable to which it has been equated. This is called *copy propagation*. For example, after evaluating the statement X=Y, the compiler replaces later occurrences of X with Y.

In the following Fortran example, if the compiler determines that X and Y are unchanged between the assignment and the reference, it replaces X with Y:

```
X = Y
.
.
.
W = Z - X
```

becomes

```
.
.
.
W = Z - Y
```

Common subexpression elimination

The compiler eliminates common subexpressions at the global level. The compiler retains the value of the common subexpression in a register if one is available; otherwise, the compiler assigns the value to a temporary variable. The compiler then replaces subsequent occurrences of the common subexpression with references to the register or temporary variable.

In the following Fortran example, the compiler determines that the subexpression must be calculated whether the condition associated with the IF statement evaluates to .TRUE. or .FALSE.:

```
SUBROUTINE GCSE2
.
.
.
IF (K .LT. L) THEN
  A = (C * 4) / -(J * B + SQRT(C))
ELSE
  E = (E * 4) / -(J * B + SQRT(C))
ENDIF
F = (B * 4) / -(J * B + SQRT(C))
.
.
.
END
```

The compiler saves the value of the common subexpression in the temporary variable T1 and uses the variable to compute the value for assignment to A, E, and F, as follows:

```
SUBROUTINE GCSE2
.
.
.
T1 = -(J * B + SQRT(C))
IF (K .LT. L) THEN
  A = (C * 4) / T1
ELSE
  E = (E * 4) / T1
ENDIF
F = (B * 4) / T1
.
.
.
END
```

The analogous C code follows:

```
void gcse2()
{
   .
   .
   .
   if (k < l)
     a = (c * 4) / -(j * b + sqrt(c));
   else
     e = (e * 4) / -(j * b + sqrt(c));
   f = (b * 4) / -(j * b + sqrt(c));
   .
   .
   .
}
```

As with the Fortran example, the compiler recognizes that the common subexpression is used before and after the `if` statement. It saves the value of the subexpression in the temporary variable `t1` before the `if` statement and uses this variable to compute the values of `a`, `e`, and `f`, as shown in the following example:

```
void gcse2()
{
   ...
   t1 = -(j * b + sqrt(c));
   if (k < 1)
     a = (c * 4) / t1;
   else
     e = (e * 4) / t1;
   f = (b * 4) / t1;
   ...
}
```

Code motion

Code motion is the movement of invariant expressions out of loops. An invariant expression yields the same result on every iteration of a loop.

In the following Fortran example, all variables used in the assignment to `A` remain invariant within the loop:

```
SUBROUTINE GCM
REAL AR(10)
.
.
.
DO I = 1, 10
  A = C / (-(E * B) + SQRT(C))
  AR(I) = A + B * C
ENDDO
.
.
.
END
```

The compiler recognizes this and moves the calculations and assignments out of the loop, performing these costly calculations only once.

Optimizations

The optimized code follows:

```
SUBROUTINE GCM
REAL AR(10)
.
.
.
A = C / (-(E * B) + SQRT(C))
DO I = 1, 10
  AR(I) = A + B * C
ENDDO
.
.
.
END
```

In C:

```
void gcm() {
  float ar[10]
  .
  .
  .
  for(i=0;i<10;i++) {
    a = c/(-(e*b) + sqrt(c));
    ar[i] = a+b+c;
  }
  .
  .
  .
}
```

After optimization:

```
void gcm() {
  float ar[10]
  .
  .
  .
  a = c/(-(e*b) + sqrt(c));
  for(i=0;i<10;i++)
    ar[i] = a+b+c;
  .
  .
  .
}
```

If an invariant expression does not lie on a path to all loop exits, the compiler does not move the invariant expression unless you use the -uo (potentially unsafe optimizations) compiler option.

For more information about using the -uo option, refer to Chapter 8, "Potentially unsafe optimizations."

Strength reduction

In some cases, the compiler can replace an arithmetic operation with an equivalent operation (possibly nonarithmetic) that executes more quickly. Such replacements are called strength reductions.

Explicit arithmetic reductions

The compiler can reduce the strength of various arithmetic operations. For example, the compiler transforms integer multiplication on positive numbers by 2, 4, 8, and 16 into integer shifts, as shown in the following Fortran code:

`J * 2` becomes `IISHFT(J, 1)`

`J * 4` becomes `IISHFT(J, 2)`

Multiplication involving integer constants is reduced to addition:

`X * 2` becomes `X + X`

When the -uo (potentially unsafe optimizations) command-line option is specified, division by a constant is reduced to multiplication, which is substantially faster:

`X/C` becomes `D*X` where `D=1/C`

Because C is a constant, D also is a constant, which can be computed at compile time.

Induction variables and constants

The compiler can reduce the strength of operations to optimize loop induction variables and loop constants. Multiplications within a loop that calculate the address of a subscripted variable are often candidates for strength reduction.

The compiler does not reduce operations that only involve REAL, float or double variables. Because floating-point arithmetic is imprecise, reduced operations do not always yield equivalent results. If an expression does not lie on a path to all loop exits, the compiler does not reduce the expression unless you use the -uo option.

Optimizations

In the following Fortran example, the compiler recognizes that `I` is incremented by 2 on each iteration and that `X` is incremented by `2*C`, a loop constant:

```
      SUBROUTINE GSR
      I = 1       !induction var
10    X = I * C   !loop induction value
      .
      .
      .
      I = I + 2
      IF(I .LE. 100) GOTO 10
      .
      .
      .
      END
```

The analogous C code follows:

```
void gsr()
{
   int i = 1;      /* induction var */
   ...
   do {
      x = i * c;   /* loop induction value */
      ...
      i += 2;
   } while(i<=100);
}
```

As shown in the following optimized version, the compiler produces code that calculates `2*C` only once and increments `X` by the value saved in `T2` instead of calculating `I*C` on every iteration:

```
      SUBROUTINE GSR
      I = 1
      T1 = C
      T2 = 2 * C
10    X = T1
      .
      .
      .
      T1 = T1 + T2
      I = I + 2
      IF(I .LE. 100) GOTO 10
      .
      .
      .
      END
```

In C:

```
void gsr()
{
   int i = 1;
   ...
   t1 = c;
   t2 = c + c;
   do {
      x = t1;
      ...
      t1 += t2;
      i += 2;
   } while(i <= 100);
}
```

Global register allocation

Scalar variables can often be stored in registers, eliminating the need for costly main memory accesses. Global register allocation (GRA) attempts to store commonly-referenced scalar variables in registers throughout the code in which they are most frequently accessed. Consider the following Fortran example:

```
DO I = 1, N
  A(I) = X
  .
  .
  .
ENDDO
```

Here, X is referenced on every iteration of the loop. Eliminating loads and stores of X to main memory can substantially improve performance. Using GRA, the compiler generates code equivalent to that shown below:

```
REG = X
DO I = 1, N
  A(I) = REG
  .
  .
  .
ENDDO
X = REG
```

Where REG represents a register.

Optimizations

The analogous C example follows:

```
for(i=0;i<n;i++) {
  a[i] = x;
   .
   .
   .
}
```

After GRA:

```
REG = x;
for(i=0;i<n;i++){
  a[i] = REG;
   .
   .
   .
}
x = REG;
```

The compiler automatically determines which scalar variables are the best candidates for GRA and allocates registers accordingly.

GRA can sometimes cause wrong answers in Fortran code that violate ANSI standard argument-passing conventions. The problem arises when a constant is passed into a subroutine that potentially attempts to assign to it. If such an assignment executes, a bus error will occur; however, some codes may conditionally execute the assignment based on whether the argument is a variable or constant in the calling program.

In such cases, GRA may be unaware that the subroutine's dummy arguments may represent constant actual arguments. If the dummy arguments are allocated registers, a bus error will result even if the executable never attempts to assign to the constant, because GRA always generates code to store the register back to the dummy argument variable at some point.

Consider the following Fortran example:

```
.
.
.
CALL ASSIGNER(1,IMAX,A)
CALL ASSIGNER(0,10,A)
.
.
.
SUBROUTINE ASSIGNER(IVAR,I,A)
INTEGER A(100,100)
DO ICNT = 1,10
  IF (TEST(IVAR) .EQ. 1) THEN
    .
    .
    I = ...
    .
    .
  ELSE
    .
    .   ! NO ASSIGNMENT TO I
    .
  ENDIF
ENDDO
RETURN
END

INTEGER FUNCTION TEST(ITEST)
.
.
.
```

Here, the `IVAR` argument indicates whether the `I` dummy argument is a variable or constant. If it is a variable, it is assigned in the `IF` statement.

GRA, unaware that `I` may be a constant, allocates a register for it (assuming nothing in the missing code prevents it from doing so). When the register is stored back to `I` at runtime, a runtime error results.

Optimizations

The following Fortran example shows code that is equivalent to code produced by the GRA optimization for the subroutine ASSIGNER:

```
SUBROUTINE ASSIGNER(IVAR,I,A)
INTEGER A(100,100)
REG = I              ! I PLACED IN REGISTER
DO ICNT = 1,10
  IF (TEST(IVAR) .EQ. 1) THEN
    .
    .
    I = ...
    .
    .
  ELSE
    .
    .   ! NO ASSIGNMENT TO I
    .
  ENDIF
ENDDO
I = REG ! REG STORED BACK TO I--IMPOSSIBLE
RETURN  ! IF I IS A CONSTANT
END
```

This situation can be avoided by specifying the -nga compiler option. -nga disables global register allocation for arguments passed by reference. If the above example is compiled with -nga, I will never be allocated a register, and the code will work as expected.

GRA can also sometimes cause problems when parallel threads attempt to update a shared variable that has been allocated a register. In this case, each parallel thread will allocate a register for the shared variable; it is then unlikely that the copy in main memory will be updated correctly as each thread executes.

Parallel assignments to the same shared variables from multiple threads make sense only if the assignments are contained inside critical or ordered sections, or are executed conditionally based on thread ID. GRA will not allocate registers for shared variables that are assigned within critical or ordered sections, as long as the sections are implemented using compiler directives or sync_routine-defined functions (refer to Chapter 6, "Advanced shared-memory programming"). However, for conditional assignments based on thread ID, GRA may allocate registers that may cause wrong answers when stored.

In such cases, GRA can be disabled only for shared variables that are visible to multiple threads by specifying the -ngs compiler option.

In procedures with large numbers of loops, GRA can contribute to long compile times; therefore, GRA is only performed if the number of loops in the procedure is below a predetermined limit. You can remove this limit (and possibly significantly increase compile time and compile-time memory usage) by specifying the `-mrl` (more replicated loops) compiler option.

GRA is enabled by default at optimization levels `-O1` and higher.

-O2 Level optimizations

The primary goal of `-O2` optimizations is *data localization*. Data localization keeps heavily used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

Loops that manipulate arrays are the main candidates for localization optimizations. Most of these loops are eligible for the various transformations the compiler performs at level `-O2` to achieve localization. These transformations are explained in detail in this section.

Many loop transformations cause loops to be fully or partially replicated. Because unlimited loop replication can significantly increase compile times, the total number of loops replicated by all transformations is limited by default. Each optimization is allocated a percentage of this limit; if an optimization hits its limit, an advisory is issued and the optimization is disabled for any following loops. You can increase this limit (and possibly increase your program's compile time and compile-time memory usage) by specifying the `-mrl` compiler option.

In addition to localization, the compiler performs global instruction scheduling and all basic block and machine-instruction level optimizations at optimization level `-O2`.

Note

Most of the subsections that follow present code examples that demonstrate the optimization in question by showing the original code first and optimized code second. While the optimized code is shown in the same language as the original code, this is for illustrative purposes only.

Optimizations

Why localize?

The benefits of localization are best illustrated through an example.

Consider the following Fortran code:

```
REAL*4 A(100, 100), B(100, 100), C(100)
COMMON /BLK1/ A, B, C
...
DO J = 1, 100
  DO I = 1, 100
    A(I, J) = B(I, J) * C(I)
  ENDDO
ENDDO
```

The compiler will recognize that this loop nest is too small to benefit from blocking. As the loop is written, all three arrays can be reused.

First, notice that this loop operates on three `REAL*4` arrays, which total less than 80 kbyte in size. All the elements of these arrays will fit easily into the processor data cache, so elements being overwritten will not be a problem. The arrays are stored in `COMMON` to eliminate the possibility of cache thrashing, as described in Chapter 2, "Architecture overview."

Consider how these arrays are stored in memory. Fortran arrays are stored in column-major order, meaning that the elements of the first column are stored contiguously, followed by the elements of the second column and so on, as shown in Figure 11. Note that contiguous storage of the arrays in this illustration is due to the `COMMON` block storage of the arrays.

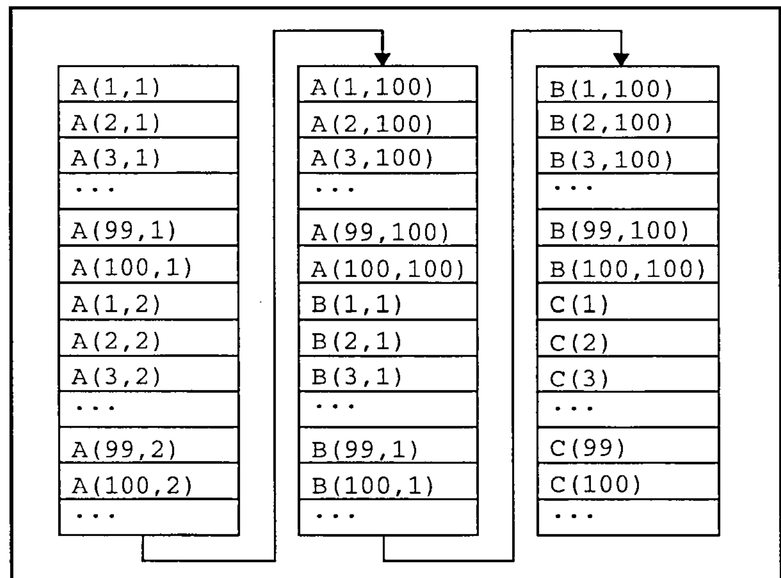


Figure 11 Array storage in memory

The boxes in Figure 11 represent contiguous virtual addresses; these could map to physical addresses in the CTIcache or in main memory. In any case, assuming the arrays have not been accessed before we reach the `J` loop and therefore are not in the processor data cache, they must be encached there as the loop executes.

On the first iteration of the loop, `A(1,1)`, `B(1,1)` and `C(1)` will be fetched from memory as part of separate 32-byte cache lines. These elements will be positioned at random relative to the cache line boundaries; in other words, there is no way of knowing whether `A(1,1)` is the first element in its cache line, or if it is positioned elsewhere. While it is likely that each cache line will contain some reusable elements of the array it is fetching from, this is not guaranteed on the first fetch. However, when any present reusable elements are used and the second fetch from memory takes place, the cache line in question will begin with the element being fetched and also contain the 7 following elements. Thus, after the first element is fetched for each array, fetching `A(I,1)` will encache `A(I:I+7,1)`; fetching `B(I,1)` will encache `B(I:I+7,1)` and fetching `C(I)` will encache `C(I:I+7)`. The `I` loop will then iterate 7 more times without needing to look beyond the processor data cache for data, boosting performance significantly.

Another significant performance boost happens when the I loop finishes executing for J = 1. At this point, all of C has been encached. While each successive column of A and B will have to be fetched a cache line at a time from memory, C will always be available immediately from the processor data cache, saving, in this simple example, 13 out-of-cache memory accesses.

More complicated loops, such as the matrix multiply algorithm, and loops that manipulate much larger arrays can often be transformed by the compiler such that far more data reuse is possible, resulting in greater performance.

The following sections explain the loop transformations that aid data localization.

Strip mining

Strip mining is a fundamental -O2 transformation that is used by loop blocking and, in a sense, by parallelization.

Strip mining involves splitting a single loop into a nested loop; the resulting inner loop iterates over a section or *strip* of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's *strip length*.

Consider the following Fortran code:

```
DO I = 1, 10000
  A(I) = A(I) * B(I)
ENDDO
```

Strip mining this loop using a strip length of 1000 yields the following loop nest:

```
DO IOUTER = 1, 10000, 1000
  DO ISTRIP = IOUTER, IOUTER+999
    A(ISTRIP) = A(ISTRIP) * B(ISTRIP)
  ENDDO
ENDDO
```

In this loop, the strip length integrally divides the number of iterations, so the loop is evenly split up. If the iteration count was not an integral multiple of the strip length; e.g., if I went from 1 to 10500 rather than 1 to 10000, the final iteration of the strip loop would execute 500 iterations instead of 1000.

An analogous C example follows:

```
for(i=0;i<10000;i++)
  a[i] = a[i] * b[i];
```

After strip mining with a strip length of 1000:

```
for(iout=0;iout<10000;iout+=1000)
  for(istrip=iout;istrip<iout+1000;istrip++)
    a[istrip] = a[istrip] * b[istrip];
```

In and of itself, strip mining is not profitable. However, strip mining is essential to the highly profitable loop blocking optimization, which is described in a following section of this chapter.

Loop distribution

Loop distribution is another fundamental `-O2` transformation that is necessary for some more advanced transformations. These advanced transformations require that all calculations in a nested loop be performed inside the innermost loop. To facilitate this, loop distribution transforms complicated nested loops into several simple loops (or nests) that contain all computations inside the body of the innermost loop.

Consider the following Fortran code:

```
DO I = 1, N
  B(I, 1) = 0
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
  D(I) = E(I) + A(I)
ENDDO
```

Optimizations

Loop distribution creates three copies of the I loop, separating the nested J loop from the assignments to arrays B and D. In this way, all three assignments are moved to innermost loops, as shown in the following transformed code:

```
DO I = 1, N
  B(I, 1) = 0
ENDDO
DO I = 1, N
  DO J = 1, M
    A(I) = A(I) + B(I, J) * C(I, J)
  ENDDO
ENDDO
DO I = 1, N
  D(I) = E(I) + A(I)
ENDDO
```

An analogous C example follows:

```
for(i=0;i<n;i++) {
  b[i][0] = 0;
  for(j=0;j<m;j++)
    a[i] = a[i] + b[i][j] * c[i][j];
  d[i] = e[i] + a[i];
}
```

This loop is distributed as shown below:

```
for(i=0;i<n;i++)
  b[i][0] = 0;
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    a[i] = a[i] + b[i][j] * c[i][j];
for(i=0;i<n;i++)
  d[i] = e[i] + a[i];
```

Distribution can improve efficiency by reducing the number of memory references per loop iteration, and can reduce cache thrashing. It also creates more opportunities for interchange.

Loop distribution can be disabled for specific loops by specifying the no_distribute directive or pragma immediately before the loop. In Fortran, it has the following form:

```
C$DIR NO_DISTRIBUTE
```

In C:

```
#pragma _CNX no_distribute
```

Loop interchange

The compiler interchanges nested loops for the following reasons:

- To facilitate other transformations
- To relocate the loop that is the most profitable to parallelize so that it is outermost (at optimization level -O3 only)
- To optimize inner-loop memory accesses

Consider the Fortran matrix addition algorithm that follows:

```
DO I = 1, N
  DO J = 1, M
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

This loop accesses the arrays A, B and C row by row, which, in Fortran, is very inefficient. Interchanging the I and J loops, as shown in the following example, will facilitate column by column access.

```
DO J = 1, M
  DO I = 1, N
    A(I, J) = B(I, J) + C(I, J)
  ENDDO
ENDDO
```

Unlike Fortran, C accesses arrays in row-major order. An analogous example in C, then, employs an opposite nest ordering, as shown below.

```
for(j=0;j<m;j++)
  for(i=0;i<n;i++)
    a[i][j] = b[i][j] + c[i][j];
```

Interchange facilitates row-by-row access. The interchanged loop is shown below.

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    a[i][j] = b[i][j] + c[i][j];
```

Loop blocking

Loop blocking is a combination of strip mining and interchange that maximizes data localization. It is provided primarily to deal with nested loops that manipulate arrays that are too large to fit into the cache. Under certain circumstances, loop blocking allows reuse of these arrays by transforming the loops that manipulate them so that they manipulate strips of the arrays that fit into the cache. Effectively, a blocked loop accesses array elements in sections that are optimally sized to fit in the cache.

Data reuse

Data reuse is important to understand when discussing blocking. There are two types of data reuse associated with loop blocking:

- *spatial* reuse
- *temporal* reuse

Spatial reuse is using data that was encached as a result of fetching another piece of data from memory. Remember that on an SPP Series system, data is fetched by cache lines; 32 bytes of data is encached on every fetch. On the initial fetch of array data from memory within a stride-one loop, the requested item can be located anywhere in the 32 bytes, unless the array is properly aligned (refer to the section "Data alignment" on page 21). Starting with the second memory fetch (which will not happen until any usable elements obtained on the first fetch are used), the requested data is at the beginning of the cache line, and the rest of the cache line will contain subsequent array elements. For a `REAL*4` array, this means the requested element and the seven following elements are encached on each fetch after the first. If any of these seven elements could then be used, say on any subsequent iterations of the loop, the loop would be exploiting spatial reuse. For loops with strides greater than one, spatial reuse can still occur; however, the cache lines may contain fewer usable elements.

Temporal reuse is using the same data item on more than one iteration of the loop. An array element whose subscript does not change as a function of the iterations of a surrounding loop exhibits temporal reuse in the context of the loop.

Loops containing either temporal or spatial reuse are candidates for blocking. Blocking exploits spatial reuse by ensuring that once fetched, cache lines are not overwritten until their spatial reuse is exhausted. Temporal reuse is similarly exploited.

Reuse example

The following Fortran loop contains arrays that are candidates for both spatial and temporal reuse:

```
REAL*4 A(100,100), B(100,100), C(100)
COMMON /BLK1/ A, B, C
.
.
.
DO J = 1, 100
  DO I= 1, 100
    A(I,J) = B(J,I) + C(I)
  ENDDO
ENDDO
```

As written, this loop gives spatial reuse on the A and B arrays, and both spatial and temporal reuse on the C array. Because the arrays are in a COMMON block and each array is of a total length that is an integral multiple of the CTIcache line length (64 bytes), we know that each array will begin on a CTIcache line boundary. All cache lines fetched will be full of reusable data. Spatial reuse is achieved on the A array because every 8th iteration of the I loop fetches a cache line containing 8 of its elements; the 7 iterations between main memory accesses can proceed with virtually no load delays. This continues throughout the entire range of the J loop.

Similar spatial reuse is achieved on the B array. During the first iteration of J, every referenced element of B, along with its containing cache line, will be fetched from memory. All the elements contained in this cache line will be reusable on some subsequent iteration of one of the loops. On subsequent iterations of J, a cache line will be fetched from memory only if the required element was not previously encached, and all the elements it contains will be usable. However, keep in mind that fetches are a function of I and may occur for different J values (after J = 1) based on the value of I. Since B's row index is J, any unused encached elements are used on the subsequent iterations of J for a given value of I. Though the data is not used as immediately as it is for A, spatial reuse is still fully exploited.

Spatial reuse is similarly achieved on the C array, but only for J = 1. Assuming the loop is compiled as written, when the first iteration of J finishes, C is completely contained in the processor data cache and will remain there for the duration of J. C can then be temporally reused for every subsequent iteration of J.

This loop does not require blocking to achieve this reuse because the arrays occupy less than 80 kbytes altogether, so they fit easily into the cache. Spatial reuse is graphically illustrated for a similar but more realistic example in the following section.

Blocking example: simple loop

In order to achieve reuse in more realistic examples that manipulate arrays that will not all fit in the cache, blocking strip mines the inner loop. The new innermost loop has an iteration space selected so that all the array elements it references will fit into the cache without overwriting themselves. The new loop nest is then interchanged such that the innermost loop is "blocked" from overwriting its strips.

Consider the following Fortran example:

```
REAL*8 A(1000,1000),B(1000,1000)
REAL*8 C(1000),D(1000)
COMMON /BLK2/ A, B, C
.
.
.
DO J = 1, 1000
  DO I = 1, 1000
    A(I,J) = B(J,I) + C(I) + D(J)
  ENDDO
ENDDO
```

Here the array elements occupy nearly 16 Mbytes of memory, and blocking becomes quite profitable.

First the compiler strip mines the I loop:

```
DO J = 1, 1000
  DO IOUT = 1, 1000, IBLOCK
    DO I = IOUT, IOUT+IBLOCK-1
      A(I,J) = B(J,I) + C(I) + D(J)
    ENDDO
  ENDDO
ENDDO
```

IBLOCK is the block factor (it is also the strip mine length) the compiler chooses based on the size of the arrays and size of the cache. Note that this example assumes the chosen IBLOCK divides 1000 evenly.

Now the compiler interchanges the J and IOUT loops to block the I loop.

```
DO IOUT = 1, 1000, IBLOCK
  DO J = 1, 1000
    DO I = IOUT, IOUT+IBLOCK-1
      A(I,J) = B(J,I) + C(I) + D(J)
    ENDDO
  ENDDO
ENDDO
```

This new nest accesses IBLOCK rows of A and IBLOCK columns of B for every iteration of J. At every iteration of IOUT, the nest accesses 1000 IBLOCK-length columns of A (or an IBLOCK × 1000 chunk of A) and 1000 IBLOCK-width rows of B are accessed. This is illustrated in Figure 12.

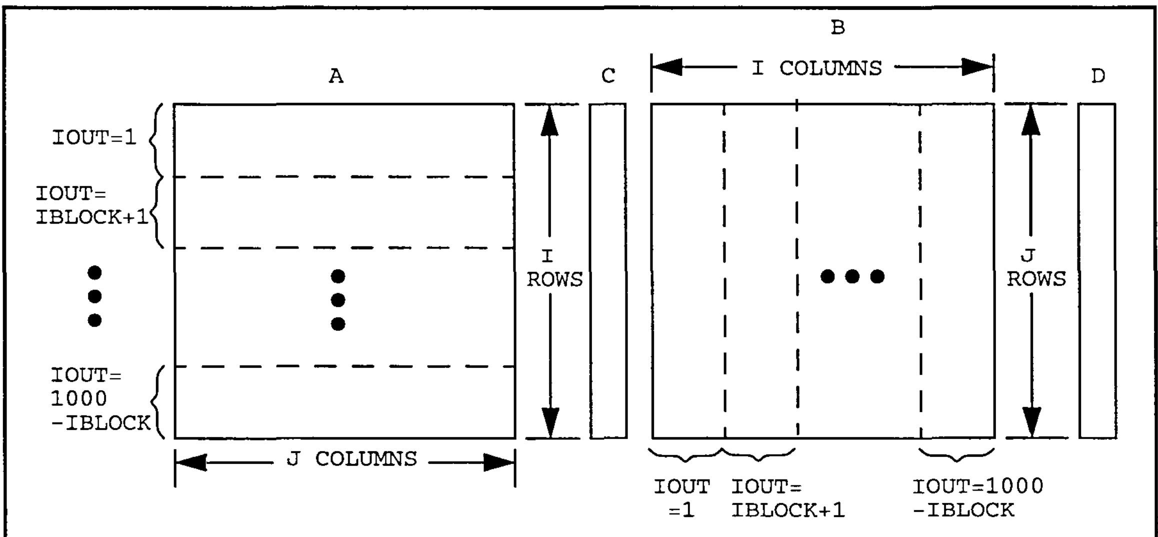


Figure 12 Blocked array access

Fetches of A encache the needed element and the three elements that are used in the three subsequent iterations, giving spatial reuse on A. Since the I loop traverses columns of B, fetches of B encache extra elements that will not be spatially reused until J increments. IBLOCK is chosen by the compiler to efficiently exploit spatial reuse of both A and B.

Figure 13 illustrates how cache lines of each array are fetched (A and B both start on cache line boundaries because they are in COMMON and are of lengths integrally divisible by the CTIcache line length). The shaded area represents the initial cache line fetched.

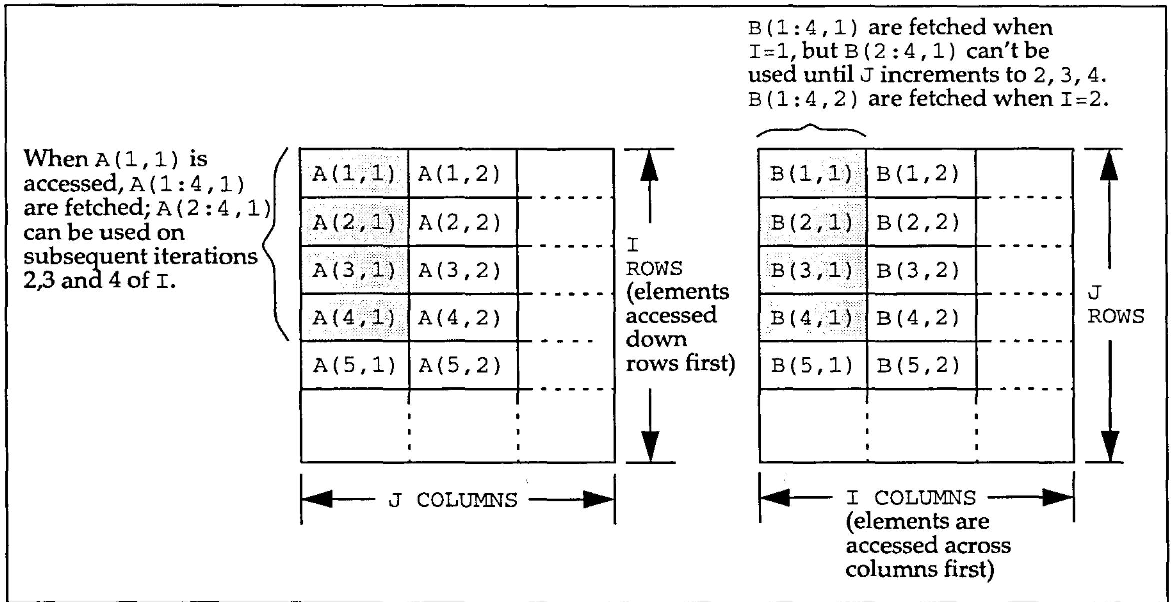


Figure 13 Spatial reuse of A and B

Typically, IBLOCK elements of C will remain in the cache for several iterations of J before being overwritten, giving temporal reuse on C for those iterations. By the time any of the arrays are overwritten, all spatial reuse has been exhausted. The load of D is hoisted out of the I loop so that it remains in a register for all iterations of I.

Blocking example: matrix multiply

The more complicated matrix multiply algorithm, which follows, is a prime candidate for blocking:

```
REAL*8 A(1000,1000),B(1000,1000),C(1000,1000)
COMMON /BLK3/ A, B, C
.
.
.
DO I = 1, 1000
  DO J = 1, 1000
    DO K = 1, 1000
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

This loop is blocked as shown below:

```
DO IOUT = 1, 1000, IBLOCK
  DO KOUT = 1, 1000, KBLOCK
    DO J = 1, 1000
      DO I = IOUT, IOUT+IBLOCK-1
        DO K = KOUT, KOUT+KBLOCK-1
          C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Here, we get:

- Spatial reuse of `B` with respect to the `K` loop
- Temporal reuse of `B` with respect to the `I` loop
- Spatial reuse of `A` with respect to the `I` loop
- Temporal reuse of `A` with respect to the `J` loop
- Spatial reuse of `C` with respect to the `I` loop
- Temporal reuse of `C` with respect to the `K` loop

An analogous C example follows:

```
static double a[1000][1000], b[1000][1000];
static double c[1000][1000];
.
.
.
for(i=0;i<1000;i++)
    for(j=0;j<1000;j++)
      for(k=0;k<1000;k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Convex C interchanges and blocks the loop in this example to provide optimal access efficiency for the row-major C arrays. The blocked loop is shown below:

```
for(jout=0;jout<1000;jout+=jblk)
  for(kout=0;kout<1000;kout+=kblk)
    for(i=0;i<1000;i++)
      for(j=jout;j<jout+jblk;j++)
        for(k=kout;k<kout+kblk;k++)
          c[i][j]=c[i][j]+a[i][k]*b[k][j];
```

As you can see, the interchange was done differently because of C's different array storage. This code yields:

- Spatial reuse of `b` with respect to the `j` loop
- Temporal reuse of `b` with respect to the `i` loop
- Spatial reuse of `a` with respect to the `k` loop
- Temporal reuse of `a` with respect to the `j` loop
- Spatial reuse on `c` with respect to the `j` loop
- Temporal reuse on `c` with respect to the `k` loop

Blocking is inhibited when loop interchange is inhibited. If a candidate loop nest contains loops that cannot be interchanged, blocking will not be performed.

Blocking directives, pragmas and options

Loop blocking can be disabled for specific loops using the NO_BLOCK_LOOP compiler directive and pragma. You can advise the compiler to use a specific block factor via the BLOCK_LOOP directive and pragma. In Fortran, these directives have the following form:

```
C$DIR NO_BLOCK_LOOP
C$DIR BLOCK_LOOP[(BLOCK_FACTOR = n)]
```

In C, these directives have the following form:

```
#pragma _CNX no_block_loop
#pragma _CNX block_loop[(block_factor = n)]
```

In the BLOCK_LOOP directive and pragma, *n* is the requested block factor, which must be an integer constant. The compiler will use this value as stated, so, for best performance, the block factor multiplied by the data type size of the data in the loop should be an integral multiple of the cache line size. In absence of the block_factor argument, this directive is useful for indicating which loop in a nest to block. In this case, the compiler will pick an appropriate block factor.

These directives affect the loop that immediately follows them.

Reconsider the matrix multiply example, this time with a BLOCK_LOOP directive.

```
      REAL*8 A(1000,1000),B(1000,1000)
      REAL*8 C(1000,1000)
      COMMON /BLK3/ A, B, C
      .
      .
      .
      DO I = 1,1000
        DO J = 1, 1000
C$DIR     BLOCK_LOOP(BLOCK_FACTOR = 112)
          DO K = 1,1000
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
          ENDDO
        ENDDO
      ENDDO
```

We know from the original example involving this code that the compiler blocks the I and K loops. In this example, the BLOCK_LOOP directive instructs the compiler to use a block factor of 112 for the K loop. This is an efficient blocking factor for this example because 112 × 8 bytes = 896 bytes, and 896/32 bytes (the cache line size) = 28, which is an integer, so partial cache lines will not be needed. The compiler-chosen value is still used on the I loop.

To disable blocking for all loops, use the -noblock compiler option. To specify a blocking factor for all loops, use the -blockloop *n* compiler option, where *n* is the blocking factor to be used for all loops that the compiler blocks.

As with all optimizations that replicate code, the number of new loops created when the compiler strip mines for blocking is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler will issue an advisory and blocking will be disabled for all following loops. To remove the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the -mrl compiler option.

Loop fusion

Loop fusion involves creating one loop out of two or more neighboring loops that have identical loop bounds and trip counts. This reduces loop overhead, memory accesses, and register usage, and can lead to other optimizations such as assignment substitution, redundant-assignment elimination and loop blocking. By potentially reducing the number of parallelizable loops in a program, loop fusion can greatly reduce parallelization overhead, since fewer spawns and joins will be necessary.

Consider the following Fortran code:

```
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
DO J = 1, N
  IF(A(J) .LT. 0) A(J) = B(J)*B(J)
ENDDO
```

These two loops can be fused into the following loop:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  IF(A(I) .LT. 0) A(I) = B(I)*B(I)
ENDDO
```

And this loop can be further optimized by applying assignment substitution and common subexpression elimination as shown in the following code:

```
DO I = 1, N
  TEMP1 = B(I)
  TEMP2 = TEMP1 + C(I)
  IF(TEMP2 .LT. 0) TEMP2 = TEMP1*TEMP1
  A(I) = TEMP2
ENDDO
```

Here assignment substitution is exploited when the array reference to `B(I)` is replaced by the scalar `TEMP1`, and the reference to `A(I)` is replaced by `TEMP2`. `TEMP2` is used to hold the value of `A(I)` until there is no chance the value will change, thus eliminating redundant assignments to `A(I)`.

An analogous C example follows:

```
for(i=0; i<n; i++)
  a[i] = b[i] + c[i];
for(j=0; j<n; j++)
  if(a[j] < 0) a[j] = b[j]*b[j];
```

After loop fusion, assignment substitution and common subexpression elimination, the compiler produces code equivalent to the following:

```
for(i=0;i<n;i++) {
  temp1 = b[i];
  temp2 = temp1 + c[i];
  if(temp2 < 0) temp2 = temp1*temp1;
  a[i] = temp2;
}
```

Occasionally loops that do not appear to be fusible become fusible as a result of compiler transformations that precede fusion. For instance, peeling or interchanging a loop may make it suitable for fusing with another loop.

Loop fusion is especially beneficial when applied to Fortran 90 array assignments. The compiler translates these statements into loops; when such loops do not contain code that would inhibit fusion, they can be fused.

Consider the following Fortran example:

```
REAL A(1000,1000), B(1000,1000), C(1000,1000)
.
.
.
C = 2.0 * B
A = A + B
```

The compiler would first transform these Fortran 90 array assignments into loops, generating code similar to that shown below.

```
DO TEMP1 = 1, 1000
  DO TEMP2 = 1, 1000
    C(TEMP2, TEMP1) = 2.0 * B(TEMP2, TEMP1)
  ENDDO
ENDDO
DO TEMP3 = 1, 1000
  DO TEMP4 = 1, 1000
    A(TEMP4,TEMP3)=A(TEMP4,TEMP3)+B(TEMP4,TEMP3)
  ENDDO
ENDDO
```

These two loops would then be fused as shown in the following loop nest:

```
DO TEMP1 = 1, 1000
  DO TEMP2 = 1, 1000
    C(TEMP2,TEMP1) = 2.0 * B(TEMP2, TEMP1)
    A(TEMP2,TEMP1)=A(TEMP2,TEMP1)+B(TEMP2,TEMP1)
  ENDDO
ENDDO
```

And the compiler can apply further optimizations to this new nest.

Loop fusion is enabled by default and by the `-fl` compiler option. It can be disabled by specifying the `-nfl` compiler option. The `no_fuse` compiler directive and pragma can be used to disable loop fusion for specific loops. In Fortran, it has the following form:

```
C$DIR NO_FUSE
```

In C, it has the following form:

```
#pragma _CNX no_fuse
```

`no_fuse` is effective for the immediately following loop only.

When fusion is disabled for the program, you can request it for particular loops using the `prefer_fuse` directive and pragma. In Fortran, it has the following form:

```
C$DIR PREFER_FUSE
```

In C, it has the following form:

```
#pragma _CNX prefer_fuse
```

A `prefer_fuse` directive or pragma must immediately precede the loop for which you want to enable fusion. If fusion is enabled for the program, `prefer_fuse` has no affect. If fusion is disabled for the program, `prefer_fuse` must be specified on two or more neighboring loops, and will override the `-nfl` option.

Loop unrolling

Loop unrolling involves increasing a loop's step value and replicating the loop body, with each replication appropriately offset from the induction variable so that all iterations are performed given the new step.

Unrolling can be total or partial. Total unrolling involves eliminating the loop structure completely, replicating the loop body a number of times equal to the iteration count, and replacing the iteration variable with constants. This only makes sense for

loops with small iteration counts. Consider the following Fortran example:

```
DO I = 1, 4
  A(I) = B(I) + C(I)
ENDDO
```

This loop is completely unrolled as shown in the following example:

```
A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
A(3) = B(3) + C(3)
A(4) = B(4) + C(4)
```

Partial unrolling is performed on loops with larger or unknown iteration counts. It retains the loop structure, but replicates the body a number of times equal to the *unrolling depth* (also known as the *unroll factor*) and adjusts references to the iteration variable accordingly.

Consider the following Fortran example:

```
DO I = 1, 100
  A(I) = B(I) + C(I)
ENDDO
```

This example can be unrolled to a depth of four as shown below:

```
DO I = 1, 100, 4
  A(I) = B(I) + C(I)
  A(I+1) = B(I+1) + C(I+1)
  A(I+2) = B(I+2) + C(I+2)
  A(I+3) = B(I+3) + C(I+3)
ENDDO
```

Each iteration of the loop now computes four values of A instead of 1 value.

An analogous C example follows:

```
for(i=0;i<100;i++)
  a[i] = b[i] + c[i];
```

This can be unrolled to a depth of four as shown in the following code:

```
for(i=0;i<100;i+=4) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
```

Loop unrolling improves efficiency by eliminating loop overhead; in totally unrolled loops, the overhead is completely eliminated, and in partially unrolled loops the number of tests and induction variable increments are reduced. Unrolling can also create opportunities for other optimizations, such as improved register use and more efficient scheduling.

Convex compilers perform loop unrolling by default at optimization levels `-O2` and higher. The compiler completely unrolls loops with iteration counts determinable at compile time to be less than 5. Loops with undeterminable iteration counts or determinable counts of 5 or more are partially unrolled.

Unrolling is enabled by default and through use of the `-ur` compiler option. It can be disabled for all loops by specifying the `-nur` compiler option. You can specify an unroll factor by using the `-urn` *n* option, where *n* is the unroll factor for all unrolled loops.

When `-nur` is used, unrolling can also be specified for individual loops using the `unroll` directive and pragma. In Fortran, this directive has the following form:

```
C$DIR UNROLL[(UNROLL_FACTOR=n)]
```

The C pragma has the following form:

```
#pragma _CNX unroll[(unroll_factor=n)]
```

Where the optional `unroll_factor=`*n* argument allows you to specify an unroll factor of *n* for the loop in question.

When unrolling is enabled for the entire program, it can be disabled for individual loops using the `no_unroll` directive and pragma. In Fortran, this directive has the following form:

```
C$DIR NO_UNROLL
```

The C pragma has the following form:

```
#pragma _CNX no_unroll
```

Note

Unrolling is only performed on innermost loops. If you use `unroll` or `no_unroll` on a loop nest, you must specify it on the loop which ends up, after any interchanges performed by the compiler, to be innermost.

To ensure that the directive or pragma is on the innermost loop, compile the nest in question and determine from the optimization report (which is discussed in the "Optimization report" appendix) which loop is innermost after compilation, then place the directive or pragma on that loop in your source. Placing the directive or

pragma on a loop that is not innermost after compilation will have no effect.

As with all optimizations that replicate code, the number of new loops created when the compiler performs the unroll optimization is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler issues an advisory and unrolling is disabled for all following loops. To remove the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the `-mrl` compiler option.

Loop unroll and jam

The loop unroll and jam transformation is primarily intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory accesses and sometimes allows better exploitation of certain machine instructions.

Unroll and jam involves partially unrolling one or more loops higher in the nest than the innermost loop, and fusing ("jamming") the resulting loops back together. For unroll and jam to be effective, a loop must be nested and must contain data references that can be temporally reused with respect to some loop other than the innermost.

Blocked loops are often prime candidates for unroll and jam because their deep nesting provides many candidates for temporal reuse with respect to the noninnermost loops.

Consider our blocked matrix multiply loop:

```
DO IOUT = 1, N, ISTRIP
  DO KOUT = 1, N, KSTRIP
    DO J = 1, N
      DO I = IOUT, IOUT+ISTRIP-1
        DO K = KOUT, KOUT+KSTRIP-1,
          C(I,J) = C(I,J) + A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Here, the compiler can exploit a maximum of 3 registers: one for `C(I,J)`, one for `A(I,K)`, and one for `B(K,J)`.

Optimizations

Register exploitation can be vastly increased on this loop by unrolling and jamming the J and I loops. First, the compiler unrolls these loops. To simplify the illustration, we will use an unrolling factor of 2 for both I and J. This is the number of times the contents of the loops will be replicated.

The following Fortran example shows this replication:

```
DO IOUT = 1, N, ISTRIP
 DO KOUT = 1, N, KSTRIP
  DO J = 1, N, 2
   DO I = IOUT, IOUT+ISTRIP-1, 2
     DO K = KOUT, KOUT+KSTRIP-1
       C(I,J)  = C(I,J)+A(I,K)*B(K,J)
     ENDDO
     DO K = KOUT, KOUT+KSTRIP-1
       C(I+1,J)  = C(I+1,J)+A(I+1,K)*B(K,J)
     ENDDO
   ENDDO
   DO I = IOUT, IOUT+ISTRIP-1, 2
     DO K = KOUT, KOUT+KSTRIP-1
       C(I,J+1)  = C(I,J+1)+A(I,K)*B(K,J+1)
     ENDDO
     DO K = KOUT, KOUT+KSTRIP-1
       C(I+1,J+1)  = C(I+1,J+1)+A(I+1,K)*B(K,J+1)
     ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

The "jam" part of unroll and jam occurs when the loops are fused back together, to create the following:

```
DO IOUT = 1, N, ISTRIP
 DO KOUT = 1, N, KSTRIP
  DO J = 1, N, 2
   DO I = IOUT, IOUT+ISTRIP-1, 2
     DO K = KOUT, KOUT+KSTRIP-1
      C(I,J)  = C(I,J)+A(I,K)*B(K,J)
      C(I+1,J)  = C(I+1,J)+A(I+1,K)*B(K,J)
      C(I,J+1)  = C(I,J+1)+A(I,K)*B(K,J+1)
      C(I+1,J+1)  = C(I+1,J+1)+A(I+1,K)*B(K,J+1)
     ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

This new loop exploits more registers and requires fewer loads and stores than the original. Recall that the original blocked loop could use no more than 3 registers. This unrolled-and-jammed loop can use 8, one for each of the following references:

`C(I,J)`	`A(I,K)`	`B(K,J)`	`C(I+1,J)`
`A(I+1,K)`	`C(I,J+1)`	`B(K,J+1)`	`C(I+1,J+1)`

Fewer loads and stores per operation are required because all of the registers containing these elements are referenced at least twice. This particular example can also benefit from the PA-RISC `FMPYADD` instruction, which doubles the speed of the operations in the body of the loop by simultaneously performing unrelated adds and multiplies.

Remember, this is a very simplified example. In reality, the compiler attempts to exploit as many of the PA-RISC processor's registers as possible. For the matrix multiply algorithm used here, the compiler would pick a larger unrolling factor, creating a much larger `K` loop body. This would result in increased register exploitation and fewer loads and stores per operation.

Unroll and jam is enabled by default and through use of the `-uj` option at optimization levels `-O2` and higher. You can disable unroll and jam by specifying the `-nuj` option. You can specify an unrolling factor by using the `-ujn` *n* option, where *n* is the desired unrolling factor for all loops which the compiler unrolls and jams.

You can specify unroll and jam for individual loops by using the `unroll_and_jam` compiler directive and pragma. In Fortran, it has the following form:

```
C$DIR UNROLL_AND_JAM[(UNROLL_FACTOR=n)]
```

The C pragma has the following form:

```
#pragma _CNX unroll_and_jam[(unroll_factor=n)]
```

Where the optional `unroll_factor=`*n* argument allows you to specify an unroll factor for the loop in question.

You can disable unroll and jam for individual loops using the `no_unroll_and_jam` directive and pragma. In Fortran, it has the following form:

```
C$DIR NO_UNROLL_AND_JAM
```

The C pragma has the following form:

```
#pragma _CNX no_unroll_and_jam
```

Note

All of these directives and pragmas apply to the immediately following loop nest. Since unroll and jam is only performed on nested loops, you must ensure that the directive or pragma is specified on a loop that, after any compiler-initiated interchanges, is not the innermost loop. You can determine which loop in a nest will be innermost by compiling the nest without any directives and examining the optimization report.

As with all optimizations that replicate code, the number of new loops created when the compiler performs the unroll and jam optimization is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler issues an advisory and unroll and jam is disabled for all following loops. To remove the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the `-mrl` compiler option.

IF-DO and if-for optimizations

These optimizations modify loops containing tests to improve performance. Tests can be promoted out of the loops or eliminated completely. By minimizing the number of tests within a loop, the compiler reduces the number of branches that must be executed, thereby improving performance.

In Fortran, these optimizations are performed on DO loops containing IF statements; in C, they are performed on for loops containing if statements. These optimizations fall into one of three categories:

- Redundant test elimination
- Loop peeling
- Test promotion

Each of these is described in detail below.

Redundant-test elimination

Redundant-test elimination is the simplest of these optimizations. The compiler recognizes when a test against some index variable is evaluated more than once and eliminates that test as well as any accompanying redundant code.

This optimization is especially relevant when you are optimizing FORTRAN 66 programs that contain DO loops surrounded by IF tests, as shown in the following example:

```
DO I = 1, N
  IF (I .GT. 0) THEN
    DO J = 1, I
      A(I,J) = 0
    ENDDO
  ENDIF
ENDDO
```

With the test removed, the loop looks like this:

```
DO I = 1, N
  DO J = 1, I
    A(I,J) = 0
  ENDDO
ENDDO
```

Here the explicit test, `IF (I .GT. 0)`, is redundant because the test is implicit in the DO loop. It is therefore removed during redundant-test elimination.

Similarly, the test in the following C code always succeeds:

```
for (i=0; i<n; i++)
  if (i>-1)
    a[i] += b[i];
```

The `if` test is redundant because of the loop control, which initializes `i` to 0 and increments it by one each iteration. The compiler recognizes that the condition always occurs and removes the test:

```
for (i=0; i<n; i++)
  a[i] += b[i];
```

Redundant-test elimination is always performed at optimization levels `-O2` and above.

Loop boundary-value peeling

Loop boundary-value peeling involves removing the first iteration, last iteration, or first and last iterations of a loop to remove conditional tests from the loop. This is done when the loop contains a test, or tests, involving an explicit reference to the loop index variable that, when evaluated, either always causes or always prevents execution of the first iteration, last iteration, or both. Last-iteration peeling is also sometimes necessary when the compiler automatically privatizes a loop variable which is referenced after the loop. In this case, peeling is used to assign the last-iteration value to the private variable so that it can be later referenced.

Given the Fortran code shown below, the compiler automatically peels off the first and last tests and rewrites the loop to cover the remaining indexes:

```
DO I = 1, 100
  IF (I .EQ. 1) THEN
    A(I) = B(I)
  ELSE IF (I .EQ. 100) THEN
    A(I) = C(I)
  ELSE
    A(I) = -A(I)
  ENDIF
ENDDO
```

After peeling, the code looks like this:

```
A(1) = B(1)
DO I = 2, 99
  A(I) = -A(I)
ENDDO
A(100) = C(100)
```

The analogous C code follows:

```
for (i=0; i<100; i++)
  if (i==0)
    a[i]=b[i];
  else
  if (i==99)
    a[i]=c[i];
  else
    a[i]= -a[i];
```

After peeling:

```
a[0]= b[0];
for (i=1; i<99; i++)
  a[i]=-a[i];
a[99]=c[99];
```

In some cases, boundary-value peeling requires replicating large amounts of code, and this can greatly increase the size of the executable file. By default, the compiler peels boundary values and expands the code up to a predetermined conservative limit; you can increase this limit by using the `-peel` compiler option or (if you wish to do so on a loop-by-loop basis) by using the `PEEL` Fortran compiler directive or `peel` C pragma.

You can allow the compiler to expand code without bound by using the `-peelall` compiler option, the `PEEL_ALL` Fortran directive, or the `peel_all` C pragma. In codes containing large numbers of boundary-value operations, allowing code expansion without bound can greatly lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

Using any of the compiler options, directives or pragmas described above increases *only* the loop replication limit for the loop peeling optimization; it does not increase the limit on the total number of replicated loops. This leaves a smaller percentage of the total limit for other loop-replicating options. To increase the total limit, specify the `-mrl` compiler option.

Boundary-value peeling can be disabled completely with the `-nopeel` compiler option. Similarly, you can disable peeling on a loop-by-loop basis with the `NO_PEEL` Fortran compiler directive or the `no_peel` C pragma. See Appendix A, "Compiler directives and pragmas," for more information.

Note

Loop boundary-value peeling is not performed on loops that have no tests on boundary values. In other words, the compiler does not try to peel unpeelable loops.

Test promotion

Test promotion involves promoting a test out of the loop that encloses it by replicating the containing loop(s) for each branch of the test. The replicated loops contain fewer tests than the originals, or no tests at all, so the loops execute much faster. Multiple tests can be promoted, and copies of the loop are made for each test.

Consider the following Fortran loop:

```
DO I = 1, N
  IF (FOO .EQ. BAR) THEN
    A(I) = B(I)
  ELSE
    A(I) = 0
  ENDIF
ENDDO
```

Test promotion produces the following code:

```
IF (FOO .EQ. BAR) THEN
  DO I = 1, N
    A(I) = B(I)
  ENDDO
ELSE
  DO I = 1, N
    A(I) = 0
  ENDDO
ENDIF
```

In C:

```
for(i=0; i<n; i++)
  if (foo==bar)
    a[i]=b[i];
  else
    a[i]=0;
```

After test promotion:

```
if (foo==bar)
  for(i=0; i<n; i++)
    a[i]=b[i];
else
  for (i=0; i<n; i++)
    a[i]=0;
```

For loops containing large numbers of tests, loop replication can greatly increase the size of the code.

You can control the amount of code replication and test promotion with compiler options and directives. By default, the compiler

Optimizations

promotes tests and replicates code up to a predetermined conservative limit.

The -ptst compiler option increases this limit and can cause a noticeable increase in compile time.

The -ptstall option promotes all tests regardless of code replication. This can cause a large increase in compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables.

The -noptst option disables test promotion.

The PROMOTE_TEST, PROMOTE_TEST_ALL and NO_PROMOTE_TEST Fortran compiler directives and promote_test, promote_test_all and no_promote_test C pragmas provide similar functionality on a loop-by-loop basis. See Appendix A, "Compiler directives and pragmas," for more information about these directives.

Using -ptst, -ptstall, -promote_test, -promote_test_all, -PROMOTE_TEST, or -PROMOTE_TEST_ALL will *only* increase the loop replication limit for the test promotion optimization; they will not increase the limit on the total number of replicated loops. This leaves a smaller percentage of the total limit for other loop-replicating options. To increase the total limit, specify the -mrl compiler option.

At optimization levels -O2 and above, the Convex compilers automatically perform these optimizations on Fortran DO and hand-rolled loops, and on for, while, do-while and hand-rolled loops in C. Simple and nested loops, and loops with exits, are handled.

Scalar replacement

Scalar replacement involves storing loop-invariant array elements in scalars, which can then be stored in registers by global register allocation. This substantially increases the performance of the loop by eliminating the necessity of accessing the array element in main memory or the cache.

Consider the following Fortran loop:

```
DO I = 1, N
  DO J = 1, M
    A(I) = A(I) + B(J)
  ENDDO
ENDDO
```

Here, `A(I)` is invariant with respect to the `J` loop. The compiler can therefore replace `A(I)` with a scalar before entering the `J` loop, and store the scalar back to `A(I)` between iterations of `I`. Global register allocation can then assign this scalar to a register for the duration of the iteration, further improving efficiency.

Employing scalar replacement with global register allocation, the compiler generates code equivalent to the following Fortran:

```
DO I = 1, N
  REG = A(I)    ! PLACE A(I) IN REGISTER
  DO J = 1, M
    REG = REG + B(J)
  ENDDO
  A(I) = REG    ! STORE REGISTER BACK TO A(I)
ENDDO
```

Here, the variable "REG" represents a register.

An analogous C example follows:

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    a[i] = a[i] + b[i];
```

After scalar replacement:

```
for(i=0;i<n;i++) {
  reg = a[i];
  for(j=0;j<m;j++
    reg = reg + b[i];
  a[i] = reg;
}
```

Scalar replacement increases performance for this kind of loop by about 50%.

Optimizations

The compiler does not attempt scalar replacement if the loop-invariant array element is aliased or if the array element is contained in conditional code.

At optimization level `-O2` and above, scalar replacement is enabled by default and through use of the `-sr` compiler option; it can be disabled by the `-nsr` compiler option.

Inhibitors of localization

Any of the following conditions can inhibit or prevent data localization:

- Loop-carried dependences
- Aliased scalar or array variables
- Computed or assigned `GOTO` statements in Fortran
- Multiple loop entries or exits
- `RETURN` or `STOP` statements in Fortran
- Procedure calls
- I/O statements

The following sections discuss these conditions and their effects on data localization.

Loop-carried dependences

A loop-carried dependence (LCD) exists when one iteration of a loop assigns a value to an address that is referenced or assigned on another iteration. In some cases, LCDs can inhibit loop interchange, thereby inhibiting localization. Typically, these cases involve array indexes which are offset in opposite directions. The Fortran loop below contains an interchange-inhibiting LCD:

```
DO I = 2, M
  DO J = 1, N
    A(I,J) = A(I-1,J+1) + B(I,J)
  ENDDO
ENDDO
```

C loops can contain similar constructs, but to simplify illustration, we will only consider this Fortran example.

As written, this loop uses `A(I-1,J+1)` to compute `A(I,J)`. Table 3 shows the sequence in which values of `A` are computed for this loop.

Table 3 Computation sequence of A(I,J): original loop

I	J	A(I,J)	A(I-1,J+1)
2	1	A(2,1)	A(1,2)
2	2	A(2,2)	A(1,3)
2	3	A(2,3)	A(1,4)
...	...	...	...
3	1	A(3,1)	A(2,2)
3	2	A(3,2)	A(2,3)
3	3	A(3,3)	A(2,4)
...	...	...	...

As enumerated in Table 3, the original loop computes the elements of the current row of A using the elements of the previous row of A. For all rows except the first (which is never written), the values contained in the previous row must be written before the current row is computed. This dependence must be honored for the loop to yield its intended results. If a row element of A is computed before the previous row element is computed, the result will be incorrect.

Interchanging the I and J loops yields the following code:

```
DO J = 1, N
  DO I = 2, M
    A(I,J) = A(I-1,J+1) + B(I,J)
  ENDDO
ENDDO
```

After interchange, the loop computes values of A in the sequence shown in Table 4.

Table 4 Computation sequence of `A(I,J)`: interchanged loop

I	J	A(I,J)	A(I-1,J+1)
2	1	A(2,1)	A(1,2)
3	1	A(3,1)	A(2,2)
4	1	A(4,1)	A(3,2)
...	...	...	...
2	2	A(2,2)	A(1,3)
3	2	A(3,2)	A(2,3)
4	2	A(4,2)	A(3,3)
...	...	...	...

Here, the elements of the current column of `A` are computed using the elements of the *next* column of `A`.

The problem here is that columns of A are being computed using elements that have not been written yet, which violates the dependence illustrated in Table 3. The element-to-element dependences in both the original and interchanged loop are illustrated in Figure 14.

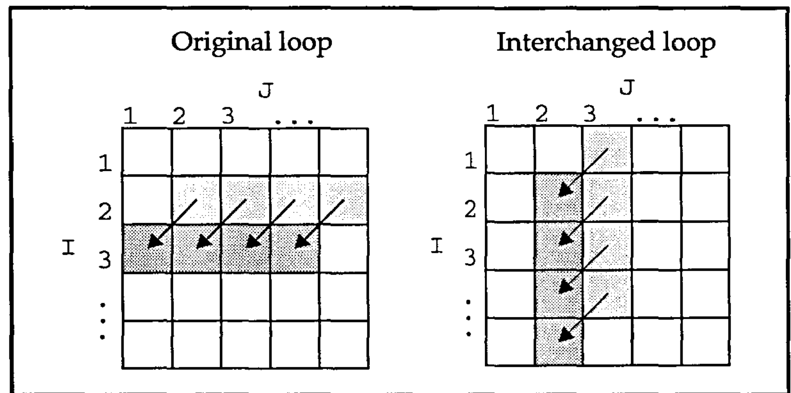


Figure 14 LCDs in original and interchanged loops

The arrows in Figure 14 represent dependences from one element to another; the arrows point at elements that depend on the elements at the arrows' bases. Shaded elements indicate a typical row or column computed in the inner loop:

- Lightly shaded elements have not been computed yet.
- Darkly shaded elements have already been computed.

This figure helps to illustrate the sequence in which the array elements are cycled through by the respective loops: the original

loop cycles across all the columns in a row, then moves on to the next row; the interchanged loop cycles down all the rows in a column first, then moves on to the next column.

Interchange is only inhibited by loops that contain dependences that change when the loop is interchanged. Most LCDs do not fall into this category and thus do not inhibit data localization.

Occasionally the compiler encounters an apparent LCD. If it cannot determine whether the LCD actually inhibits interchange, it conservatively avoids interchanging the loop.

The following Fortran example illustrates this situation:

```
DO I = 1, N
  DO J = 2, M
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
  ENDDO
ENDDO
```

An analogous C example follows:

```
for(j=0;j<n;j++)
  for(i=1;i<m;i++)
    a[i][j] = a[i+IADD][j+JADD] + b[i][j];
```

In these examples, if `IADD` and `JADD` are either both positive or both negative, the loop contains no interchange-inhibiting dependence. However, if one and only one of the variables is negative, interchange is inhibited. The compiler has no way of knowing the runtime values of `IADD` and `JADD`, so it will avoid interchanging the loop. If you are sure the `IADD` and `JADD` will either both be negative or both be positive, you can indicate to the compiler that the loop is free of dependences using the `NO_LOOP_DEPENDENCE` compiler directive or pragma. In Fortran, this directive has the following form:

`C$DIR NO_LOOP_DEPENDENCE(`*namelist*`)`

The `no_loop_dependence` C pragma has the form:

`#pragma _CNX no_loop_dependence(`*namelist*`)`

Where *namelist* is a comma-delimited list of variables and/or arrays that have no dependences for the immediately following loop.

Optimizations

The previous Fortran loop can be interchanged when the NO_LOOP_DEPENDENCE directive is specified for A and B on the J loop as shown in the following code:

```
      DO I = 1, N
C$DIR   NO_LOOP_DEPENDENCE(A)
        DO J = 2, M
          A(I,J) = A(I+IADD,J+JADD) + B(I,J)
        ENDDO
      ENDDO
```

The no_loop_dependence pragma can similarly be used on the C loop:

```
for(i=0;i<n;i++)
#pragma _CNX no_loop_dependence(a)
  for(j=1;j<m;j++)
    a[i][j] = a[i+IADD][j+JADD] + b[i][j];
```

If either IADD or JADD (but not both) acquire negative values at runtime, these loops may result in incorrect answers.

Dependences and loop fusion

In some cases, loop fusion is also inhibited by simpler dependences than those that inhibit interchange. Consider the following Fortran example:

```
DO I = 1, N-1
  A(I) = B(I+1) + C(I)
ENDDO
DO J = 1, N-1
  D(J) = A(J+1) + E(J)
ENDDO
```

It would appear that this loop would profit from fusion. Fusing it would yield the following incorrect code:

```
DO ITEMP = 1, N-1
  A(ITEMP) = B(ITEMP+1) + C(ITEMP)
  D(ITEMP) = A(ITEMP+1) + E(ITEMP)
ENDDO
```

This loop produces different answers than the original loops, because the reference to A(ITEMP+1) in the fused loop accesses a value that has not been assigned yet, while the analogous reference to A(J+1) in the original J loop accesses a value that was assigned in the original I loop.

An analogous C example follows:

```
for(i=0; i<n-1; i++)
  a[i] = b[i+1] + c[i];
for(j=0; j<n-1; j++)
  d[j] = a[j+1] + e[j];
```

After fusion:

```
for(itemp=0; itemp<n-1; itemp++) {
  a[itemp] = b[itemp+1] + c[itemp];
  d[itemp] = a[itemp+1] + e[itemp];
}
```

Aliasing

An *alias* is an alternate name for some object. Aliasing occurs in a program when two or more names are attached to the same memory location. Aliasing is typically caused in Fortran by use of the EQUIVALENCE statement and in C by use of pointers. Passing identical actual arguments into different dummy arguments in a Fortran subprogram can also cause aliasing, as can passing the same address into different pointers in a C function.

Aliasing interferes with data localization because it can mask LCDs, as shown in the following Fortran example, where the arrays A and B have been EQUIVALENCEd:

```
INTEGER A(100,100), B(100,100), C(100,100)
EQUIVALENCE(A,B)
.
.
.
DO I = 1, N
  DO J = 2, M
    A(I,J) = B(I-1,J+1) + C(I,J)
  ENDDO
ENDDO
```

This loop has the same problem as the loop used to demonstrate LCDs in the previous section; because A and B refer to the same array, the loop contains an LCD on A, which prevents interchange and thus interferes with localization.

The C equivalent of this loop follows. Keep in mind that C stores arrays in row-major order, which requires different subscripting to access the same elements.

```
int a[100][100], c[100][100], i, j;
int (*b)[100];
b = a;
.
.
.
for(i=1;i<n;i++){
   for(j=0;j<m;j++){
      a[j][i] = b[j+1][i-1] + c[j][i];
   }
}
```

C has nothing analogous to Fortran's `EQUIVALENCE` statement, but arrays can be effectively `EQUIVALENCE`d through the use of pointers, as shown.

Passing the same address into different dummy procedure arguments can yield the same result. Fortran passes arguments by reference while C passes them by value, but pass-by-reference can be simulated in C by passing the argument's address into a pointer in the receiving procedure.

The following Fortran code exhibits the same aliasing problem as the previous example, but the alias is created by passing the same actual argument into different dummy arguments.

Note **The following code violates the Fortran standard.**

```
.
.
.
CALL ALI(A,A,C)
.
.
.
SUBROUTINE ALI(A,B,C)
INTEGER A(100,100), B(100,100), C(100,100)
DO J = 1, N
  DO I = 2, M
    A(I,J) = B(I-1,J+1) + C(I,J)
  ENDDO
ENDDO
.
.
.
```

The following code shows the same argument-passing problem in C (this code is legal ANSI C):

```
.
.
.
ali(&a,&a,&c);
.
.
.
void ali(a,b,c)
int a[100][100], b[100][100], c[100][100];
{
  int i,j;
  for(j=0;j<n;j++){
    for(i=1;i<m;i++){
      a[j][i] = b[j+1][i-1] + c[j][i];
    }
  }
}
```

Multiple loop entries or exits

Loops that contain multiple entries or exits inhibit data localization because they cannot safely be interchanged. Extra loop entries are usually created when a loop contains a branch destination. Extra exits are more common; they are often created in C using the `break` statement and in Fortran using the `GOTO` statement.

Consider the following C code:

```
for(j=0;j<n;j++){
  for(i=0;i<m;i++){
    a[i][j] = b[i][j] + c[i][j];
    if(a[i][j] == 0) break;
    .
    .
    .
  }
}
```

Interchanging this loop would change the order in which the values of `a` are computed; the original loop computes `a` column-by-column, whereas the interchanged loop would compute it row-by-row. This means that the interchanged loop may hit the `break` statement and exit after computing a different set of elements than the original loop. Interchange therefore may cause the results of the loop to differ and must be avoided.

A similar loop construct written in Fortran follows:

```
      DO J = 1, M
        DO I = 1, N
          A(I,J) = B(I,J) + C(I,J)
          IF(A(I,J) .EQ. 0) GOTO 50
            .
            .
            .
        ENDDO
      ENDDO
        .
        .
        .
50    CONTINUE
```

Again, the order of computation changes if the loops are interchanged.

RETURN or STOP statements in Fortran

Like loops with multiple exits, RETURN and STOP statements in Fortran inhibit localization because they inhibit interchange. If a loop containing a RETURN or STOP is interchanged, its order of computation may change, giving wrong answers.

Computed or assigned GOTO statements in Fortran

When the Fortran compiler encounters a computed or assigned GOTO statement in an otherwise interchangeable loop, it cannot always determine whether the branch destination is within the loop. Since an out-of-loop destination would be a loop exit, these statements often prevent loop interchange and therefore data localization.

Procedure calls

The standard Convex compilers are unaware of the side effects of most procedures, and therefore cannot determine whether they might interfere with loop interchange. These side effects may include data dependences involving loop arrays, aliasing (as described in the section "Aliasing" on page 99), and processor data cache usage that conflicts with the loop's usage of the cache, rendering useless any data localization optimizations performed on the loop.

I/O statements

The order in which values are read into or written from a loop may change if the loop is interchanged, so I/O statements inhibit interchange and therefore data localization. For example, consider the following Fortran code:

```
DO I = 1, 4
  DO J = 1, 4
    READ *, IA(I,J)
  ENDDO
ENDDO
```

Given a data stream consisting of alternating zeros and ones (0,1,0,1,0,1...), the contents for A(I,J) for both the original loop and the interchanged loop are shown in Figure 15.

Optimizations

Original loop

I \ J	1	2	3	4
1	0	1	0	1
2	0	1	0	1
3	0	1	0	1
4	0	1	0	1

Interchanged loop

I \ J	1	2	3	4
1	1	1	1	1
2	0	0	0	0
3	1	1	1	1
4	0	0	0	0

Figure 15 Values read into array A

C loops exhibit the same limitations. A C example that produces the data patterns shown in Figure 15 follows:

```
for(i=1;i<5;i++)
  for(j=1;j<5;j++)
    scanf("%d",&ia[i][j]);
```

Preventing data localization

The SCALAR directive allows you to prevent all loop-reordering transformations on the immediately following loop. In Fortran, it has the following form:

```
C$DIR SCALAR
```

In C it has the following form:

```
#pragma _CNX scalar
```

-O3 Level optimizations

The primary goal of -O3 optimizations is *parallelization*. Parallelization divides a program into threads. A *thread* is a single flow of control within a process. It can be a unique flow of control that performs a specific function, or one of several instances of a flow of control, each of which is operating on a unique data set.

The Convex SPP Series compilers find parallelism at the loop level and generate parallel code that will automatically run on as many processors as are available at runtime. Normally, these are all the processors of the subcomplex on which your program is running. You can specify a smaller number of processors via the `mpa` utility. Refer to the mpa(1) man page for more information. You can also specify a smaller number of processors using the `loop_parallel(max_threads=`*m*`)` or `prefer_parallel(max_threads=`*m*`)` compiler directives and pragmas, as described in the chapters "Basic shared-memory programming" and "Advanced shared-memory programming."

Automatic parallelization is useful for programs containing loops. You can use compiler directives or pragmas to improve on the automatic optimizations and to also assist the compiler in locating additional opportunities for parallelization.

If you are writing your program entirely under the message-passing paradigm, you must explicitly handle parallelism yourself as discussed in the appropriate manual: *Convex MPICH User's Guide for Exemplar Systems* or *PVM/GSM User's Guide for Exemplar Systems*. Such pure message-passing programs should be compiled at optimization level -O2 or lower.

Basic operation

Parallelism can exist at the loop level, region level, and task level. Convex SPP Series compilers automatically exploit loop-level parallelism. You can easily identify task-level parallelism using the `begin_tasks`, `next_task` and `end_tasks` directives and pragmas, as discussed in the section "Task parallelization" on page 137. You can also identify parallel regions using the `parallel` and `end_parallel` directives and pragmas, as discussed in the section "Region parallelization" on page 142. These directives and pragmas allow the compiler to run identified sections of code in parallel.

Loop-level parallelism involves dividing a loop up into several smaller iteration spaces and parceling these out to be run simultaneously on the available processors.

Note

Only loops with iteration counts determinable prior to loop invocation at runtime are candidates for parallelization. Loops with iteration counts that depend on values or conditions calculated within the loop cannot be parallelized by any means.

Consider the following Fortran code:

```
PROGRAM PARAXPL
.
.
.
DO I = 1, 1024
  A(I) = B(I) + C(I)
   .
   .
   .
ENDDO
```

Assuming the `I` loop does not contain any parallelization-inhibiting code, this program can be parallelized to run on 8 processors by running 128 iterations per processor (1024 iterations divided by 8 processors = 128 iterations each). One processor would run the loop for `I` = 1 to 128; the next would run `I` = 129 to 256, and so on. The loop could similarly be parallelized to run on any number of processors, with each one taking its appropriate share of iterations. At a certain point, however, adding more processors will not improve performance. The compiler generates code that will run on as many processors as are available, but the dynamic selection optimization (described in the section "Dynamic selection" on page 115) ensures that parallel code is generated only if it is profitable to do so. If the number of available processors does not evenly divide the number of iterations, some processors will perform fewer iterations than others.

On an Exemplar system, shared-memory programs run as a collection of threads on multiple processors. When a program starts, a separate execution thread is created on each of the processors of the subcomplex on which the program is running. SPP-UX identifies each of these threads by a unique *kernel* thread ID. All but one of these threads is then idle; the nonidle thread is known as thread 0, and this thread runs all of the serial code in the program. *Spawn* thread IDs are assigned only to nonidle threads when they are spawned—that is, when thread 0 encounters parallelism and "wakes up" other idle threads to

execute the parallel code. Spawn thread IDs are consecutive. They range from 0 to *nt*-1, where *nt* is the number of threads spawned as a result of the spawn operation; this operation defines the current *spawn context*. The spawn context is the loop or task list that initiates the spawning of the threads. Spawn thread IDs are valid only within a given spawn context.

Therefore, the idle threads are not assigned spawn thread IDs at the time of their creation. When thread 0 encounters a parallel loop or task, it spawns the other threads, signaling them to begin execution. The threads then become active, acquire spawn thread IDs, run until their portion of the parallel code is finished, and go idle once again, as shown in Figure 16. Machine loading does not affect the number of threads spawned, but it may affect the order in which the threads in a given spawn context complete. SPP-UX ensures that, regardless of the machine load, all child threads of a given process execute to the completion of their spawn context before that process's thread 0 continues.

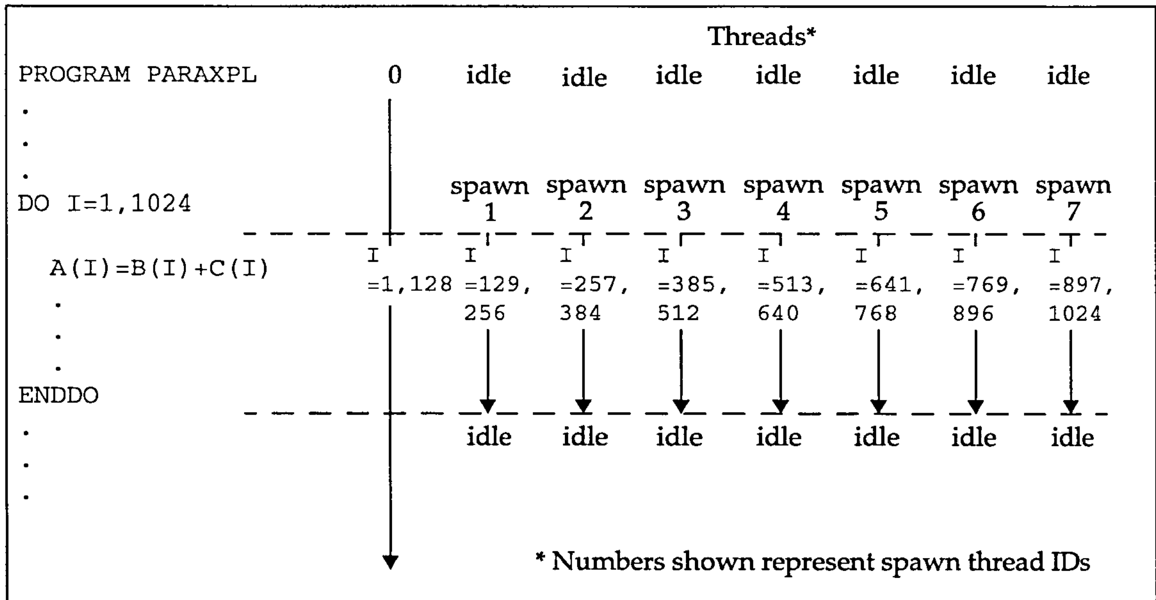


Figure 16 Thread activity: one-dimensional parallelism

Note

This example is greatly simplified for illustrative purposes. Various loop transformations can affect the manner in which a loop is parallelized.

To actually implement this, the compiler transforms the loop in a manner similar to strip mining. However, unlike the strip mining described in the section "-O2 Level optimizations" on page 63, the outer loop is conceptual; because the strips will be executing

on different CPUs, there is no processor to run an outer loop like the one created in traditional strip mining.

Instead, the loop is transformed such that the starting and stopping iteration values are variables that are determined at runtime based on how many threads are available and which thread is running the strip in question.

Consider our previous Fortran example written for an unspecified number of iterations:

```
DO I = 1, N
  A(I) = B(I) + C(I)
ENDDO
```

The code shown in Figure 17 is a conceptual representation of the transformation the compiler performs on this example when it is compiled for parallelization, assuming that `N >= NumThreads`. For `N < NumThreads`, the compiler uses `N` threads, assuming there is enough work in the loop to justify the overhead of parallelizing it. If `NumThreads` is not an integral divisor of `N`, some threads will do fewer iterations than others.

```
For each available thread do:
  DO I = ThrdID*(N/NumThreads)+1,ThrdID*(N/NumThreads)+N/NumThreads
    A(I) = B(I) + C(I)
  ENDDO
ENDDO
```

Figure 17 Conceptual strip mine for parallelization

`NumThreads` is the number of available threads. `ThrdID` is the ID number of the thread this particular loop will run on, which is between 0 and `NumThreads-1`. A unique `ThrdID` is assigned to each thread, and the `ThrdIDs` are consecutive. So, for `NumThreads = 8`, as in Figure 16, 8 loops would be spawned, with `ThrdIDs` = 0 through 7. These 8 loops are illustrated in Figure 18.

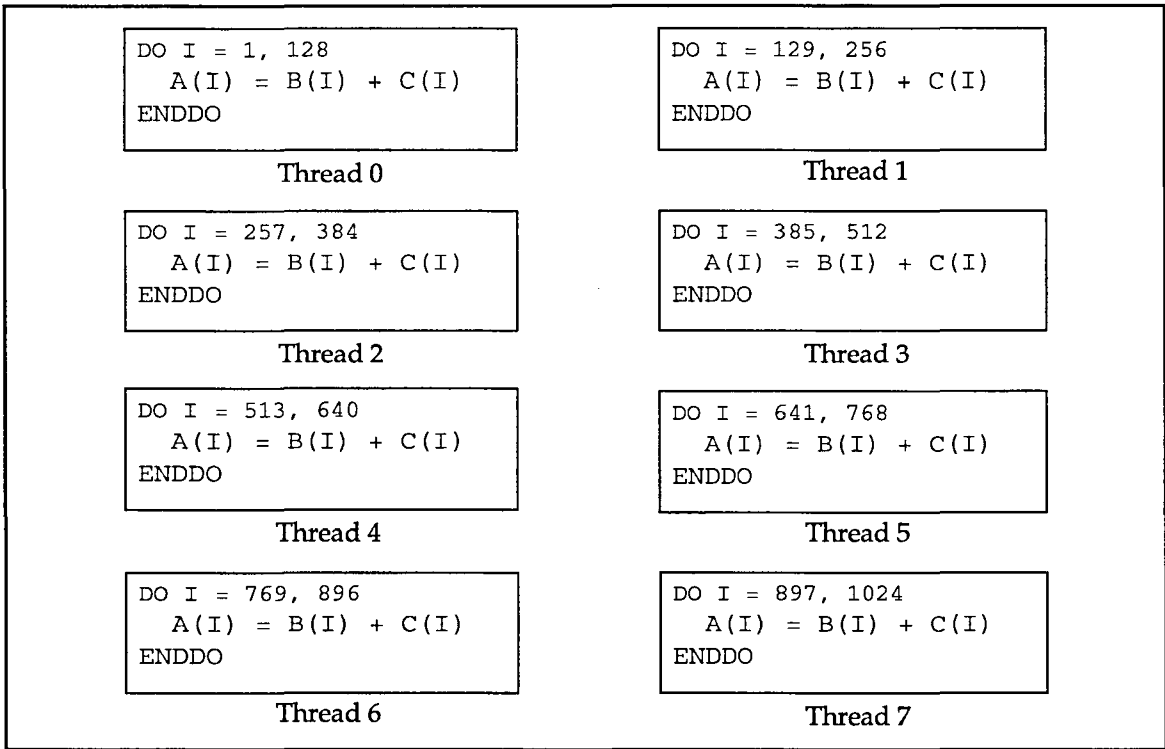


Figure 18 Parallelized loop

Note

The strip-based parallelism described here is the default. Stride-based parallelism is possible through use of the `PREFER_PARALLEL` and `LOOP_PARALLEL` compiler directives and pragmas, which are described in Chapter 4, "Basic shared-memory programming."

In these examples, the data being manipulated within the loop is disjoint; that is, no two threads attempt to write the same data item. If two parallel threads attempt to update the same storage location, their actions must be synchronized. This is discussed further in Chapter 4 "Basic shared-memory programming."

Idle thread states

Idle threads can be suspended or spin-waiting. Suspended threads release control of the processor; spin-waiting threads repeatedly check an encached global semaphore that indicates whether or not they have code to execute. Obviously, this prevents any other process from gaining control of the CPU and can severely degrade multiprocess performance. On the other hand, waking a suspended thread takes substantially longer than activating a spin-waiting thread. Therefore, by default, idle threads spin-wait briefly after creation or a join, then suspend themselves if no work is received. This spin-wait time can be changed by using the `cps_wait_attr()` function, which is described in the "Compiler Parallel Support Library" appendix.

When threads are suspended, SPP-UX may schedule threads of another process on their CPUs in order to balance the machine load. However, threads have an *affinity* for their original CPUs; SPP-UX attempts to schedule the threads that have been swapped out or suspended to their original CPUs in order to exploit the presence of any data encached during the thread's last timeslice. This affinity is only realized if the original CPU is available; otherwise, the thread is assigned to the first CPU to become available within the hypernode on which it was spawned.

Node-parallelism vs. thread-parallelism

SPP Series compilers support two dimensions of parallelism: thread-parallelism and, on multihypernode systems, node-parallelism. The compilers support node-parallelism through the specification of the `nodes` attribute in the `loop_parallel`, `prefer_parallel`, `parallel`, and `begin_tasks` directives and pragmas. Unlike thread-parallelism, node-parallelism is not performed automatically; you must use directives or pragmas. By supporting two dimensions of parallelism, the compilers allow you to exploit parallelism within parallelism when it occurs in your program.

Using the `prefer_parallel (nodes)` directive or pragma, the compilers perform parallelization analysis on the loop. For example, the compilers will automatically parallelize two loops in a loop nest if it is safe to do so; the outermost loop (after interchange) will be node-parallelized, and the innermost loop will be thread-parallelized to run on the processors of each hypernode the other loop is running on. However, automatic parallelization analysis is disabled when you use the `loop_parallel(nodes)`, `parallel(nodes)`, and `begin_tasks(nodes)/next_task/end_tasks` directives or pragmas, so you must explicitly indicate thread-parallel loops, tasks, or regions contained within these node-parallel constructs by using the appropriate directive or pragma.

If two-dimensional parallelism is not present in a particular loop nest, or if exploiting it would be inefficient, you can also run single-dimensional (thread) parallelism across two or more hypernodes of the subcomplex. In such cases, you can use the `prefer_parallel`, `loop_parallel`, or `parallel` compiler directives or pragmas with the `threads` attribute to parallelize across all available threads on multiple hypernodes, or to limit parallelization to a subset of available threads or hypernodes. Refer to Chapter 4, "Basic shared-memory programming," for more information on these and other parallelization directives and pragmas.

Parallel threads are started at program startup as described in the section "Basic operation" on page 105 without regard to the presence or absence of two-dimensional parallelism. The level of parallelism affects only the physical location of the threads that are activated when the parallel construct is encountered; it does not affect the total number of threads available to the program.

When node-parallelism is encountered, assuming the program is running on a multihypernode subcomplex, a single thread on each hypernode is activated. These threads are given spawn thread IDs ranging from 0 (the thread that encountered the node-parallelism) to one less than the number of hypernodes. If thread-parallelism is then encountered within the node-parallelism, a new spawn context is established, and the available threads on each hypernode (assuming the parallel construct does not limit the number of spawned threads) are activated. In this new spawn context, the threads on each hypernode are given spawn thread IDs ranging from 0 to one less than the number of threads on the hypernode. This means that spawn thread IDs are duplicated on each hypernode in the context of thread-parallel code within node-parallel code (on a given hypernode, however, they are unique).

Consider the following Fortran example:

```
      PROGRAM 2DXPL
      .
      .
      .
C$DIR PREFER_PARALLEL(NODES)
      DO J = 1, 1024
C$DIR   PREFER_PARALLEL(THREADS)
        DO I = 1, 1024
          A(I,J) = B(I,J) + C(I,J)
          .
          .
          .
        ENDDO
      ENDDO
```

Here, assuming the loop nest does not contain any parallelization inhibitors, the compiler parallelizes the `J` loop across hypernodes, and the `I` loop across threads within those hypernodes. Assuming this program is running on a subcomplex consisting of two four-processor hypernodes, the parallelization is illustrated in Figure 19.

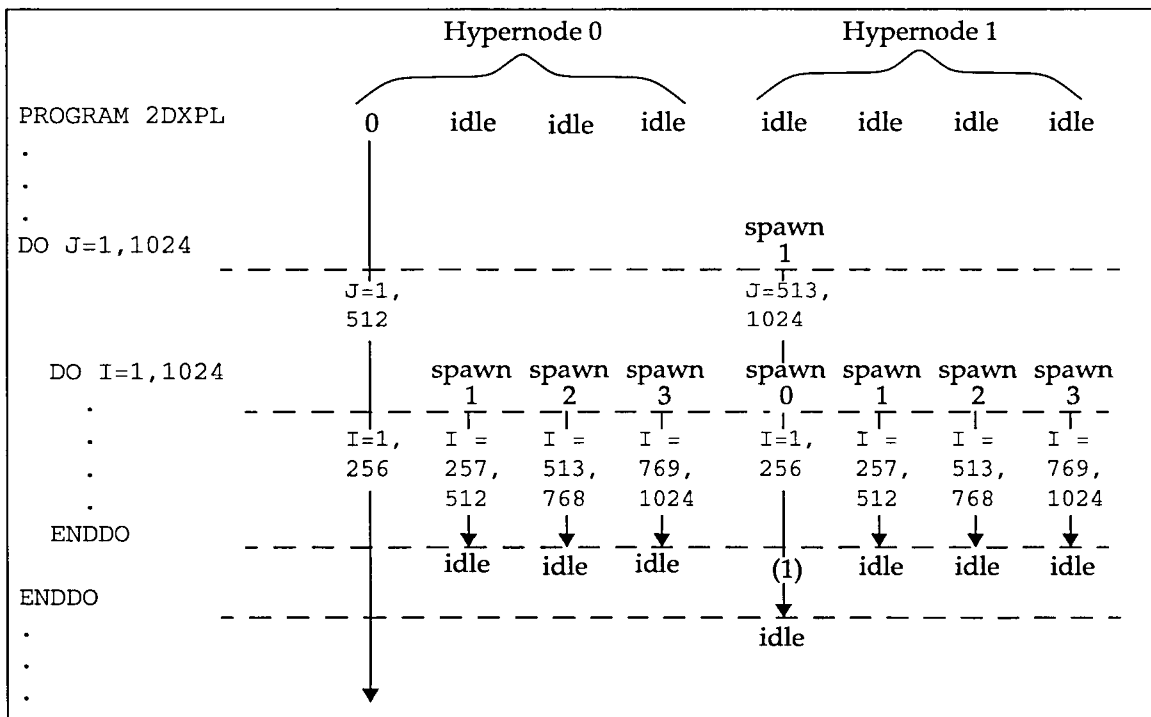


Figure 19 Thread activity: two-dimensional parallelism

Note

This example is greatly simplified for illustrative purposes. Various loop transformations can affect the manner in which a loop is parallelized.

As shown in Figure 19, the node-parallel `J` loop spawns two threads, one on each hypernode. Thread ID 0 was already running the serial code in the program; ID 1 is activated when the node-parallel loop begins execution.

The `I` loop then spawns thread-parallelism on each hypernode. The original thread 0 maintains its spawn thread ID, and IDs 1-3 are also spawned on hypernode 0. The thread that is spawn thread 1 in the context of the `J` loop becomes spawn thread 0 in the context of the `I` loop, and IDs 1-3 are also spawned on hypernode 1 in the context of the `I` loop.

Note that when the `I` loop terminates, the spawn context returns to that of the `J` loop, and thread 0 on hypernode 1 becomes thread 1 again, as it had been before the `I` loop began.

Node-parallelism can be disabled by specifying the `-nonodepar` command-line option. This prevents the compiler from implementing node-parallelism, but allows the implementation of both automatic and directive-specified thread-parallelism.

Fortran 90 constructs

Fortran 90 array expressions are translated into loops by the compiler, and parallelism in these loops will be automatically exploited. For example, the following Fortran 90 statement:

```
X(1:M:2, 1:N) = Y(2:M+1:2,2:N+1)
```

is translated by the compiler into a loop similar to the following:

```
DO I = 1, N
  DO J = 1, M, 2
    X(J,I) = Y(J+1,I+1)
  ENDDO
ENDDO
```

which can then be automatically parallelized.

Masked array assignments are similarly parallelized. Consider the following `WHERE` statement:

```
REAL DATA(1000), LIMIT
LOGICAL NORMAL(1000)
.
.
.
WHERE(DATA .LE. LIMIT) NORMAL = .TRUE.
```

The compiler translates the `WHERE` statement into a loop similar to the following:

```
DO I = 1, 1000
  IF(DATA(I) .LE. LIMIT) NORMAL(I) = .TRUE.
ENDDO
```

which can then be automatically parallelized.

Parallel optimizations

Simple loops can be parallelized without the need for extensive -O2 transformations, as shown in the section "Basic operation" on page 105. However, most loop transformations, if they are applicable to the loop in question, can aid parallelization in some way. For instance, loop interchange orders loops such that the innermost loop best exploits the processor data cache, and the outermost loop is the most efficient loop to parallelize. Loop blocking similarly aids parallelization by maximizing cache data reuse on each of the processors that the loop runs on, and by ensuring that each processor is working on nonoverlapping array data.

All optimizations performed at lower optimization levels are performed at optimization level -O3.

Dynamic selection

The compiler has no way of determining how many processors will be available to run compiled code; therefore, it must generate both serial and parallel code for loops that can be parallelized. Replicating the loop in this manner is called *cloning*, and the resulting versions of the loop are called *clones*.

It is not always profitable, however, to run the parallel clone when multiple processors are available. A great deal of overhead is involved in executing parallel code. This overhead includes the time it takes to spawn parallel threads, to privatize any variables used in the loop that must be privatized, and to join the parallel threads when they complete their work.

Convex SPP Series compilers use a powerful form of dynamic selection known as *workload-based dynamic selection*. When a loop's iteration count is available at compile time, workload-based dynamic selection determines the profitability of parallelizing the loop and only writes a parallel version to the executable if it is profitable to do so. Omitting the parallel version from the executable when it will never be needed enhances performance further by eliminating the runtime decision as to which version to use.

The power of dynamic selection becomes more apparent when the loop's iteration count is unknown at compile time. In this case, the compiler generates code that, at runtime, compares the amount of work performed in the loop nest (given the actual iteration counts) to the parallelization overhead for the available number of processors and runs the parallel version of the loop only if it is profitable to do so.

Workload-based dynamic selection is enabled by default at optimization level `-O3` on SPP Series machines. The `-nds` compiler option can be used to disable dynamic selection. When dynamic selection is disabled, the compiler assumes that it is profitable to parallelize all parallelizable loops, and generates both serial and parallel clones for them. In this case the parallel version is run if there are multiple processors at runtime regardless of the profitability of doing so.

You can enable workload-based dynamic selection for selected loops by using the `dynsel` compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR DYNSEL[(THREAD_TRIP_COUNT = n | NODE_TRIP_COUNT = m)]
```

The C pragma has the following form:

```
#pragma _CNX dynsel[(thread_trip_count = n | node_trip_count = m)]
```

Where `thread_trip_count` and `node_trip_count` are optional attributes used to specify threshold iteration counts. If `thread_trip_count = ` *n* is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the thread-parallel version is run. If `node_trip_count = ` *m* is specified, the serial version of the loop is run if the iteration count is less than *m*; otherwise, the node-parallel version is run. *n* and *m* must be compile-time constants.

The thread_trip_count attribute cannot be used on loops that also specify the loop_parallel(nodes) directive or pragma. Similarly, the node_trip_count attribute cannot be used on loops that also specify the loop_parallel(threads) directive or pragma. These combinations are contradictory. These directives can be used to specify dynamic selection for specific loops in programs compiled using the -nds option, or to provide trip count information for specific loops in programs compiled with dynamic selection enabled. You can disable dynamic selection for selected loops by using the no_dynsel compiler directive or pragma. In Fortran, this directive has the following form:

```
C$DIR NO_DYNSEL
```

The C pragma has the following form:

```
#pragma _CNX no_dynsel
```

This directive or pragma can be used to disable dynamic selection on specific loops in programs compiled with dynamic selection enabled.

As with all optimizations that replicate loops, the number of new loops created when the compiler performs dynamic selection is limited by default to ensure reasonable compile times. If this limit is reached during compilation, the compiler issues an advisory and workload-based dynamic selection is disabled for any following loops. To increase the replication limit (and possibly increase your compile time and compile time memory usage significantly), specify the -mrl compiler option.

Inhibitors of parallelization

Most constructs that inhibit data localization also inhibit parallelization. Specifically, these are:

- Loop-carried dependences
- Aliased scalar or array variables
- Multiple loop entries or exits
- Procedure calls
- I/O statements

Most of these items inhibit parallelization for the same reasons they inhibit localization. An exception to this is that more categories of loop-carried dependences can inhibit parallelization than data localization, as described in the following sections.

Loop-carried dependences

The specific loop-carried dependences that inhibit data localization represent a very small portion of all loop-carried dependences. A much broader set of LCDs, including those that inhibit data localization, can inhibit parallelization.

LCDs fall into three categories:

- *forward* LCDs
- *backward* LCDs
- *output* LCDs

The LCD that inhibits localization is a combination of these. All of these LCDs inhibit parallelization.

Forward LCDs

A forward LCD exists when one iteration references a variable whose value is assigned on a later iteration. The Fortran loop below contains a forward LCD on the array A.

```
DO I = 1, N - 1
  A(I) = A(I + 1) + B(I)
ENDDO
```

In this example, the first iteration assigns a value to A(1) and references A(2). The second iteration assigns a value to A(2) and references A(3). The reference to A(I) depends on the fact that the I+1th iteration, which assigns a new value to A(I), has not yet executed. Forward LCDs inhibit parallelization because if the loop is broken up to run on several processors, when I reaches its terminal value on one processor, A(I+1) will usually have already been computed by another processor (it is, in fact, the first value computed by another processor). Because the calculation

depends on A(I+1) being uncomputed, this would produce wrong answers.

An analogous C loop follows:

```
for(i=0;i<n-1;i++)
  a[i] = a[i+1] + b[i];
```

Backward LCDs

A backward LCD exists when one iteration references a variable whose value was assigned on an earlier iteration. The Fortran loop below contains a backward LCD on the array A.

```
DO I = 2, N
  A(I) = A(I-1) + B(I)
ENDDO
```

Here, each iteration assigns a value to A based on the value assigned to A in the previous iteration. If A(I-1) has not been computed before A(I) is assigned, wrong answers will result. Backward LCDs inhibit parallelism because if the loop is broken up to run on several processors, A(I-1) will not have been computed for the first iteration of the loop on every processor except the processor running the chunk of the loop containing I = 1.

An analogous C loop follows:

```
for(i=1;i<n;i++)
  a[i] = a[i-1] + b[i];
```

Output LCDs

An output LCD exists when the compiler cannot determine whether an array subscript contains the same values between loop iterations. The Fortran loop below contains a potential output LCD on the array A:

```
DO I = 1, N
  A(J(I)) = B(I)
ENDDO
```

Here, if any referenced elements of J contain the same value, the same element of A might be assigned several different elements of B. In this case, as this loop is written, any A elements that are assigned more than once should contain the final assignment at the end of the loop. If the loop is run in parallel, however, this cannot be guaranteed.

An analogous C loop follows:

```
for(i=0;i<n;i++)
  a[j[i]] = b[i];
```

Apparent LCDs

As at optimization level -O3, the compiler will not parallelize loops containing apparent LCDs rather than risk wrong answers by doing so.

If you are sure that a loop with an apparent LCD is safe to parallelize, you can indicate this to the compiler using the NO_LOOP_DEPENDENCE directive or pragma, which is explained in the section "-O2 Level optimizations" on page 63.

The following Fortran example illustrates a NO_LOOP_DEPENDENCE directive being used on the output LCD example presented previously:

```
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, N
        A(J(I)) = B(I)
      ENDDO
```

This effectively tells the compiler that no two elements of J are identical, so there is no output LCD and the loop is safe to parallelize. If any of the J values are identical, wrong answers could result.

Use of the no_loop_dependence pragma is illustrated in the following C example:

```
#pragma _CNX no_loop_dependence(a)
for(i=0;i<n;i++)
  a[j[i]] = b[i];
```

Reductions

In many cases, the compiler can recognize and parallelize loops containing a special class of dependence known as a reduction. In general, a reduction has the form:

X = X *operator* Y

where X is a variable not assigned or used elsewhere in the loop, Y is a loop constant expression not involving X, and *operator* is +, -, *, .AND., .OR., .EQV., or .NEQV.

The compiler also recognizes reductions of the form:

X = *function*(X,Y)

where X is a variable not assigned or referenced elsewhere in the loop, Y is a loop constant expression not involving X, and *function* is the intrinsic MAX function or intrinsic MIN function.

Reductions commonly appear in the form of sum operations, as shown in the following Fortran example:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  .
  .
  .
  ASUM = ASUM + A(I)
ENDDO
```

Assuming this loop does not contain any parallelization-inhibiting code, the compiler would automatically parallelize it. The code generated to accomplish this creates temporary, thread-private copies of ASUM for each thread that will be running the loop. When each parallel thread completes its portion of the loop, it updates the global ASUM variable in a section of code that prevents other threads from accessing ASUM simultaneously.

The optimization report would indicate that ASUM was privatized and its value saved for use after the loop.

An analogous C example follows:

```
for(i=0;i<n;i++) {
  a[i] = b[i] + c[i];
  .
  .
  .
  asum = asum + a[i];
}
```

Optimizations

Preventing parallelization

Parallelization can be disabled on a loop basis by using the NO_PARALLEL directive or pragma. The Fortran directive has the following form:

```
C$DIR NO_PARALLEL
```

The C pragma has the following form:

```
#pragma _CNX no_parallel
```

You can use these directives to prevent parallelization of the loop that immediately follows them. Only parallelization is inhibited; all other loop optimizations will still be applied. The following Fortran example illustrates the use of this directive:

```
      DO I = 1, 1000
C$DIR   NO_PARALLEL
        DO J = 1, 1000
          A(I,J) = B(I,J)
        ENDDO
      ENDDO
```

In this example, parallelization of the J loop is prevented. The I loop can still be parallelized.

An analogous C example follows:

```
for(i=0;i<1000;i++)
#pragma _CNX no_parallel
  for(j=0;j<1000;j++)
    a[i][j] = b[i][j];
```

The -noautopar compiler option is available to disable automatic parallelization but will allow parallelization of directive- or pragma-specified loops. Refer to Chapter 4, "Basic shared-memory programming," for more information on -noautopar.

Other parallelization directives and pragmas

Several directives and pragmas are available to allow you to manually control certain aspects of loop parallelization, and to parallelize tasks outside of loops. These directives are:

`prefer_parallel`

Requests parallelization of the immediately following loop; accepts attributes for node- and thread-parallelism, strip-length adjustment, maximum number of threads, and ordered execution. The compiler handles data privatization and does not parallelize the loop if it is not safe to do so.

`loop_parallel`

Forces parallelization of the immediately following loop. Accepts the same attributes as `PREFER_PARALLEL`, but requires you to manually privatize loop data and synchronize data dependences.

`begin_tasks`, `next_task` and `end_tasks`

Allow you to parallelize consecutive blocks of code outside of loops. Accepts attributes for node- and thread-parallelism, ordered execution, and maximum number of threads.

`parallel`, `end_parallel`

Allow you to parallelize a single code region to run on multiple threads. Unlike the tasking directives, which run discrete sections of code in parallel, `parallel/end_parallel` run multiple copies of a single section. Accepts attributes for node- and thread-parallelism, and maximum number of threads.

`critical_section`, `end_critical_section`

Allow you to isolate nonordered manipulations of a shared variable within the loop. Only one parallel thread can execute the code contained in the critical section at a time, eliminating possible contention.

`ordered_section`, `end_ordered_section`

Allow you to isolate dependences within a loop so that code contained within the ordered section executes in iteration order. Only useful when used with the `loop_parallel(ordered)` or `prefer_parallel(ordered)` directives or pragmas.

These directives and pragmas are discussed in detail in Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming."

Basic shared-memory programming

4

This chapter discusses programming techniques that allow you to increase code efficiency with minimal effort.

Simple manual loop, task, and region parallelization

The Convex SPP Series compilers automatically exploit strip-based loop parallelism in loops that are clearly dependence-free, as described in Chapter 3, "Compiler optimizations." The `prefer_parallel`, `loop_parallel`, and `parallel` directives and pragmas allow you to increase parallelization opportunities and to manually control many aspects of parallelization.

The compiler cannot automatically locate task parallelism, but the tasking directives mentioned in Chapter 3, "Compiler optimizations," (and discussed here) allow you to specify consecutive blocks of code that can be run in parallel. Similarly, the `parallel` and `end_parallel` directives and pragmas allow you to specify a code region that can be run in its entirety on several processors.

The subsections that follow discuss specifying simple, unordered loop, task, and region parallelism using the `prefer_parallel`, `loop_parallel`, `begin_tasks/next_task/ end_tasks`, and `parallel/end_parallel` directives and pragmas. Critical sections that do not rely on ordered execution are also covered here. Any necessary variable privatization is provided by the `loop_private`, `task_private` and `parallel_private` directives and pragmas, which are described in detail in the "Loop-specific, task-specific, and region-specific data privatization" section of this chapter.

For a detailed discussion of ordered parallelism, parallel synchronization, and the effective use of memory classes, refer to

Chapter 5, "Memory classes," and Chapter 6, "Advanced shared-memory programming."

Loop parallelization

This section discusses simple uses of the `prefer_parallel` and `loop_parallel` directives and pragmas, which, when specified, apply to the immediately following loop. The data privatization necessary when using `loop_parallel` is implemented in this chapter's examples using the `loop_private` directive discussed on page 150. Manual data privatization using memory classes is discussed in Chapter 5, "Memory classes," and Chapter 6, "Advanced shared-memory programming."

Note

These directives should only be used on Fortran `DO` and C `for` loops that have iteration counts determinable prior to loop invocation at runtime.

`loop_parallel` and `prefer_parallel` generally take the same attributes. In Fortran, these directives have the following form:

`C$DIR PREFER_PARALLEL[(`*attribute-list*`)]`

and

`C$DIR LOOP_PARALLEL[(`*attribute-list*`)]`

In C, they have the following form:

`#pragma _CNX prefer_parallel[(`*attribute-list*`)]`

and

`#pragma _CNX loop_parallel(ivar = `*indvar*`[, `*attribute-list*`])`

where the optional *attribute-list* can contain one of the following case-insensitive attributes. The `dist` attribute, which works only with the `loop_parallel` directive and pragma, can be used with any combination that does not include the `nodes` attribute or the `threads` attribute. See the section "`dist` attribute" on page 132 for information.

Note **The values of *n* and *m* must be compile-time constants for all of the below attributes in which they appear.**

`threads`

Causes thread-parallelism. This is the default.

`nodes`

Causes thread-based node-parallelism.

`chunk_size =` *n*

Divides the loop into chunks of *n* or fewer iterations, and distributes the chunks round-robin to the processors. *n* must be a constant integer that has a value at compile time.

`max_threads =` *m*

Allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer that has a value at compile time.

`ordered`

Causes ordered invocation of each loop iteration; provides no automatic synchronization. Designed for use with the `loop_parallel` directive and pragma, on loops containing ordered sections.

`ordered, nodes`

Causes ordered invocation of each iteration across hypernodes.

`ordered, threads`

Causes ordered invocation of each iteration across threads.

`nodes, chunk_size =` *n*

Node-parallelism by chunks.

`threads, chunk_size =` *n*

Thread-parallelism by chunks.

`chunk_size =` *n*`, max_threads =` *m*

Chunk parallelism on no more than *m* threads.

`ordered, max_threads =` *m*

Ordered parallelism on no more than *m* threads.

`nodes, max_threads =` *m*

Node-parallelize on no more than *m* nodes; this starts one thread per node on no more than *m* hypernodes.

`threads, max_threads =` *m*

Thread-parallelism on no more than *m* threads.

`ordered, nodes, max_threads =` *m*

Ordered node-parallelism on no more than *m* hypernodes.

`ordered, threads, max_threads =` *m*

Ordered thread-parallelism on no more than *m* threads.

`nodes, chunk_size =` *n*`, max_threads =` *m*

Node-parallelize by chunks of size *n* on no more than *m* hypernodes.

`threads, chunk_size =` *n*`, max_threads =` *m*

Thread-parallelize by chunks of size *n* on no more than *m* threads.

`dist`

Use only with `loop_parallel` inside a `parallel/end_parallel` region. Causes the compiler to distribute the iterations of a loop across active threads instead of spawning new threads. The level of parallelism is determined by the attribute used in the `parallel` directive or pragma. (See "Region parallelization" on page 142.) The `dist` attribute can be used with any combination of attributes that does not include the `nodes` attribute or the `threads` attribute.

`ivar =` *indvar*

Use only with `loop_parallel`. Specifies the primary loop induction variable; optional in Fortran, but required in C.

Combining the attributes

The allowed combinations of attributes are those combinations listed in the preceding section. The `dist` attribute can be added to any combination from the preceding section that does not include the `nodes` attribute or the `threads` attribute.

In such combinations the attributes can be listed in any order. The `loop_parallel` C pragma requires the `ivar =` *indvar* attribute, which specifies the primary loop induction variable. If this is not present, the compiler will issue a warning and the pragma will be ignored. `ivar` should specify only the primary induction variable; any other loop induction variables should be a function of this variable and should be declared `loop_private`. In Fortran, `ivar` is optional for `DO` loops; if not provided, the compiler will pick the primary induction variable for the loop. `ivar` is required for `DO WHILE` and hand-rolled loops in Fortran. `prefer_parallel` does not require `ivar`, and the compiler will issue an error if it encounters this combination.

Using the attributes

The attributes associated with the `prefer_parallel` and `loop_parallel` directives and pragmas are explained in the following sections.

`threads` attribute

The optional `threads` attribute causes parallelization across threads; this is the default for `loop_parallel` and `prefer_parallel`. If the `threads` attribute appears in a parallelization directive on the outermost loop in a nest, the loop will go parallel on all the threads available to the process. If the `threads` attribute appears in a parallelization directive nested within a node-parallel construct, the specified loop will go thread-parallel on the processors of each parallel hypernode.

`nodes` attribute

The optional `nodes` attribute causes parallelization across hypernodes. In this case, a single thread on each available hypernode will execute a portion of the specified loop. A `prefer_parallel(nodes)` directive or pragma allows the compiler to find and exploit any thread-parallelism present within the node-parallel loop. A `loop_parallel(nodes)` directive or pragma disables parallelization analysis for any loops inside the node-parallel loop; in this case if you want to exploit nested parallelism, you must explicitly indicate the thread parallel construct contained within the node-parallel loop. A node-parallel construct cannot exist inside a thread-parallel construct.

`chunk_size` = *n* attribute

The optional `chunk_size` = *n* attribute specifies a number of iterations by which to strip mine the loop for parallelization. If this attribute is present alone or with the `threads` attribute, *n* or fewer loop iterations are distributed round-robin to each available thread. If `chunk_size` = *n* is combined with the `nodes` attribute, the chunks are distributed to one thread per available hypernode. If the number of threads does not evenly divide the number of iterations, some threads will perform one less chunk than others. *n* must be an integer constant or constant expression determinable at compile time.

This stride-based parallelism differs from the default strip-based parallelism described in Chapter 3, "Compiler optimizations," that divides the loop's iterations into a number of contiguous chunks equal to the number of available threads, and each thread computes one chunk. The `chunk_size = n` attribute allows each thread to do several noncontiguous chunks. Specifying `chunk_size =` (*number of iterations/number of threads*) is equivalent to default strip mining for parallelization.

Consider the following Fortran example, which uses the `PREFER_PARALLEL` directive, but applies to `LOOP_PARALLEL` as well:

```
C$DIR PREFER_PARALLEL(CHUNK_SIZE = 4)
      DO I = 1, 100
        A(I) = B(I) + C(I)
      ENDDO
```

In this example, the loop is parallelized by parcelling out chunks of 4 iterations to each available thread. Figure 20 uses Fortran 90 array syntax to illustrate the iterations performed by each thread, assuming 8 available threads.

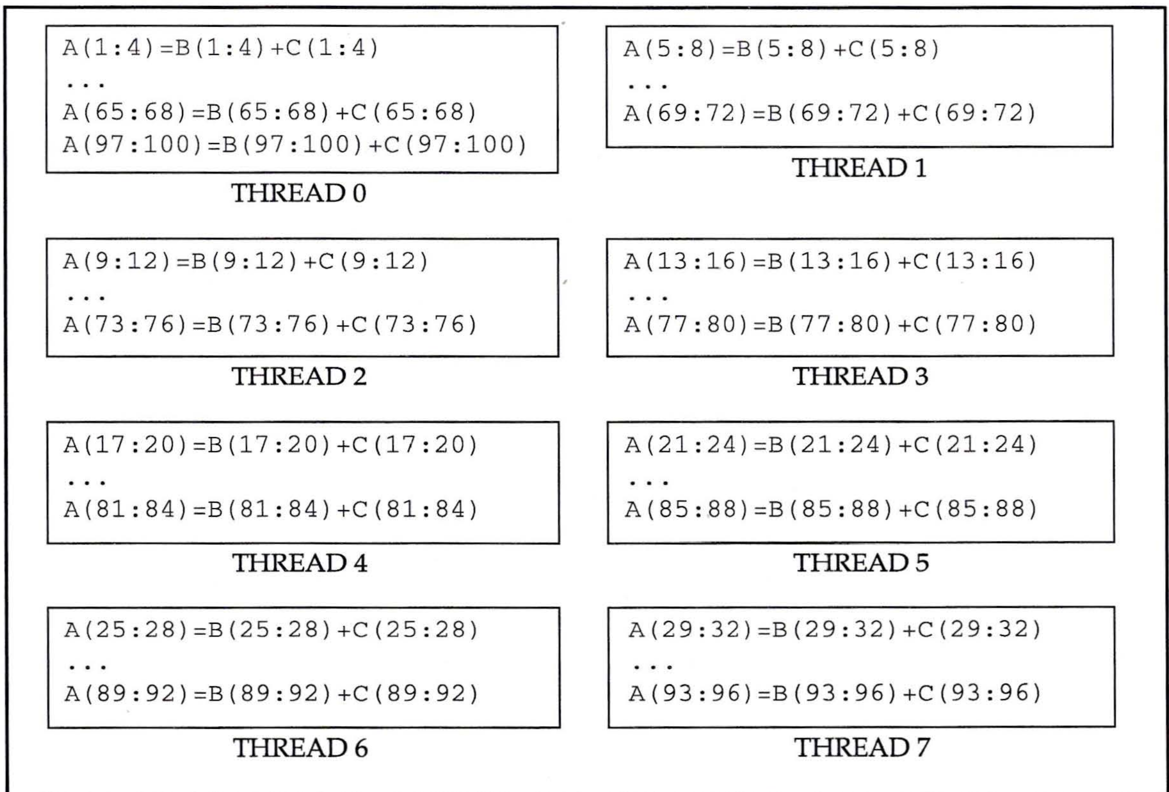


Figure 20 Stride-parallelized loop

Figure 20 shows that the 100 iterations of I are parcelled out in chunks of 4 iterations to each of the 8 available threads; after the chunks are distributed evenly to all threads, there is one chunk left over (iterations 97:100), which executes on thread 0.

An analogous C example follows:

```
#pragma _CNX prefer_parallel(chunk_size = 4)
for(i=0;i<100;i++)
  a[i] = b[i] + c[i];
```

The chunk_size = *n* attribute is most useful on loops in which the amount of work increases or decreases as a function of the iteration count. (These loops are also known as *triangular loops*.) The following Fortran example shows such a loop. Again, PREFER_PARALLEL is used here, but the concept applies to LOOP_PARALLEL also.

```
C$DIR PREFER_PARALLEL(CHUNK_SIZE = 4)
      DO J = 1,N
        DO I = J, N
          A(I,J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the work of the I loop decreases as J increases. By specifying a chunk_size for J, we more evenly balance the load across the threads executing the loop. If this loop was strip mined in the traditional manner, the amount of work contained in the strips would decrease with each successive strip; the threads performing early iterations of J would do substantially more work than those performing later iterations.

An analogous C example follows:

```
#pragma _CNX prefer_parallel(chunk_size = 4)
for(j=0;j<n;j++)
  for(i=j;i<n;i++) {
    a[i][j] = ...
    .
    .
    .
  }
```

For more information and examples on using the chunk_size = *n* attribute, see the sections "Triangular loops" on page 290 and "Distributing iterations on cache line boundaries" on page 271.

Basic shared memory

`max_threads` = *m* attribute

The `max_threads = m` attribute restricts execution of the specified loop to no more than *m* threads if specified alone or with the `threads` attribute; if specified with the `nodes` attribute, execution is restricted to *m* `nodes` running one thread each. If specified with the `chunk_size = n` attribute, the chunks are parallelized across no more than *m* threads. `max_threads = m` is useful when you know the maximum number of threads your loop will run on efficiently.

`ordered` attribute

The `ordered` attribute causes the iterations of the loop to be initiated in loop order across the processors. It is useful only in loops with manually-synchronized dependences, so it is only useful with the `loop_parallel` directive. To achieve ordered parallelism, dependences must be synchronized within ordered sections, such as those constructed using the `ordered_section` and `end_ordered_section` directives. Using `loop_parallel(ordered)` and its associated synchronization directives is covered in Chapter 6, "Advanced shared-memory programming."

`dist` attribute

The `dist` attribute tells the compiler to distribute the iterations of a loop across the currently active threads—instead of spawning new threads. `dist` should only be used with the `loop_parallel` directive and pragma inside a `parallel/end_parallel` region. The attribute to `parallel` determines the level of parallelism. See "Region parallelization" on page 142 for information on the attributes available to the `parallel` directive and pragma.

The `dist` attribute can be used with any `loop_parallel` attribute, except the `nodes` or `threads` attributes. See "`loop_parallel[(`*attribute_list*`)]`" on page 319 for a list of allowed attribute combinations.

In the following example, threads are spawned when the `parallel` directive is used. No additional threads are spawned until the `loop_parallel` directive is used without the `dist` attribute:

```
C$DIR PARALLEL (NODES, MAX_THREADS = 8), PARALLEL_PRIVATE(A, C)
C SPAWN ONE THREAD PER NODE, UP TO A MAXIMUM OF 8

      A = B   ! THIS STATEMENT WILL BE EXECUTED BY ALL 8 NODE-WAY THREADS

C$DIR LOOP_PARALLEL(DIST, MAX_THREADS = 6)
      DO I = 1, 10000
                      ! THIS LOOP WILL BE DISTRIBUTED TO AT MOST 6 OF
        X(I) = Y(I)   ! THE 8 ACTIVE NODE-WAY THREADS; THIS MEANS THAT
                      ! EACH NODE-WAY THREAD EXECUTES
                      ! ABOUT 10000/6 ITERATIONS
      ENDDO

      C = X(1)       ! THIS STATEMENT WILL BE EXECUTED BY ALL
                     ! NODE-WAY THREADS



C$DIR LOOP_PARALLEL(DIST)
      DO J = 1, 10000
                     ! THIS LOOP WILL BE DISTRIBUTED TO THE 8 ACTIVE
        Y(J) = X(J)  ! NODE-WAY THREADS, MEANING THAT EACH THREAD
                     ! EXECUTES 10000/8 ITERATIONS
      ENDDO

C$DIR LOOP_PARALLEL
      DO K = 1, 10000
                     ! SPAWN ADDITIONAL THREADS ON EACH NODE, UP TO THE
        W(K, MY_NODE()) = X(K) ! MAXIMUM AVAILABLE (TYPICALLY 8) AND
                     ! ON EACH NODE, DISTRIBUTE THE WORK ACROSS
                     ! ALL THE THREADS SO THAT
                     ! EACH THREAD EXECUTES 10000/8 ITERATIONS
      ENDDO
C$DIR END_PARALLEL
```

`loop_parallel` and `loop_parallel(dist)` directives can be nested as long as node-parallel loops are outside all thread-parallel loops. The compiler will pick the loop that is most appropriate for the directive or pragma being processed (the loop picked is usually the outermost parallel loop).

Do not use the save_last directive or pragma for private variables with the dist attribute. For example, the following line of code is illegal:

```
C ILLEGAL USAGE
C$DIR LOOP_PARALLEL(DIST), LOOP_PRIVATE(X), SAVE_LAST
```

prefer_parallel

The prefer_parallel directive and pragma cause the compiler to parallelize the immediately following loop if it is free of dependences and other parallelization inhibitors. The compiler automatically privatizes any loop variables that must be privatized. On multihypernode subcomplexes, when prefer_parallel is specified without a nodes or threads attribute, the compiler will determine if opportunities for parallelism exist within the loop and, if possible, parallelize the loop across threads. If the threads attribute is specified, the compiler attempts to find and exploit thread-parallelism within the loop. If the nodes attribute is specified, the compiler trys to locate and exploit node-parallelism within the loop. prefer_parallel requires less manual intervention and is less forceful than the loop_parallel directive and pragma.

prefer_parallel can also be used to indicate the preferred loop in a nest to parallelize, as shown in the following Fortran example:

```
      DO J = 1, 100
C$DIR   PREFER_PARALLEL
        DO I = 1, 100
           .
           .
           .
        ENDDO
      ENDDO
```

In this example, PREFER_PARALLEL causes the compiler to choose the innermost loop for parallelization, provided it is free of dependences. PREFER_PARALLEL does not inhibit loop interchange.

An analogous C example follows:

```
for(j=0;j<100;j++)
  #pragma _CNX prefer_parallel
  for(i=0;i<100;i++) {
     .
     .
     .
  }
```

The `ordered` attribute should not be used in a `prefer_parallel` directive, as it is only useful if the loop contains synchronized dependences, and `prefer_parallel` will not parallelize a loop containing any dependences. This attribute is useful in the `loop_parallel` directive, as described in Chapter 6, "Advanced shared-memory programming."

loop_parallel

The `loop_parallel` directive forces parallelization of the immediately following loop. The compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must synchronize any dependences manually, manually privatize loop data as necessary, and if you wish to exploit nested parallelism, you must explicitly do so using directives or pragmas to define thread-parallel constructs within the node-parallel loop. In absence of a `nodes` or `threads` attribute, `loop_parallel` defaults to thread parallelization.

The section "Critical sections" on page 146 contains an example of using `loop_parallel` to parallelize a loop with a dependence; the dependence is manually handled in a critical section.

The `threads`, `nodes`, `chunk_size` = *n* and `max_threads` = *m* attributes and combinations of these attributes have exactly the same effect as explained for `prefer_parallel`. `loop_parallel(ordered)` is useful for manually parallelizing loops containing manually-ordered dependences as described in Chapter 6, "Advanced shared-memory programming."

Parallelizing loops with calls

`loop_parallel` can be useful for manually parallelizing loops containing procedure calls. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
      DO I = 1, N
        X(I) = FUNC(I)
      ENDDO
```

The call to `FUNC` in this loop would normally prevent it from parallelizing. However, if you are sure that `FUNC` has no side effects (i.e., it does not modify its argument, it does not modify the same memory location from one call to the next, it performs no I/O, and it does not call any procedures that have side effects) and is compiled for reentrancy using the `-re` option (the default on SPP Series compilers), this loop can be safely parallelized as shown. If `FUNC` does have side effects or is not reentrant, this loop may yield wrong answers.

An analogous C example follows:

```
#pragma _CNX loop_parallel
  for(i=0;i<n;i++)
    x[i] = func(i);
```

Note

In some cases, global register allocation can interfere with `loop_parallel` loops that contain procedure calls. Refer to the "Global register allocation" section of Chapter 3, "Compiler optimizations," for more information.

Unparallelizable loops

The compiler will not parallelize any loop that does not have a number of iterations determinable prior to loop invocation at execution time, even when `loop_parallel` is specified. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL
      DO WHILE(A(I) .GT. 0)!WILL NOT PARALLELIZE
        .
        .
        A(I) = ...
        .
        .
      ENDDO
```

There is no way the compiler can determine the loop's iteration count prior to loop invocation here, so the loop cannot be parallelized.

An analogous C example follows:

```
#pragma _CNX loop_parallel
while(a(i) > 0) { /* will not parallelize */
   .
   .
   a[i] = ...
   .
   .
}
```

Task parallelization

The compiler does not automatically parallelize code outside a loop, but you can use tasking directives and pragmas to instruct the compiler to parallelize such code. The `begin_tasks` directive and pragma tells the compiler to begin parallelizing a series of tasks. The `next_task` directive and pragma marks the end of a task and the start of the next task. The `end_tasks` directive and pragma marks the end of a series of tasks to be parallelized and prevents execution from continuing until all tasks have completed. The sections of code delimited by these directives is referred to as a *task list.*

Within a task list, the compiler does not check for data dependences, perform variable privatization, or perform parallelization analysis. You must synchronize any dependences between tasks manually, manually privatize data as necessary, and if you wish to exploit nested parallelism, you must explicitly do so using directives or pragmas to define thread-parallel constructs within the node-parallel task list. In absence of a `nodes` or `threads` attribute, `begin_tasks` defaults to thread parallelization.

Basic shared memory

The Fortran tasking directives have the following forms:

```
C$DIR BEGIN_TASKS[(attribute-list)]
C$DIR NEXT_TASK
C$DIR END_TASKS
```

The C tasking pragmas have the following forms:

```
#pragma _CNX begin_tasks[(attribute-list)]
#pragma _CNX next_task
#pragma _CNX end_tasks
```

The optional *attribute-list* can contain one of the following attributes:

- `ordered`
- `nodes`
- `threads`
- `max_threads=`*m*
- `ordered, nodes`
- `ordered, threads`
- `ordered, max_threads=`*m*
- `nodes, max_threads=`*m*
- `threads, max_threads=`*m*
- `ordered, nodes, max_threads=`*m*
- `ordered, threads, max_threads=`*m*

The `ordered` attribute causes the tasks to be initiated in their lexical order; i.e., the first task in the sequence begins to run on its respective thread before the second and so on. In the absence of the `ordered` argument, the starting order will be indeterminate. While this argument ensures an ordered starting sequence, it does not provide any synchronization between tasks, and does not guarantee any particular ending order. You can manually synchronize the tasks as described in Chapter 6, "Advanced shared-memory programming," if necessary.

The `nodes` attribute causes the tasks to run node-parallel, on one thread per available hypernode. If you want to exploit nested parallelism inside a `begin_tasks(nodes)` list, you must do so explicitly using thread-parallel constructs within the tasks.

The `threads` attribute causes the tasks to run thread-parallel, and is the default. As with parallel loops, node-parallelism cannot be nested within thread-parallelism in task lists.

The `ordered, nodes` and `ordered, threads` attributes cause the tasks to run ordered node-parallel and ordered thread-parallel, respectively.

The attributes specifying `max_threads = m` will run on no more than *m* threads, where *m* is an integer constant of known value at compile time. As shown, these attributes can include any combination of thread- or node-parallel, ordered or unordered execution.

Caution

Do not use tasking directives or pragmas unless you ensure that dependences do not exist or you insert your own synchronization code, if necessary, in the code delimited by the tasking directives or pragmas. The compiler performs no dependence checking or synchronization on the code in these regions. Synchronization is discussed in Chapter 6, "Advanced shared-memory programming."

The following Fortran example shows how to insert tasking directives into a section of code containing three tasks that can be run in parallel:

```
C$DIR BEGIN_TASKS
           parallel task 1
C$DIR NEXT_TASK
           parallel task 2
C$DIR NEXT_TASK
           parallel task 3
C$DIR END_TASKS
```

The example above specifies thread-parallelism by default. The compiler transforms the code into a parallel loop and creates machine code equivalent to the following Fortran:

```
C$DIR LOOP_PARALLEL(THREADS)
      DO 40 I = 1,3
        GOTO (10,20,30)I
10      parallel task 1
        GOTO 40
20      parallel task 2
        GOTO 40
30      parallel task 3
        GOTO 40
40    CONTINUE
```

If there are more tasks than available threads, some threads will execute multiple tasks; if there are more threads than tasks, some threads will not execute tasks.

The `END_TASKS` directive and pragma acts as a barrier; all parallel tasks must complete before the code following `END_TASKS` can execute.

Examples

The following Fortran example illustrates how to use these directives to specify simple task-parallelism:

```
C$DIR BEGIN_TASKS
      DO I = 1, N - 1
        A(I) = A(I+1) + B(I)
      ENDDO
C$DIR NEXT_TASK
      CALL TSUB(X,Y)
C$DIR NEXT_TASK
      C(1:1000:2) = D(1:500)
C$DIR END_TASKS
```

In this example, one thread executes the `DO I` loop, another thread executes the `CALL TSUB(X,Y)`, and a third thread assigns the elements of the array `D` to every other element of `C`. These threads execute in parallel, but their starting and ending orders are indeterminate.

Unless the `nodes` attribute is supplied with the `BEGIN_TASKS` directive, the tasks are thread-parallelized. This means that there is no room for nested parallelization within the individual parallel tasks of this example, so the forward LCD on the `DO I` loop is inconsequential; there is no way for the loop to run but serially. The Fortran 90 array assignment in the last task will not parallelize either, even though it is technically parallelizable.

An analogous C example follows:

```
#pragma _CNX begin_tasks, task_private(i)
for(i=0;i<n-1;i++)
  a[i] = a[i+1] + b[i];
#pragma _CNX next_task
tsub(x,y);
#pragma _CNX next_task
for(i=1;i<=500;i++)
  c[i*2-1] = d[i];
#pragma _CNX end_tasks
```

The loop induction variable `i` must be manually privatized here because it is used to control loops in two different tasks. If `i` was not private, both tasks would modify it, causing wrong answers. This is not necessary in the Fortran example because the second loop is implemented as a Fortran 90 array assignment, for which the compiler generates an independent induction variable. The `task_private` directive and pragma is described in detail in the section "`task_private`" on page 158.

Nested task parallelism is also possible. In order to nest any parallelism on SPP Series machines, thread-parallelism must be

nested within node-parallelism; when nesting tasking directives or pragmas, `begin_tasks(nodes)` must enclose `begin_tasks(threads)`. Also, if a node-parallel task contains a parallel loop, the loop cannot go node-parallel; if you want to parallelize it, you must explicitly do so using a thread-parallel construct. Thread-parallelism nested within node-parallelism can only run on the threads of the hypernode it is contained within.

The following Fortran example is more involved and exploits two-dimensional parallelism:

```
C$DIR BEGIN_TASKS(NODES)
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1,N
        IF(B(I) .NE. 0) THEN
          A(I) = B(I)*C(I)
        ELSE
          A(I) = C(I)*D(I)
        ENDIF
      ENDDO
C$DIR NEXT_TASK
C$DIR   BEGIN_TASKS(THREADS)
        CALL T1SUB()
C$DIR   NEXT_TASK
        CALL T2SUB()
C$DIR   NEXT_TASK
        CALL T3SUB()
C$DIR   END_TASKS                  !(THREADS)
C$DIR NEXT_TASK
      X(1:1000) = Y(1:1000)
C$DIR END_TASKS                    !(NODES)
```

Here, the first node-parallel task contains a `LOOP_PARALLEL(THREADS)` loop that goes parallel on the threads of the hypernode on which this task is running. The second node-parallel task contains a task list of three subroutine calls, each of which run on a separate thread within the hypernode. The third node-parallel task contains a Fortran 90 array section assignment which, while parallelizable, runs serially because automatic parallelization analysis is disabled within the task list.

An analogous C example follows:

```
#pragma _CNX begin_tasks(nodes)
#pragma _CNX loop_parallel(threads, ivar=i)
for(i=0;i<n;i++)
  if(b[i] != 0)
    a[i][j] = b[i]*c[i];
  else
    a[i] = c[i]*d[i];
#pragma _CNX next_task
  #pragma _CNX begin_tasks(threads)
  t1sub();
  #pragma _CNX next_task
  t2sub();
  #pragma _CNX next_task
  t3sub();
  #pragma _CNX end_tasks    /* (threads) */
#pragma _CNX next_task
for(j=0;j<1000;j++)
  x[j] = y[j];
#pragma _CNX end_tasks      /* (nodes) */
```

Task parallelism can become even more involved, as described in Chapter 6, "Advanced shared-memory programming."

Region parallelization

A parallel region is a single block of code that is written to run in its entirety on separate processors; typically, parallelism is exploited in a parallel region by conditionally executing certain code based on thread ID.

Region parallelism differs from task parallelism in that parallel tasks are separate, contiguous blocks of code; when parallelized using the tasking directives and pragmas, each block generally runs on a separate thread, whereas a single parallel region runs on several threads. Specifying parallel tasks is also typically less time consuming because each thread's work is implicitly defined by the task boundaries; in region parallelism, you must manually modify the region to identify thread-specific code.

The beginning of a parallel region is denoted by the `parallel` directive or pragma; the end is denoted by the `end_parallel` directive or pragma. `end_parallel` also prevents execution from continuing until all copies of the parallel region have completed.

Within a region, the compiler does not check for data dependences, perform variable privatization, or perform

parallelization analysis; you must synchronize any dependences between copies of the region manually, manually privatize data as necessary, and if you wish to exploit nested parallelism, you must explicitly do so by enclosing thread-parallel constructs (`loop_parallel(threads)` loops, or thread-parallel tasks or regions) within a `parallel(nodes)` region. In absence of a `nodes` or `threads` attribute, `parallel` defaults to thread parallelization.

The `parallel/end_parallel` Fortran directives have the following form:

```
C$DIR PARALLEL[(attribute-list)]
C$DIR END_PARALLEL
```

The C pragmas have the following form:

```
#pragma _CNX parallel(attribute-list)
#pragma _CNX end_parallel
```

The optional *attribute-list* can contain one of the following attributes:

- `nodes`
- `threads`
- `max_threads=` *m*
- `nodes, max_threads=` *m*
- `threads, max_threads=` *m*

The `nodes` attribute causes the region to run node-parallel, on one thread per available hypernode. If you want to exploit nested parallelism inside a `parallel(nodes)` list, you must do so explicitly using thread-parallel constructs within the region.

The `threads` attribute causes the region to run thread-parallel, and is the default. As with parallel loops, node-parallelism cannot be nested within thread-parallelism in regions.

The `max_threads =` *m* attribute will cause the region to run on no more than *m* threads, where *m* is an integer constant of known value at compile time. As shown, these attributes can include any combination of thread- or node-parallel execution.

Caution

Do not use the parallel region directives or pragmas unless you ensure that dependences do not exist or you insert your own synchronization code, if necessary, in the region. The compiler performs no dependence checking or synchronization on the code delimited by the parallel region directives and pragmas. Synchronization is discussed in Chapter 6, "Advanced shared-memory programming."

Consider the following Fortran example:

```
      REAL A(1000,8), B(1000,8), C(1000,8), RDONLY(1000), SUM(8)
      INTEGER MYTID
      .
      .
      .
C     FIRST INITIALIZATION OF RDONLY IN SERIAL CODE:
      CALL INIT1(RDONLY)
      IF(NUM_THREADS .LT. 8) THEN STOP "NOT ENOUGH THREADS; EXITING"
C$DIR PARALLEL(MAX_THREADS = 8), PARALLEL_PRIVATE(I, J, K, MYTID)
      MYTID = MY_THREAD()
      DO I = 1, 1000
        A(I, MYTID) = B(I, MYTID) * RDONLY(I)
      ENDDO
      IF(MYTID .EQ. 0) THEN ! ONLY THREAD 0 EXECUTES SECOND
        CALL INIT2(RDONLY)  ! INITIALIZATION
      ENDIF
      DO J = 1, 1000
        B(J, MYTID) = B(J, MYTID) * RDONLY(J)
        C(J, MYTID) = A(J, MYTID) * B(J, MYTID)
      ENDDO
      DO K = 1, 1000
        SUM(MYTID) = SUM(MYTID) + A(K,MYTID) + B(K,MYTID) + C(K,MYTID)
      ENDDO
C$DIR END_PARALLEL
```

In this example, all arrays that are written to in the parallel code have one dimension for each of the anticipated number of parallel threads, so that each thread can work on disjoint data and there is no chance of two threads attempting to update the same element, and therefore no need for explicit synchronization. The RDONLY array is one-dimensional, but it is never written to by parallel threads. Before the parallel region, RDONLY is initialized in serial code.

The PARALLEL_PRIVATE directive is used to privatize the induction variables used in the parallel region. This must be done so that the various threads processing the region do not attempt to write to the same shared induction variables. PARALLEL_PRIVATE is covered in more detail in the section "parallel_private" on page 160.

At the beginning of the parallel region, the NUM_THREADS intrinsic, which is described in detail in Chapter 6, "Advanced shared-memory programming," is called to ensure that the expected number of threads are available. Then the MY_THREAD intrinsic, which is also described in Chapter 6, is called by each thread to determine its thread ID; all subsequent code in the region is executed based on this ID. In the I loop, each thread

computes one row of A using RDONLY and the corresponding row of B.

RDONLY is reinitialized in a subroutine call that is only executed by thread 0 before it is used again in the computation of B in the J loop, where again each thread computes a row. The J loop similarly computes C.

Finally, the K loop sums each dimension of A, B and C into the SUM array. No synchronization is necessary here because each thread is running the entire loop serially, and assigning into a discrete element of SUM.

An analogous C example follows:

```
float a[8][1000], b[8][1000], c[8][1000], rdonly[1000], sum[8]
int i, j, k, mytid;
   .
   .
   .
/* first initialization of rdonly in serial code: */
init1(rdonly);
if(num_threads() < 8) {
  fprintf(stderr, "not enough threads; exiting\n");
  exit(2);
}
#pragma _CNX parallel(max_threads = 8), parallel_private(i,j,k,mytid)
mytid = my_thread();
for(i=0; i<1000; i++)
  a[mytid][i] = b[mytid][i] * rdonly[i];
if(mytid == 0) then init2(rdonly);
for(j=0; j<1000; j++) {
  b[mytid][j] = b[mytid][j] * rdonly[j];
  c[mytid][j] = a[mytid][j] * b[mytid][j];
}
for(k=0; k<1000; k++)
  sum[mytid] = sum[mytid] + a[mytid][k] + b[mytid][k] + c[mytid][k];
#pragma _CNX end_parallel
```

Critical sections

The critical_section and end_critical_section directives and pragmas allow you to specify sections of code in parallel loops or tasks that must be executed by only one thread at a time. These directives cannot be used for ordered synchronization within a loop_parallel(ordered) loop, but are suitable for simple synchronization in any other loop_parallel loops.

A critical_section directive or pragma and its associated end_critical_section must appear in the same procedure, but they do not have to appear in the same procedure as the parallel construct in which they are used; i.e., the pair can appear in a procedure called from a parallel loop.

As discussed in this chapter, these directives have the following form in Fortran:

```
C$DIR CRITICAL_SECTION
C$DIR END_CRITICAL_SECTION
```

The C pragmas have the following form:

```
#pragma _CNX critical_section
#pragma _CNX end_critical_section
```

The critical_section directive and pragma can take an optional gate attribute that allows the declaration of multiple critical sections as described in Chapter 6, "Advanced shared-memory programming;" however, we will only discuss simple critical sections here.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(FUNCTEMP)
      DO I = 1, N   ! LOOP IS PARALLELIZABLE
        .
        .
        .
        FUNCTEMP = FUNC(X(I))
C$DIR   CRITICAL_SECTION
        SUM = SUM + FUNCTEMP
C$DIR   END_CRITICAL_SECTION
        .
        .
        .
      ENDDO
```

Here, because FUNC has no side effects and can be called in parallel, the I loop can be parallelized as long as the SUM variable

is only updated by one thread at a time. The critical section created around SUM ensures this behavior.

The LOOP_PARALLEL directive and the critical section are required to parallelize this loop because the call to FUNC would normally inhibit parallelization. If this call were not present, and if the loop did not contain other parallelization inhibitors, the compiler would automatically parallelize the reduction of SUM as described in the section "Reductions" on page 121. However, the presence of the call necessitates the LOOP_PARALLEL directive, which prevents the compiler from automatically handling the reduction, and this, in turn, requires the critical section. Placing the call to FUNC outside of the critical section allows FUNC to be called in parallel, decreasing the amount of serial work within the critical section.

An analogous C example follows:

```
#pragma _CNX loop_parallel(ivar = i)
#pragma _CNX loop_private(functemp)
for(i=0;i<n;i++) {  /* loop is parallelizable */
   .
   .
   .
   functemp = func(x(i));
   #pragma _CNX critical_section
   sum = sum + functemp;
   #pragma _CNX end_critical_section
   .
   .
   .
}
```

In order to justify the cost of the compiler-generated synchronization code associated with the use of critical sections, loops that contain them must also contain a large amount of parallelizable (non-critical section) code. If you are unsure of the profitability of using a critical section to help parallelize a certain loop, time the loop with and without the critical section to see if parallelization justifies the overhead of the critical section.

-noautopar compiler option

You can disable *automatic* loop thread-parallelism by specifying the -noautopar option on the compiler command line. -noautopar is only meaningful when specified with the -O3 option.

This option causes the compiler to parallelize only those loops that are immediately preceded by a loop_parallel or prefer_parallel directive or pragma; all other loops, even if they could normally be automatically parallelized, are not analyzed for parallelization. Because the compiler does not automatically find parallel tasks or regions, user-specified task and region parallelization is not affected by this option.

-nonodepar compiler option

You can disable task and loop node-parallelism by using the -nonodepar option on the compiler command line. This option is only meaningful when specified with the -O3 option.

The -nonodepar option causes the compiler to ignore all directives containing the nodes attribute. The compiler generates a warning message for each ignored directive. The entire directive—not just the attribute—is disregarded. Thread-parallelism is still implemented.

Reentrant compilation

SPP Series Convex Fortran and C compilers both compile for reentrancy by default. Reentrant compilation causes procedures to store uninitialized local variables on the stack; no locals can carry values from one invocation of the procedure to the next (unless the variables appear in Fortran COMMON blocks or DATA or SAVE statements). This allows loops containing procedure calls to be manually parallelized, assuming no other inhibitors of parallelization exist.

When procedures are called in parallel, each thread receives a private stack on which to allocate local variables. This allows each parallel copy of the procedure to manipulate its local variables without interfering with any other copy's locals of the same name. When the procedure returns and the parallel threads join, all values on the stack are lost.

Because the compiler cannot tell if a procedure is going to be called in parallel at runtime, it must create both parallel and serial copies—called *clones*—of every profitably parallelizable loop in

the procedure. At runtime, the appropriate clone is executed based on whether the procedure is being called in parallel. This is similar to the dynamic selection optimization (discussed in Chapter 3, "Compiler optimizations") but dynamic selection for reentrancy is not workload-based, and is not subject to any limit on the number of loop clones. Dynamic selection for reentrancy is unaffected by the compiler options, directives and pragmas used with workload-based dynamic selection.

Reentrant compilation can be disabled with the `-nore` option; keep in mind that calling a procedure that is compiled with this option in parallel will result in wrong answers.

Default stack size

Thread 0's stack can grow to the size specified in the `maxssiz` operating system tunable. Refer to the *SPP-UX System Administration Guide* for more information on tunables.

Any threads your program spawns (as the result of `loop_parallel` or tasking directives or pragmas, for example) receive a default stack size of 8 Mbytes. This means that if a parallel construct declares more than 8 Mbytes of `loop_private` or `task_private` data, or if a subroutine with more than 8 Mbytes of local data is called in parallel, you must modify the stack size of the spawned threads via the `CPS_STACK_SIZE` environment variable. Under `csh`, this can be done with the following command:

`setenv CPS_STACK_SIZE` *size_in_kbytes*

Where *size_in_kbytes* is the desired stack size in kbytes. This value is read at program startup; it cannot be changed during execution.

Loop-specific, task-specific, and region-specific data privatization

Once assigned, the memory classes discussed in detail in Chapter 5, "Memory classes," are in effect throughout your entire program. Any loops that manipulate variables that have been explicitly assigned a memory class must be manually parallelized, and once a variable is assigned a class, its class cannot change. While very efficient programs can be written using these memory classes, they also require a great deal of manual intervention.

To get around these problems, the Convex SPP Series C and Fortran compilers support the `loop_private`, `task_private` and `parallel_private` directives and pragmas. The `save_last` directive and pragma is provided to save the value of `loop_private` data objects assigned in the last iteration of the loop. These directives and pragmas allow you to easily privatize parallel loop or task or region data temporarily; when used with `prefer_parallel`, they do so without inhibiting any automatic compiler optimizations. They can help you further increase the performance of your shared-memory program with less extra work than is required when using the standard memory classes accompanying manual parallelization and synchronization.

You can use these directives on local variables and arrays of any type, but they should not be used on data assigned one of the static or dynamic memory classes (`thread_private`, `node_private`, `near_shared`, `far_shared` or `block_shared`).

In some cases, data declared `loop_private`, `task_private`, or `parallel_private` is stored on the stacks of the spawned threads. Spawned thread stacks default to 8 Mbytes in size; if the amount of `loop_private`, `task_private` or `parallel_private` data declared exceeds this, you can use the `CPS_STACK_SIZE` environment variable to increase the default. Refer to the "Default stack size" section of this chapter for more information.

loop_private

The `loop_private` directive and pragma declares a list of variables and/or arrays private to the immediately following Fortran `DO` or C `for` loop. The compiler assumes that data objects declared to be `loop_private` have no loop-carried dependences with respect to the parallel loops in which they are used. If dependences exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, "Advanced shared-memory programming."

loop_private data must be local to the procedure in which it is declared loop_private. loop_private array dimensions must be determinable at compile-time.

Each parallel thread of execution receives a private copy of the loop_private data object for the duration of the loop; no starting values can be assumed for the data, and unless a save_last directive or pragma is specified (as described in a following section), no ending value can be assumed. If a loop_private data object is referenced within an iteration of the loop, it must have been assigned a value previously on that same iteration.

In Fortran, the LOOP_PRIVATE directive has the following form:

C$DIR LOOP_PRIVATE(*namelist*)

In C, the pragma has the following form:

#pragma _CNX loop_private(*namelist*)

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following loop. *namelist* cannot contain structures or dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
C$DIR LOOP_PRIVATE(S)
      DO I = 1, N
C       S IS ONLY CORRECTLY PRIVATE IF AT LEAST
C       ONE IF TEST PASSES ON EACH ITERATION:
        IF(A(I) .GT. 0) S = A(I)
        IF(U(I) .LT. V(I)) S = V(I)
        IF(X(I) .LE. Y(I)) S = Z(I)
        B(I) = S * C(I) + D(I)
      ENDDO
```

An apparent LCD on S exists in this example; if none of the IF tests are true on a given iteration, the value of S must wrap around from the previous iteration. The LOOP_PRIVATE(S) directive indicates to the compiler that S does, in fact, get assigned on every iteration, and therefore it is safe to parallelize this loop.

If on any iteration none of the IF tests pass, an actual LCD exists and privatizing S will result in wrong answers.

An analogous C example follows:

```
#pragma _CNX loop_private(s)
for(i=0;i<=n;i++) {
/* s is only private if at least one if
   test passes: */
  if(a[i] > 0) s = a[i];
  if(u[i] < v[i]) s = v[i];
  if(x[i] < y[i]) s = z[i];
  b[i] = s * c[i] + d[i];
}
```

Using `loop_private` with `loop_parallel`

Because the compiler does not automatically perform variable privatization in `loop_parallel` loops, you must manually privatize loop data requiring privatization. This can be easily done using the `loop_private` directive or pragma.

Consider the following Fortran example:

```
      SUBROUTINE PRIV(X,Y,Z)
      REAL X(1000), Y(4,1000), Z(1000)
      REAL XMFIED(1000)
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(XMFIED, J)
      DO I = 1, 4
C     INITIALIZE XMFIED; MFY MUST NOT WRITE TO X:
        CALL MFY(X, XMFIED)
        DO J = 1, 999
          IF (XMFIED(J) .GE. Y(I,J)) THEN
            Y(I,J) = XMFIED(J) * Z(J)
          ELSE
            XMFIED(J+1) = XMFIED(J)
          ENDIF
        ENDDO
      ENDDO
      END
```

Here, the `LOOP_PARALLEL` directive is required to parallelize the `I` loop because of the call to `MFY`. The `X` and `Y` arrays are in shared memory by default. `X` and `Z` are not written to, and the portions of `Y` written to in the `J` loop's `IF` statement are disjoint, so these shared arrays require no special attention. The local array `XMFIED`, however, is written to. But because `XMFIED` carries no values into or out of the `I` loop, it can be privatized using `LOOP_PRIVATE`. This gives each thread running the `I` loop its own private copy of `XMFIED`, eliminating the expensive necessity of synchronized access to `XMFIED`. Note that a loop-carried dependence exists for `XMFIED` in the `J` loop, but since this loop runs serially on each processor, this dependence is safe.

J is privatized as discussed in the section "Privatizing induction variables in nested loops" on page 156.

An analogous C example follows:

```
void priv(float x[1000], float y[4][1000], float z[1000]) {
  float xmfied[1000];
  int i,j;
#pragma _CNX loop_parallel(ivar=i), loop_private(xmfied,j)
  for(i=0;i<4;i++) {
    mfy(x,xmfied);
    for(j=0;j<999;j++) {
      if(xmfied[j] >= y[i][j]) y[i][j] = xmfied[j]*z[j];
      else xmfied[j+1] = xmfied[j];
    }
  }
}
```

Denoting induction variables in parallel loops

To safely parallelize a loop with the `loop_parallel` directive or pragma, the compiler must be able to correctly determine the loop's primary induction variable.

The compiler can find primary Fortran `DO` loop induction variables; it may, however, have trouble with `DO WHILE` or hand-rolled Fortran loops, and with all `loop_parallel` loops in C. Therefore, when you use the `loop_parallel` directive or pragma to manually parallelize a loop other than an explicit Fortran `DO` loop, you should indicate the loop's primary induction variable using the `IVAR=`*indvar* attribute to `loop_parallel`. Consider the following Fortran example:

```
      I = 1
C$DIR LOOP_PARALLEL(IVAR = I)
10    A(I) = ...
      .
      .           ! ASSUME NO DEPENDENCES
      .
      I = I + 1
      IF(I .LE. N) GOTO 10
```

This is a hand-rolled loop that uses `I` as its primary induction variable. To ensure parallelization, the `LOOP_PARALLEL` directive has been placed immediately before the start of the loop, and the induction variable, `I`, has been specified.

Primary induction variables in C loops can be difficult for the compiler to find, so `ivar` is required in all `loop_parallel` C loops. Its use is shown in the following example:

```
#pragma _CNX loop_parallel(ivar = i)
  for(i=0; i<n; i++) {
    a[i] = ...;
    .
    . /* assume no dependences */
    .
  }
}
```

Secondary induction variables are variables used to track loop iterations even though they do not appear in the Fortran `DO` statement. They cannot appear in addition to the primary induction variable in the C `for` statement. Such variables *must* be a function of the primary loop induction variable; they cannot be independent. Secondary induction variables must also either be assigned a memory class manually (as described in Chapter 5, "Memory classes") or declared `loop_private`.

The following Fortran example contains an incorrectly incremented secondary induction variable:

```
C WARNING: INCORRECT EXAMPLE!!!!
      J = 1
C$DIR LOOP_PARALLEL
      DO I = 1, N
        J = J + 2  ! WRONG!!!
```

Here, `J` is *not* a legal secondary induction variable because it is not a function of `I`. It is not private because the value assigned to it in the current iteration is a function of its value in the previous iteration. This example can be corrected by privatizing `J` and making it a function of `I`, as shown below.

```
C CORRECT EXAMPLE:
      J = 1
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(J)  ! J IS PRIVATE
      DO I = 1, N
        J = (2*I)+1  ! J IS PRIVATE
```

Here `J` is a legal secondary induction variable because it is a function of `I`, and can be safely privatized.

In C, secondary induction variables are sometimes included in for statements, as shown in the following example:

```
/* warning: unparallelizable code follows */
#pragma _CNX loop_parallel(ivar = i)
  for(i=j=0; i<n;i++,j+=2) {
    a[i] = ...;
    .
    .
    .
  }
}
```

Because secondary induction variables must be private to the loop and must be a function of the primary induction variable, this example cannot be safely parallelized using loop_parallel(ivar = i). In the presence of this directive, the secondary induction variable will not be recognized. To manually parallelize this loop, you must remove j from the for statement and either privatize it and make it a function of i, or declare j to be shared (which is the default storage class) and increment it within a critical section inside the loop. This latter method is costly in terms of synchronization overhead and may degrade the performance of the loop.

The following example demonstrates how to restructure the loop so that j is a valid secondary induction variable:

```
#pragma _CNX loop_parallel(ivar = i)
#pragma _CNX loop_private(j)
  for(i=0; i<n; i++) {
    j = 2*i;
    a[i] = ...;
    .
    .
    .
  }
}
```

This method runs faster than placing j in a critical section because it requires no synchronization overhead, and the private copy of j used here can typically be more quickly accessed than a shared variable.

Privatizing induction variables in nested loops

The induction variables of nonparallel loops that are contained within parallel loops must be declared `loop_private` with respect to their closest enclosing parallel loop.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(THREADS)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, N      ! I LOOP GOES PARALLEL
        DO J = 1, M    ! J LOOP IS SERIAL
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the `LOOP_PARALLEL` causes the `I` loop to be parallelized across threads. The `J` loop, then, runs serially. `J` must be private with respect to the `I` loop so that the threads that run the `I` loop do not attempt to update the same copy of `J`. If the loop is automatically parallelized by the compiler, or parallelized due to the presence of a `PREFER_PARALLEL` directive, this privatization will be automatic. But the presence of the `LOOP_PARALLEL` directive requires manual privatization.

An analogous C example follows:

```
#pragma _CNX loop_parallel(threads, ivar = i)
#pragma _CNX loop_private(j)
for(i=0;i<50;i++) {
  for(j=0;j<50;j++) {
    .
    .
    .
  }
}
```

This also applies to nested parallel outer loops. In this case loop variables contained within a parallel construct—even if they are used in a parallel loop themselves—must be declared private with respect to the innermost enclosing parallel loop. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, N       ! I LOOP GOES NODE PAR
C$DIR LOOP_PARALLEL(THREADS)
C$DIR LOOP_PRIVATE(K)
        DO J = 1, M       ! J LOOP GOES THREAD PAR
          DO K = 1, L     ! K LOOP IS SERIAL
          .
          .
          .
          ENDDO
        ENDDO
      ENDDO
```

Here, LOOP_PARALLEL is used to parallelize the I loop across hypernodes, and the J loop across processors on each hypernode. K must be declared private to the J loop to ensure that the thread-parallel threads do not interfere with each other in updating it. J must be declared private to the I loop to insure that each node-parallel thread gets its own copy.

An analogous C example follows:

```
#pragma _CNX loop_parallel(nodes, ivar=i), loop_private(j)
for(i=0;i<n;i++) {
#pragma _CNX loop_parallel(threads, ivar=j)
#pragma _CNX loop_private(k)
  for(j=0;j<m;j++) {
    for(k=0;k<l;k++) {
    .
    .
    .
    }
  }
}
```

Basic shared memory

task_private

The task_private directive declares a list of variables and/or arrays private to the immediately following tasks; it serves the same purpose for parallel tasks that loop_private serves for loops.

The task_private directive must immediately precede or appear on the same line as its corresponding begin_tasks directive. The compiler assumes that data objects declared to be task_private have no dependences between the tasks in which they are used. If dependences exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, "Advanced shared-memory programming."

Each parallel thread of execution receives a private copy of the task_private data object for the duration of the tasks; no starting or ending values can be assumed for the data. If a task_private data object is referenced within a task, it must have been assigned a value previously in that task.

In Fortran, the TASK_PRIVATE directive has the following form:

```
C$DIR TASK_PRIVATE(namelist)
```

In C, the pragma has the following form:

```
#pragma _CNX task_private(namelist)
```

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following tasks. *namelist* cannot contain dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
      REAL*8 A(1000), B(1000), WRK(1000)
      .
      .
      .
C$DIR BEGIN_TASKS, TASK_PRIVATE(WRK)
      DO I = 1, N
        WRK(I) = A(I)
      ENDDO
      DO I = 1, N
        A(I) = WRK(N+1-I)
        .
        .
        .
      ENDDO
C$DIR NEXT_TASK
      DO J = 1, M
        WRK(J) = B(J)
      ENDDO
      DO J = 1, M
        B(J) = WRK(M+1-J)
        .
        .
        .
      ENDDO
C$DIR END_TASKS
```

Here, the WRK array is used in the first task to temporarily hold the A array so that its order can be reversed. It serves the same purpose for the B array in the second task. WRK is assigned before it is used in each task.

Basic shared memory

An analogous C example follows:

```
float a[1000], b[1000], wrk[1000];
.
.
.
#pragma _CNX task_private(wrk)
#pragma _CNX begin_tasks(nodes)
for(i=0;i<n;i++)
  wrk[i] = a[i];
for(i=0;i<n;i++) {
  a[i] = wrk[n-1-i];
   .
   .
   .
}
#pragma _CNX next_task
for(j=0;j<m;j++)
  wrk[j] = b[j];
for(j=0;j<m;j++) {
  b[j] = wrk[m-1-j];
   .
   .
   .
}
#pragma _CNX end_tasks
```

parallel_private

The parallel_private directive declares a list of variables and/or arrays private to the immediately following parallel region; it serves the same purpose for parallel regions that task_private serves for tasks.

The parallel_private directive must immediately precede or appear on the same line as its corresponding parallel directive. The compiler assumes that data objects declared to be parallel_private have no dependences between the parallel copies of the region in which they are used. If dependences exist, you must handle them manually using the synchronization directives and techniques described in Chapter 6, "Advanced shared-memory programming."

Each parallel thread of execution receives a private copy of the parallel_private data object for the duration of the region; no starting or ending values can be assumed for the data. If a parallel_private data object is referenced within a region, it must have been assigned a value previously in the region.

In Fortran, the PARALLEL_PRIVATE directive has the following form:

C$DIR PARALLEL_PRIVATE(*namelist*)

In C, the pragma has the following form:

#pragma _CNX parallel_private(*namelist*)

where *namelist* is a comma-delimited list of variables and/or arrays that are to be private to the immediately following parallel region. *namelist* cannot contain dynamic, allocatable, or automatic arrays.

Consider the following Fortran example:

```
      REAL A(1000,8), B(1000,8), C(1000,8), AWORK(1000)
      INTEGER MYTID
      .
      .
      .
C$DIR PARALLEL(MAX_THREADS = 8), PARALLEL_PRIVATE(I,J,K,L,M,AWORK,MYTID)
      IF(NUM_THREADS .LT. 8) THEN STOP "NOT ENOUGH THREADS; EXITING"
      MYTID = MY_THREAD()
      DO I = 1, 1000
        AWORK(I) = A(I, MYTID)
      ENDDO
      DO J = 1, 1000
        A(J, MYTID) = AWORK(J) + B(J, MYTID)
      ENDDO
      DO K = 1, 1000
        B(K, MYTID) = B(K, MYTID) * AWORK(K)
        C(K, MYTID) = A(K, MYTID) * B(K, MYTID)
      ENDDO
      DO L = 1, 1000
        SUM(MYTID) = SUM(MYTID) + A(L,MYTID) + B(L,MYTID) + C(L,MYTID)
      ENDDO
      DO M = 1, 1000
        A(M, MYTID) = AWORK(M)
      ENDDO
C$DIR END_PARALLEL
```

This example is similar to the one presented in the "Region parallelization" section in the way it checks for a certain number of threads and divides up the work among those threads. However, the parallel_private variable AWORK is introduced.

Each thread initializes its private copy of AWORK to the values contained in a dimension of the array A at the beginning of the parallel region; this allows the threads to reference AWORK without regard to thread ID, since no thread can access any other thread's

copy of AWORK. Note that AWORK cannot carry values into or out of the region, so it must be initialized within the region.

All induction variables contained in a parallel region must be privatized. Remember that the code contained in the region runs on all available threads, so failing to privatize an induction variable would allow each thread to update the same shared variable, creating indeterminate loop counts on every thread.

In the J loop after AWORK is initialized, AWORK is effectively used in a reduction on A (since at this point its contents are identical to the MYTID dimension of A). After A is modified here and used in the K and L loops, each thread restores a dimension of A's original values from its private copy of AWORK, which carried the appropriate dimension through the region unaltered.

An analogous C example follows:

```
float a[8][1000], b[8][1000], c[8][1000], awork[1000]
int i, mytid;
   .
   .
   .
#pragma _CNX parallel(max_threads = 8)
#pragma _CNX parallel_private(i,j,k,l,m,awork,mytid)
if(num_threads() < 8) {
  fprintf(stderr, "not enough threads; exiting\n");
  exit(2);
}
mytid = my_thread();
for(i=0; i<1000; i++)
  awork[i] = a[mytid][i];
for(j=0; j<1000; j++)
  a[mytid][j] = awork[j] + b[mytid][j];
for(k=0; k<1000; k++) {
  b[mytid][k] = b[mytid][k] * awork[k];
  c[mytid][k] = a[mytid][k] * b[mytid][k];
}
for(l=0; l<1000; l++)
  sum[mytid] = sum[mytid] + a[mytid][l] + b[mytid][l] + c[mytid][l];
for(m=0; m<1000; m++)
  a[mytid][m] = awork[m];
#pragma _CNX end_parallel
```

save_last

The save_last directive and pragma allow you to save the final value of all loop_private data objects assigned in the last iteration of the immediately following loop. The values must be assigned in the last iteration; if the assignment is executed conditionally, it is your responsibility to ensure that the condition is met and the assignment executes. Incorrect answers can result if the assignment does not execute on the last iteration. For loop_private arrays, only those elements of the array assigned on the last iteration will be saved.

In Fortran, the SAVE_LAST directive has the following form:

```
C$DIR SAVE_LAST
```

In C, the pragma has the following form:

```
#pragma _CNX save_last
```

save_last must appear immediately before or after the associated loop_private directive or pragma, or on the same line.

Consider the following Fortran example:

```
C$DIR LOOP_PRIVATE(ATEMP), SAVE_LAST
      DO I = 1, N
        IF (I .EQ. D(I)) ATEMP = A(I)
        IF (I .EQ. E(I)) ATEMP = B(I)
        IF (I .EQ. F(I)) ATEMP = C(I)
        A(I) = B(I) + C(I)
        B(I) = ATEMP
      ENDDO
      .
      .
      .
      IF(ATEMP .GT. AMAX) THEN
        .
        .
        .
```

Here, the LOOP_PRIVATE variable ATEMP is conditionally assigned in the loop; in order for ATEMP to be truly private, you must be sure that at least one of the conditions is met so that ATEMP is assigned on every iteration. When the loop terminates, the SAVE_LAST directive ensures that ATEMP contains the value it is assigned on the last iteration, which is used later in the program.

An analogous C example follows:

```
#pragma _CNX loop_private(atemp), save_last
for(i=0;i<n;i++) {
  if(i==d[i]) atemp = a[i];
  if(i==e[i]) atemp = b[i];
  if(i==f[i]) atemp = c[i];
  a[i] = b[i] + c[i];
  b[i] = atemp;
}
.
.
.
if(atemp > amax) {
.
.
.
```

Note that the `save_last` directive can be misleading in certain loop contexts. Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL, LOOP_PRIVATE(S), SAVE_LAST
      DO I = 1, N
        IF(G(I) .GT. 0) THEN
          S = G(I) * G(I)
        ENDIF
      ENDDO
```

While it may appear that the last value of `S` assigned (on whatever iteration) is saved in this example, you must remember that the `SAVE_LAST` directive applies only to the last (`N`th) iteration, without regard for any conditionals contained in the loop. For `SAVE_LAST` to be valid here, `G(N)` must be greater than 0 so that the assignment to `S` takes place on the final iteration. Obviously, if this condition can be predicted, the loop can be more efficiently written to exclude the `IF` test, so the presence of a `SAVE_LAST` in such a loop is suspect.

A `save_last` directive or pragma causes the compiler to peel the last iteration of the loop to facilitate saving. In this way, all but the last iteration of the loop can be parallelized; the final iteration runs serially, and carries out the final assignment to the saved variable.

The optimization report describes such peeled variables in the loop table, the test table, and the privatization table. Refer to Appendix C for more information on the optimization report.

Performance analysis

If you are willing to make the manual modifications to your program suggested here and in Chapter 6, "Advanced shared-memory programming," you may wish to use CXpa, the Convex visual profiler, to analyze its performance and determine the areas of code that would most benefit from manual optimizations. CXpa has both an interactive graphical user interface and a character-based line mode. Through either interface, it profiles the following performance factors on a per-thread basis over user-selectable regions of application code without source modification:

- On all SPP Series architectures:
  - Wall clock time
  - CPU time
  - Concurrency factor (CPU/Wall clock time)
  - Iteration/execution counts
  - Dynamic call graph
- On SPP1000/SPP1600 systems:
  - Locally resolved (CTI ring not used) cache miss counts and latency
  - Remotely resolved (CTI ring used) cache miss counts and latency
  - Locally and remotely resolved (total) cache miss counts and latency
  - Average cache miss latency
- On SPP1200/SPP1600 systems:
  - Data cache misses, accesses, and latency
  - Data cache hit rates
  - Average data cache miss latency
  - Instruction cache misses and latency
  - Average instruction cache miss latency
  - Instructions completed
  - Clock cycles
  - Average clock cycles per instruction

CXpa supports:

- Visualization of profiling results in 2D and 3D graphs, including a dynamic call graph
- Presentation of profiling results in text reports
- Routine-level profiling of object files and archive libraries created with Hewlett-Packard PA-RISC targeting compilers such as `f77` and `c89` (refer to the cxoi(1) man page)
- Profiling of message-passing (PVM/MPI) programs

CXpa is an optional product. Refer to the *CXpa Reference* (DSW-605), to the cxpa(1) man page, or contact your Convex sales representative for more information.

Memory classes

5

Chapter 2 discusses the three partitions of physical memory available on SPP Series systems. These partitions are:

- Hypernode-local memory, which is accessed via the `thread_private` and `node_private` virtual memory classes.
- Subcomplex-global memory, which is accessed via the `near_shared`, `far_shared` and `block_shared` virtual memory classes.
- CTIcache physical memory, which holds copies of shared-memory data which is not resident in the hypernode's physical memory, but is accessed by threads running on the hypernode.

Note

The memory classes discussed here are of interest to programmers who wish to manually optimize their shared-memory programs by using compiler directives or pragmas to partition memory and otherwise control compiler optimizations as discussed in Chapter 6, "Advanced shared-memory programming." ***Using these memory classes requires you to manually handle parallelization.***

Private versus shared memory

Private and shared data are differentiated by their accessibility, and, as noted above, by the physical memory classes in which they are stored.

Both `node_private` and `thread_private` data are stored in hypernode-local memory, and are therefore inaccessible to any hypernode other than the one on which they reside (in the case of `thread_private`, access is further restricted to the declaring thread). Latency is identical for private data items that must be fetched from main memory. `near_shared`, `far_shared` and `block_shared` data, on the other hand, are stored in subcomplex-global physical memory and are therefore accessible from any hypernode in the subcomplex on which the process is running. Main memory latency can vary for the shared-memory classes depending on whether or not the data is resident on the requesting hypernode.

Memory class addressing

Figure 21 shows the virtual addresses associated with a data item stored in each memory class by a single process running on a conceptual 4-hypernode, 8-processors-per-hypernode system. Each oval represents a unique virtual address for the same data item within a class; the memory class is indicated by the oval's fill pattern as explained in the illustration.

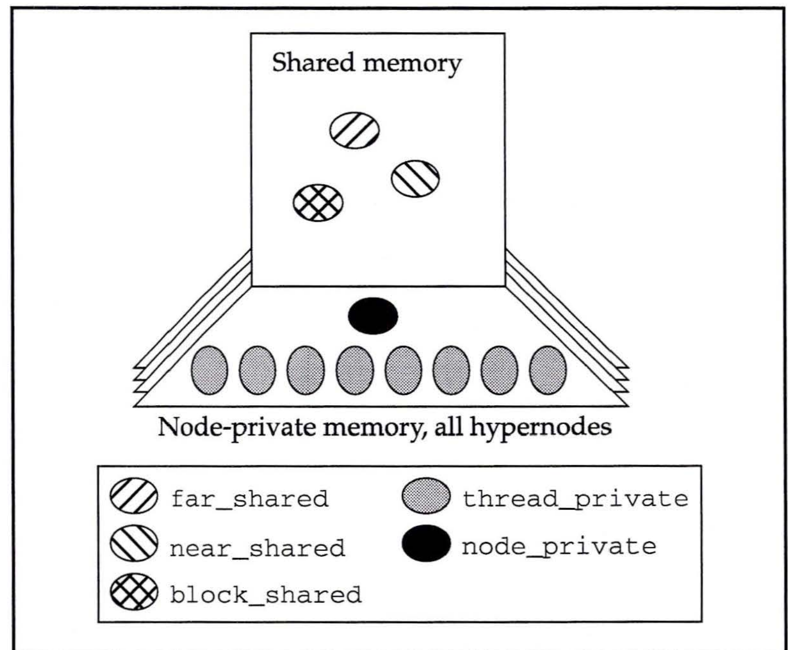


Figure 21 Virtual addresses for various memory classes

As shown, the shared data items are accessible from any hypernode using the same virtual address; the `node_private` data item has a single unique virtual address; the `thread_private` item has up to 8 unique virtual addresses, one for each thread within a hypernode.

Figure 22 shows the physical addresses associated with the data items shown in Figure 21, for an identical SPP Series system. For illustrative purposes, all shared data in Figure 22 is assumed to be array data, and is represented by circles and portions of circles rather than ovals.

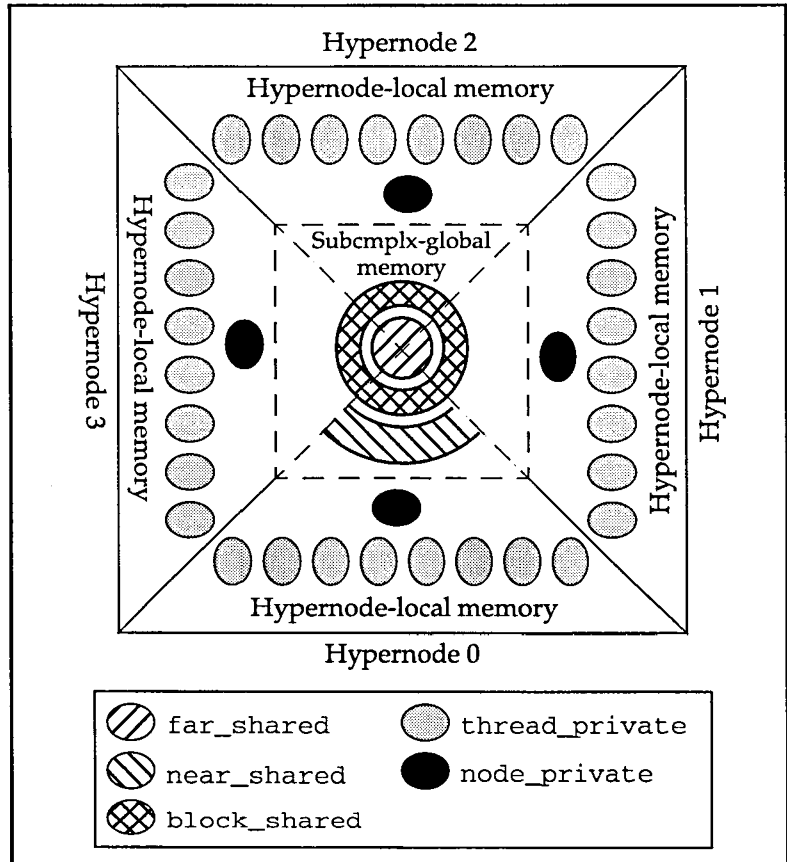


Figure 22 Physical addresses for various memory classes

Figure 22 shows the physical memory replication of the private memory classes, as well as the distributed nature of the `far_shared` and `block_shared` classes. You can control hypernode placement of the `near_shared` class, shown here residing physically on hypernode 0. All hypernodes can access this data, since it resides in subcomplex-global physical memory.

The subsections that follow explain virtual and physical addressing for each memory class in detail.

thread_private

thread_private data is private to each thread of a process. Each thread_private data object has its own unique virtual address within a hypernode. For statically-declared thread_private data on multihypernode subcomplexes, these unique virtual addresses are replicated on each hypernode. If the data is dynamically allocated, no virtual address replication is done; all virtual addresses are unique.

These virtual addresses map to unique physical addresses in hypernode-local physical memory on each hypernode; therefore, while a thread_private data item can have identical virtual addresses on different hypernodes, these virtual addresses map to unique physical addresses. For example, on a 4-hypernode, 8-processors-per-hypernode system, a single, statically allocated thread_private data item is accessed by 8 virtual addresses that map to 32 physical addresses. On a 7-hypernode, 8-processors-per-hypernode system, the same data item is accessed by 8 virtual addresses that map to 56 physical addresses.

Obviously, this physical address replication can cause a single data item to occupy a large amount of memory; similarly, virtual address replication subtracts from the total 4-Gbyte virtual address space available to the process.

Any sharing of thread_private data items between threads (regardless of whether they are running on the same hypernode) must be done by synchronized copying of the item into a shared variable, or by message passing.

thread_private data cannot be initialized in Fortran DATA statements.

node_private

node_private data is private to the threads running on a given hypernode. node_private data items have one virtual address, and any thread on a hypernode can access that hypernode's node_private data using the same virtual address. This virtual address maps to a unique physical address in hypernode-local memory. On a multihypernode subcomplex, a physical copy of the data item is contained in each hypernode's hypernode-local memory, and this copy is accessed by the same virtual address on any hypernode.

This physical replication will multiply the amount of physical memory a node_private data item takes up by the number of hypernodes in the subcomplex.

Any sharing of node_private data items between hypernodes must be done by synchronized copying into a shared variable, or by message passing.

near_shared

near_shared data is accessible by any thread running on any hypernode of a subcomplex, but is physically stored entirely within the subcomplex-global memory of a particular hypernode, allowing faster access to the threads running on that hypernode. near_shared data has a single virtual address through which it is accessed by every thread. The data's physical placement in hypernode-global memory eliminates the need for replication. Threads running on the hypernode on which the data is stored can access it via the crossbar within 60 processor clocks; threads running on other hypernodes must access the data via their CTIcache if it is encached there; if not, they must fetch it over the SCI rings. These accesses can take over 200 processor clocks, depending on whether any coherency or bank conflicts are encountered.

As explained in the "Static assignments" section later in this chapter, the near_shared class is typically used for data which is accessed heavily by threads running on the hypernode on which it resides, but which must also be easily accessible to threads running on other hypernodes.

`far_shared`

`far_shared` data is accessible by any thread running on any hypernode, and, because it is distributed evenly across the subcomplex-global memory of all hypernodes in a subcomplex, it provides the best average access time when it is used equally by threads running on all available hypernodes. A `far_shared` data item has a single virtual address and a single physical address, but the virtual pages of `far_shared` data are mapped, in an approximately round-robin manner, to physical memory pages on all the hypernodes in the subcomplex. The page size is 4 kbytes.

As explained in the "Static assignments" section later in this chapter, the `far_shared` class is typically used for data that is accessed equally by all the hypernodes in a subcomplex.

The `far_shared` memory class is the default for any data not specifically classified by the programmer.

`block_shared`

A `block_shared` data item has a unique virtual and a unique physical address, and can be accessed by any thread running on any hypernode in the subcomplex. This memory class is used to store arrays that are dynamically allocated at runtime, when the number of hypernodes on which the process is running is known. The virtual pages of the arrays are then divided into a number of chunks equal to the number of available hypernodes, and these chunks (which likely contain multiple contiguous pages each) are distributed to the subcomplex-global physical pages of the available hypernodes, 1 chunk per hypernode. If the number of pages of a `block_shared` array is not integrally divisible by the number of hypernodes, the array size is increased to allow integral division.

`block_shared` allocation is only useful when combined with manual parallelization of loops that use the arrays. In this case it allows you to parallelize the loops such that a parallel section running on a given hypernode can access the portion of the array residing on that hypernode with low latency, but interhypernode access of the remaining elements is also possible.

Using the `block_shared` class is explained in detail in the "Dynamic assignments" section later in this chapter.

Memory class assignments

In Fortran, compiler directives are used to assign memory classes to data items. In C, memory classes are assigned through the use of syntax extensions to Convex C, which are defined in header file /usr/convex/all/include/spp_prog_model.h. This file must be included in any C program that uses memory classes.

The general form for Fortran memory class directives is

C$DIR *class_name* (*namelist*)

where *namelist* is a comma-delimited list of variables, arrays, and/or COMMON block names to be assigned the class *class_name*. COMMON block names must be enclosed in slashes (/), and only entire COMMON blocks can be assigned a class. This means that arrays and variables in *namelist* must not also appear in a COMMON block, and must not be EQUIVALENCEd to data objects in COMMON blocks.

class_name can be THREAD_PRIVATE, NODE_PRIVATE, NEAR_SHARED, FAR_SHARED, or BLOCK_SHARED. For BLOCK_SHARED, *namelist* must include only allocatable arrays.

These Fortran memory class declarations must appear with other specification statements; they cannot appear within executable statements.

In C, Convex type-qualifier extensions are used, so memory classes are assigned in variable declarations. The general form for assigning memory classes in C is

```
#include <spp_prog_model.h>
.
.
.
```

[*storage_class_specifier*] *class_name type_specifier namelist*

where

storage_class_specifier

Specifies a nonautomatic storage class

class_name

Is the desired memory class (thread_private, node_private, near_shared, or far_shared; the block_shared class must be allocated dynamically, as described in the "block_shared" section on page 199)

type_specifier

Is a standard C data type (int, float, etc.), and *namelist* is a comma-delimited list of variables and/or arrays of type *type_specifier*

In C, data objects that are assigned a memory class must have static storage duration. This means that if the object is declared within a function, it must have the storage class `extern` or `static`. If such an object is not given one of these storage classes, its storage class defaults to `automatic` and it is allocated on the stack. Stack-based objects cannot be assigned a memory class, and attempting to do so will result in a compile-time error.

Data objects declared at file scope and assigned a memory class need not specify a storage class. For more information on C scoping rules, refer to *The C Programming Language* (DSW-046).

Note

All C code examples presented in this chapter assume that the line

```
#include <spp_prog_model.h>
```

appears above the C code presented. This header file maps user symbols to the implementation reserved space.

If you assign a memory class to a C structure, all structure members must be of the same class.

In both C and Fortran, once a data item is assigned a memory class, the class cannot be changed.

The Convex SPP Series compilers provide mechanisms for assigning memory classes statically and dynamically. Static assignments make sense for the private classes and for `far_shared` memory; dynamic assignments make most sense for `near_shared` and `block_shared` memory. The sections that follow explain both static and dynamic memory class assignments in detail.

Static assignments

Static memory class assignments are physically located with variable type declarations in the source. Static memory classes are typically used with data objects that are accessed with equal frequency by all threads; these include objects of the `thread_private`, `node_private`, and `far_shared` classes. Static assignments for all classes are explained in the subsections that follow.

thread_private

Since thread_private variables are replicated for every thread on every hypernode, static declarations make the most sense for them.

In Fortran, the thread_private memory class is assigned using the THREAD_PRIVATE compiler directive, as shown in the following example:

```
      REAL*8 TPX(1000)
      REAL*8 TPY(1000)
      REAL*8 TPZ(1000), X, Y
      COMMON /BLK1/ TPZ, X, Y
C$DIR THREAD_PRIVATE(TPX, TPY, /BLK1/)
```

Each array declared here is 8000 bytes in size, and each variable is 8 bytes, for a total of 24,016 bytes of data. The entire COMMON block BLK1 is placed in thread_private memory along with TPX and TPY. All memory space is replicated for each thread in hypernode-local physical memory on every hypernode.

thread_private variables and arrays cannot be initialized in Fortran DATA statements.

The following C example demonstrates several ways to declare thread_private storage in C. Note that the data objects declared here are not scoped analogously to those declared in the Fortran example:

```
/* tpa is global: */
thread_private double tpa[1000];
func() {
  /* tpb is local to func: */
  static thread_private double tpb[1000];
  /* tpc, a and b are declared elsewhere: */
  extern thread_private double tpc[1000],a,b;
  .
  .
  .
```

C's double data type provides the same precision as Fortran's REAL*8. The thread_private data declared here occupies the same amount of memory as that declared in the Fortran example. tpa is available to all functions lexically following it in the file. tpb is local to func and inaccessible to other functions. tpc, a, and b are declared at filescope in another file that is linked with this one.

Assume a Fortran or C program containing the appropriate example is running on a 4-hypernode subcomplex with 8 processors per hypernode and the `thread_private` memory is allocated from `node_private` memory (see the mpa(1) man page). Each data item will require 8 virtual addresses, for a total of 192,128 bytes of virtual space. These virtual addresses will map to 8 physical addresses per hypernode, or 32 total physical addresses per data item, requiring a total of 768,512 (32x24016) bytes of physical memory.

`thread_private` `COMMON` blocks in parallel subroutines

Data local to a procedure that is called in parallel is effectively private because storage for it is allocated on the thread's private stack. However, if the data is in a Fortran COMMON block (or if it appears in a DATA or SAVE statement), it will not be stored on the stack. Parallel accesses to such nonprivate data must be synchronized if it is assigned a shared class; or, if the parallel copies of the procedure do not need to share the data, it can be assigned a private class.

Consider the following Fortran example:

```
      INTEGER A(1000,1000)
      .
      .
      .
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, N
        CALL PARCOM(A(1,I),ISZ)
        .
        .
        .
      ENDDO

      SUBROUTINE PARCOM(A,ISZ)
      INTEGER A(*), ISZ
      INTEGER C(ISZ), D(ISZ)
      COMMON /BLK1/ C, D
C$DIR THREAD_PRIVATE(/BLK1/)
      INTEGER TEMP1, TEMP2
      D(1:ISZ) = ...
      .
      .
      .
      CALL PARCOM2(A, JTA)
      .
      .
      .
      END
```

```
      SUBROUTINE PARCOM2(B,JTA)
      INTEGER B(*), JTA
      INTEGER C(ISZ), D(ISZ)
      COMMON /BLK1/ C, D
C$DIR THREAD_PRIVATE(/BLK1/)
      DO J = 1, 1000
        C(J) = D(J) * B(J)
      ENDDO
      END
      .
      .
      .
```

Here, COMMON block BLK1 is declared THREAD_PRIVATE, so every parallel instance of PARCOM gets its own copy of the arrays C and D.

Because this code is already thread-parallel when the COMMON block is defined, no further parallelism is possible, and BLK1 is therefore suitable for use anywhere in PARCOM. The local variables TEMP1 and TEMP2 are allocated on the stack, so each thread effectively has private copies of them.

A similar concept applies to node_private COMMON blocks in node-parallel loops, as described in the following "node_private" section.

node_private

Because the space for node_private variables is physically replicated on every hypernode, static declarations make the most sense for them.

In Fortran, the node_private memory class is assigned using the NODE_PRIVATE compiler directive, as shown in the following example:

```
      REAL*8 XNP(1000)
      REAL*8 YNP(1000)
      REAL*8 ZNP(1000), X, Y
      COMMON /BLK1/ ZNP, X, Y
C$DIR NODE_PRIVATE(XNP, YNP, /BLK1/)
```

Again, the data requires 24,016 bytes. The contents of BLK1 are placed in node_private memory along with XNP and YNP. Space for each data item is replicated once per hypernode in hypernode-local physical memory. The same virtual address is used by each thread to access its hypernode's copy of a data item.

node_private variables and arrays can be initialized in Fortran DATA statements.

The following example shows several ways to declare node_private data objects in C:

```
/* npa is global: */
node_private double npa[1000];
func() {
  /* npb is local to func: */
  static node_private double npb[1000];
  /* npc, a and b are declared elsewhere: */
  extern node_private double npc[1000],a,b;
  .
  .
  .
```

The node_private data declared here occupies the same amount of memory as that declared in the Fortran example. Scoping rules for this data are similar to those given for the thread_private C example.

For either language example, assuming an 8 processor per hypernode, 4 hypernode system, each data item would require a single virtual address, for a total of 24,016 bytes of virtual space. These virtual addresses map to 4 physical addresses each, one per hypernode, for a total of 96,064 bytes of physical memory.

Because node_private data is physically replicated across hypernodes but not replicated in virtual memory, on multihypernode subcomplexes it can effectively expand the physical space available to a process. For example, if a process declares 2 Gbytes of data node_private, the virtual addresses it uses to access this data map to a total of 16 Gbytes of physical memory on an 8 hypernode subcomplex (2 Gbytes per hypernode). Assuming this node_private data is made up of an array, you could manually split up a loop that manipulates the array to run on several hypernodes. Each hypernode would then compute its entire 2 Gbytes of the array; to the hypernode, this private copy appears to be the entire array, when in fact other hypernodes are working on private 2-Gbyte chunks with identical array names (and thus identical virtual addresses) that also appear to be the entire array. Through careful manual synchronization, the results of these hypernode-private computations can be shared through the use of "communication" arrays of the far_shared or block_shared memory class.

Such an approach approximates message passing using shared-memory constructs, and can be beneficial when arrays contain quantities of data that surpass available virtual memory.

node_private COMMON blocks in parallel subroutines

Fortran COMMON blocks created in subroutines called from node-parallel loops must be handled in the same ways as those appearing in thread-parallel loops, as discussed in the preceding "thread_private" section. One way to deal with them is to assign them the node_private memory class, as shown in the following Fortran example:

```
      INTEGER A(1000,1000)
      .
      .
      .
C$DIR LOOP_PARALLEL(NODES)
      DO I = 1, N
        CALL PARCOM(A(1,I),ISZ)
        .
        .
        .
      ENDDO

      SUBROUTINE PARCOM(A,ISZ)
      INTEGER A(*), ISZ
      INTEGER C(ISZ), D(ISZ)
      COMMON /BLK1/ C, D
C$DIR NODE_PRIVATE(/BLK1/)
      INTEGER TEMP1, TEMP2
      D(1:ISZ) = ...
      .
      .
      .
      CALL PARCOM2(A, JTA)
      .
      .
      .
      END

      SUBROUTINE PARCOM2(B,JTA)
      INTEGER B(*), JTA
      INTEGER C(ISZ), D(ISZ)
      COMMON /BLK1/ C, D
C$DIR NODE_PRIVATE(/BLK1/)
      DO J = 1, 1000
        C(J) = B(J) * D(J)
      ENDDO
      END
      .
      .
      .
```

Here, PARCOM is run on one thread per available hypernode. Each hypernode gets its own copy of C and D in NODE_PRIVATE memory. All threads within a hypernode can access that node's copy of BLK1, which contains C and D, but cannot access any other node's copy of the COMMON block.

If BLK1 is declared in the calling procedure (PARCOM in the example above), it must be assigned the NODE_PRIVATE memory class there. If BLK1 is not declared in the calling procedure, but is declared in the called procedure, the COMMON block becomes inaccessible when the called procedure exits.

If PARCOM initiates a thread-parallel loop or task that modifies anything in BLK1, care must be taken to ensure that the data being modified is either thread-privatized, modified by no more than one thread at a time, or properly synchronized for shared access. Local variables TEMP1 and TEMP2 are still allocated on each thread's stack, but since each hypernode is only running one thread as the result of the I loop (and therefore only one copy of PARCOM), each hypernode only gets one copy of each of TEMP1 and TEMP2.

Placing BLK1 in shared memory (the default if no class is specified) in this example would likely cause wrong answers because each copy of PARCOM could potentially modify the same data items in BLK1. Additionally, storing a copy of BLK1 on each hypernode decreases access time compared to storing it in any shared-memory class.

near_shared

Static assignments of the near_shared memory class are of limited usefulness, because they place the declared near_shared memory on hypernode 0. The purpose of declaring the near_shared memory is defeated unless the code that most frequently accesses this data happens to be running on hypernode 0, which is unlikely on a multihypernode subcomplex.

However, a mechanism is provided for statically declaring near_shared memory. In Fortran, the near_shared memory class is statically assigned using the NEAR_SHARED compiler directive, as in the following example:

```
      SUBROUTINE FUNC()
      REAL*8 XNS(1000, 1000)
C$DIR NEAR_SHARED(XNS)
       .
       .
       .
```

Here, XNS is local to FUNC().

near_shared variables and arrays can be initialized in Fortran DATA statements.

A similar example in C follows:

```
func() {
  static near_shared double xns[1000][1000];
   .
   .
   .
```

Here, xns is local to func(). Global declarations, and local declarations of the extern storage class, are also legal.

Both language examples allocate 8,000,000 bytes of virtual address space, which maps to 8,000,000 bytes of physical memory on hypernode 0. Any thread running on any other hypernode will experience interhypernode access latency when accessing this data. Therefore, this code only makes sense for single-hypernode systems, or for code that will always be executed on hypernode 0 only.

The true power of near_shared memory is realized only when it is resident on the hypernode that most frequently accesses it. The near_shared class is therefore most efficiently assigned dynamically at runtime, by the thread that will access the near_shared data object most often, on the hypernode on which that particular thread is running. Such dynamic allocation is discussed in the "Dynamic assignments" section later in this chapter.

far_shared

Because far_shared memory is physically distributed among all hypernodes and is best used when all hypernodes will be accessing it with similar frequency, static declarations make the most sense.

In Fortran, the far_shared memory class is assigned as shown in the following example:

```
      SUBROUTINE FUNC()
      REAL*8 XFS(1000,1000)
C$DIR FAR_SHARED(XFS)
      .
      .
      .
```

Here, XFS is local to FUNC().

far_shared variables and arrays can be initialized in Fortran DATA statements.

A similar C example follows:

```
void func() {
static far_shared double xfs[1000][1000];
.
.
.
```

Here, xfs is local to func(). Global declarations, and local declarations of the extern storage class, are also legal.

These declarations allocate 8,000,000 bytes of virtual address space, which is mapped to physical memory by 4-kbyte pages round-robin to each hypernode on a multihypernode subcomplex, beginning at hypernode 0. When all hypernodes are accessing the far_shared data with relatively equal frequency, this provides the best average access times. However, if your program only spawns threads on a subset of the hypernodes in a multihypernode subcomplex, far_shared memory is a poor choice because it will still be distributed across all hypernodes in the subcomplex. If you know that your program will only spawn threads on one hypernode, use the near_shared class to allocate memory on that hypernode; if your program will spawn threads on multiple hypernodes but not every hypernode in the subcomplex, use the block_shared class as described in the "Dynamic assignments" section.

block_shared

The block_shared memory class can only be allocated dynamically, as described in the following section.

Dynamic assignments

Dynamic memory class assignments are used with Convex Fortran ALLOCATABLE arrays and with the memory_class_malloc function in Convex C. The class assignments are located with variable declarations. As with static assignments, in Fortran, compiler directives are used to specify the desired memory class for a previously-declared data object; in C, the memory class is specified in the declaration using a type-qualifier extension. The allocation is done at the specific point in the program where the memory is needed, using the Fortran ALLOCATE statement or the C memory_class_malloc function. At this point, virtual memory is allocated, and the program's available virtual space is decreased by the amount of memory allocated. This virtual memory does not map to physical memory until the allocated data objects are referenced.

While any memory class can be dynamically assigned, the block_shared class can only be assigned dynamically, and the near_shared class is most useful when dynamically assigned.

Memory class pointers

All shared memory classes are accessible to all threads when dynamically allocated in serial code, regardless of the allocating thread, because all threads access these classes from the same physical memory using the same virtual addresses. However, in Fortran, if more than one thread in a parallel construct attempts to allocate a shared class array, only the last allocation will exist. This is because there can only be one (internal) pointer to the allocated array; by default, this pointer is of the same class as the allocated memory, and each allocating thread resets this shared pointer. This problem is overcome by adding class-specification directives of the following form:

C$DIR *class_name*_POINTER(*allocatable-namelist*)

where

class_name

is one of THREAD_PRIVATE, NODE_PRIVATE, NEAR_SHARED, or FAR_SHARED. *class_name* cannot contain BLOCK_SHARED because the BLOCK_SHARED class is specifically designed to hold array objects, and a pointer is a scalar object.

allocatable-namelist

is a comma-delimited list of arrays previously declared to be ALLOCATABLE in the same procedure. The private classes are included in *allocatable-namelist* because it is often only necessary to access a particular shared array from the particular thread or hypernode on which the array is being manipulated.

Allocatable private arrays are only accessible from the thread that allocates them; threads executing ALLOCATE statements in parallel will each be able to access the private array they allocate. Private arrays allocated outside of parallel constructs will only be accessible by thread 0.

The C memory_class_malloc function is very similar to standard malloc and has the following form:

memcls_ptr = memory_class_malloc(size_t *bytes*, int *class_name*);

where

memcls_ptr

is a previously-declared pointer to a variable of the desired memory class. Note that *memcls_ptr* need not be of the class indicated in *class_name*, allowing you to allocate one class of memory which is accessed by a pointer of a different class. This is analogous to using the C$DIR *class_name*_POINTER Fortran directive to allocate a pointer of one class to an array of a different class.

bytes

is the requested number of bytes.

class_name

is one of THREAD_PRIVATE_MEM, NODE_PRIVATE_MEM, NEAR_SHARED_MEM, FAR_SHARED_MEM, or BLOCK_SHARED_MEM; these symbolic constants are defined in spp_prog_model.h.

Not all combinations of pointer classes with data classes are supported, and not all make sense. Observe the following general rules:

- thread_private memory must be referenced by thread_private pointers.
- node_private memory should be referenced by near_shared or far_shared pointers.
- when shared data objects are referenced by private pointers, direct access to the object is restricted by the scope of the pointer. For example, if a thread_private pointer is used, access is restricted to the thread from which the pointer was allocated. If a node_private pointer is used, access is restricted to the threads on the hypernode from which the pointer was allocated. To access shared data using a thread_private pointer from a thread other than the thread that allocated the pointer, you must pass the pointer between threads. node_private pointers must similarly be passed between hypernodes when they are used.
- allocatable block_shared arrays are never explicitly assigned a pointer in Fortran.
- using shared-class pointers to point to shared-class memory provides pointer access to all threads, but, as with any shared-memory data, appropriate latency rules apply to the pointers.
- in Fortran, when one of the *class_name*_POINTER directives is not used, dynamically allocated memory is accessed via a pointer of the same class as the allocated data.

Table 5 shows proper and improper pointer/data class combinations.

Table 5 Pointer class/data class combinations

Pointer class	Data class				
	thread_private	node_private	near_shared	far_shared	block_shared
thread_private_pointer	OK	NR	OK	OK	OK
node_private_pointer	NR	OK	OK	OK	OK
near_shared_pointer	NR	OK	OK	OK	OK
far_shared_pointer	NR	OK	OK	OK	OK

In the table above, OK means the pointer/data class combination is acceptable; NR means the combination is not recommended.

While pointer classes are provided for private variables, they are typically only needed when allocating shared memory.

Default classes for dynamic memory

Using standard `malloc` in a shared-memory C program will allocate, by default, `far_shared` memory. (The default can be changed using the mpa(1) utility.) Memory allocated by either standard `malloc` or `memory_class_malloc` can be deallocated using the `free` function.

In Fortran, memory allocated using the `ALLOCATE` statement and not specifically assigned a class will be assigned the `far_shared` class by default. (This default can be changed using the mpa(1) utility.) Such memory is deallocated using the `DEALLOCATE` statement. Refer to Appendix C, "Fortran 90 compatibility," of the *Fortran Language Reference, 11th Edition*, for more information.

For applications that run on single-node subcomplexes, `far_shared` memory and `near_shared` memory are the same.

The stack can exist in only one physical space, so it is allocated in `near_shared` memory by default. This means that all local data (i.e., within a C function, all data not assigned a storage class of `static` or `extern`; in Fortran, all data not declared in `COMMON` blocks or `SAVE`d) is `near_shared` and resides on hypernode 0 by

default. This default can be changed to far_shared using the mpa(1) utility. For more information, refer to the mpa(1) man page.

Using each dynamic memory class is explained in detail in the sections that follow.

thread_private

Because only the allocating thread can access dynamically allocated thread_private memory, it should be allocated (and deallocated) inside thread-parallel constructs, where each thread can allocate its own copy.

Consider the following Fortran example:

```
      REAL*8 XTP(:)
C$DIR THREAD_PRIVATE(XTP)
      ALLOCATABLE(XTP)
      .
      .
      .
C$DIR LOOP_PARALLEL(THREADS), LOOP_PRIVATE(J)
      DO I = 1, NUM_THREADS()
C       THE FOLLOWING CODE MUST OCCUR INSIDE A
C       THREAD-PARALLEL CONSTRUCT
        ALLOCATE(XTP(N))
        DO J = 1, N
          . !COMPUTATIONS USING XTP
          .
          .
        ENDDO
        DEALLOCATE(XTP)
      ENDDO
```

This example assumes that the ALLOCATE statement is contained within a manually-created parallel loop or task. Then each parallel thread would get a private copy of the XTP array, of size N elements. Note the nested construct, in which the outer loop allocates and deallocates the thread_private array, and the inner loop uses it in computation. The outer loop calls the NUM_THREADS() intrinsic, which returns the number of threads on which the process is running, as described in Chapter 6, "Advanced shared-memory programming."

If this code was executed inside a node-parallel construct, each thread on each hypernode would allocate XTP.

If the ALLOCATE in this example was executed in a nonparallel section of code, only thread 0 would be able to reference XTP.

A similar C example follows:

```
static thread_private double *xtp;
.
.
.
#pragma _CNX loop_parallel(threads, ivar=i), loop_private(j)
for(i=0;i<num_threads();i++) {
/* the following statement must occur inside a
   parallel construct                            */
  xtp=(double *)memory_class_malloc(sizeof(double)*n,
                                    THREAD_PRIVATE_MEM);
  for(j=0;j<n;j++) {
    . /*computations using xtp*/
    .
    .
  }

  free(xtp);
}
```

This example allocates memory in the same manner as the Fortran example.

node_private

Recall that all threads access node_private memory via the same virtual addresses, and that these virtual addresses map to different physical addresses on each hypernode. Dynamic allocations of node_private memory should be executed in serial code. If you require physical memory on every hypernode and wish to reference it using the same virtual addresses from every hypernode, you can allocate the memory from serial code using a shared pointer. Allocation examples are given later in this section.

In order to access dynamically allocated node_private memory from node-parallel code, a shared pointer is needed. This is because when serial Fortran code running on hypernode 0 executes the ALLOCATE statement, the default node_private array pointer is assigned on that hypernode only. It is true that the virtual address maps to physical memory on every hypernode, but only the pointer of the assigning hypernode gets a value; the pointers on other hypernodes are unassigned. If hypernode 0 is the only hypernode requiring access to the array, all the threads running on it will be able to access the array via this node_private pointer. However, since the physical memory associated with the other hypernodes' pointers has not been assigned, attempting to use the values it contains will cause an error. Therefore, arrays dynamically allocated in serial code that

are to be accessed in parallel code must be accessed with explicitly-declared near_shared or far_shared pointers. Because the allocation takes place in serial code, there is no advantage to using one shared class over the other; both will be stored in physical memory on hypernode 0, causing pointer accesses from other hypernodes to take longer.

In Fortran, pointers default to the same memory class as the objects to which they point, so compiler directives are used to assign different memory classes to pointers. In C, the pointers must be explicitly declared separately, so the memory class assignment is handled in the pointer declaration.

Consider the following Fortran example:

```
      REAL*8 XNP(:)
C$DIR NODE_PRIVATE(XNP)
C$DIR FAR_SHARED_POINTER(XNP)
      ALLOCATABLE(XNP)
      .
      .
      .
C THE FOLLOWING CODE SHOULD OCCUR OUTSIDE
C ANY PARALLEL CONSTRUCT:
      ALLOCATE(XNP(1000))
```

This example assumes that the ALLOCATE statement is contained within a nonparallel section of code. Assuming this code is running on a 4-hypernode subcomplex, when the ALLOCATE statement executes, 8,000 bytes (1000 elements × 8 bytes per element) of virtual space are allocated for XNP. If the array is then accessed from a node-parallel construct, a unique physical copy of XNP will be created on any accessing hypernodes, and each accessing thread will access its hypernode's copy of XNP. If every hypernode on the subcomplex accesses the array, it will occupy a total of 32,000 bytes of physical memory.

A similar C example follows:

```
static far_shared double *xnp;
.
.
.
/* the following statement should occur outside any
   parallel construct:                              */
xnp = (double *)memory_class_malloc(sizeof(double)*1000,
                                NODE_PRIVATE_MEM);
```

This example allocates memory in the same manner as the Fortran example. Note that the pointer to xnp is explicitly assigned the far_shared class in its declaration.

The preceding examples allow you to use the same names to access physically unique `node_private` arrays on each hypernode from node-parallel code. This can effectively increase your program's virtual memory space, because the same amount of virtual space is used for the arrays no matter how many hypernodes hold physical copies or run code that accesses them.

Parallel allocations of `node_private` memory should normally be done outside parallel code. This memory should then be accessed via shared pointers. When memory is allocated in such a fashion, threads will only be able to access the data which is stored physically on their hypernode. To access `node_private` data that is stored on another hypernode, the data must be passed via a shared variable.

These allocations are useful when private work arrays are needed by each hypernode in a node-parallel construct.

Consider the following Fortran example:

```
      REAL*8 NODE_V(:)
      ALLOCATABLE(NODE_V)
C$DIR NODE_PRIVATE(NODE_V)
C$DIR FAR_SHARED_POINTER(NODE_V)
      .
      .
      .
      NN = NUM_NODES()
      JLMT = (JTOT/NN) + 1
      ALLOCATE(NODE_V(JLMT))
C$DIR LOOP_PARALLEL(NODES, CHUNK_SIZE = 1)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, M
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = 1, JLMT
         NODE_V(J) = PI*(RAD(J)**2)*L(I,J)
         P(I,J) = (N(I,J)*R*T(I,J))/NODE_V(J)
        ENDDO
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = 2, JLMT
          IF((NODE_V(J) .GT. NODE_V(J-1)).AND.
     >       (NODE_V(J) .GT. NODE_V(J+1))
     >         LMAX_V(I,J) = NODE_V(J)
        ENDDO
        .
        .
        .
      ENDDO
      DEALLOCATE(NODE_V)
```

Here, the NODE_V array is used privately by each hypernode in the computation of the array P and to find values for LMAX_V. NODE_V is not needed outside of the I loop. Note that the NUM_NODES() intrinsic (described in Chapter 6, "Advanced shared-memory programming") is used to determine the number of hypernodes on which the process is running.

An analogous C example follows:

```
static far_shared double *node_v;
.
.
.
nn = num_nodes();
jlmt = (jtot/nn) + 1;
node_v = (double*)memory_class_malloc(jlmt*sizeof(double),
                                    NODE_PRIVATE_MEM);
#pragma _CNX loop_parallel(nodes, chunk_size = 1, ivar = i)
#pragma _CNX loop_private(j)
for(i=0; i<m; i++) {
#pragma _CNX loop_parallel(threads)
  for(j=0; j<jlmt; j++) {
    node_v[j] = pi*rad[j]*rad[j]*l[i,j];
    p[i,j] = (n[i,j]*r*t[i,j])/node_v[j];
  }
#pragma _CNX loop_parallel(threads)
  for(j=1; j<jlmt; j++) {
    if((node_v[j] > node_v[j-1]) && (node_v[j] > node_v[j+1]))
      lmax_v[i,j] = node_v[j];
  }
}
free(node_v);
```

near_shared

To be most useful, near_shared memory must be dynamically allocated on the hypernode that will most heavily access it. near_shared memory should be allocated from within explicitly-defined node-parallel structures to ensure that it is allocated on the hypernodes that will use it most. Node-parallel structures are manually defined by the programmer using the loop_parallel(nodes), parallel(nodes), or begin_tasks(nodes) directives and pragmas, which are discussed in detail in Chapter 4, "Basic shared-memory programming."

Unconditionally allocating near_shared arrays in node-parallel structures is possible and is discussed later. However, to take full advantage of near_shared arrays, they should be allocated

conditionally, based on the allocating hypernode, within node-parallel code.

A Fortran example of this follows. For simplicity, we assume this example will always run on a 2-hypernode subcomplex.

```
      REAL*8 XNS(:), YNS(:)
      ALLOCATABLE(XNS, YNS)
C$DIR NEAR_SHARED(XNS, YNS)
      .
      .
      .
C THE FOLLOWING CODE MUST RUN NODE-PARALLEL
C$DIR LOOP_PARALLEL(NODES) !2-NODE SUBCMPLX
C$DIR LOOP_PRIVATE(NODE_ID)
      DO I = 1, NUM_NODES()! IS ASSUMED
        NODE_ID = MY_NODE()
        IF(NODE_ID .EQ. 0) THEN
          ALLOCATE(XNS(1000))
        ELSE
          ALLOCATE(YNS(1000))
        ENDIF
        .
        .
        .
      ENDDO
```

The `LOOP_PARALLEL(NODES)` directive is used to achieve node-parallelism; the `MY_NODE()` function returns the hypernode ID of the hypernode on which the current thread is executing. These are both discussed in more detail in Chapter 6, "Advanced shared-memory programming." This example allocates a `near_shared` `XNS(1000)` array for hypernode 0, and a `near_shared` `YNS(1000)` array for hypernode 1. When accessed, `XNS` is stored in hypernode-global physical memory on hypernode 0, and is therefore accessible from that hypernode with the least latency; similarly, `YNS` is physically stored on and quickly accessible from hypernode 1. The arrays' unique virtual addresses allow either to be accessed by the hypernode it does not occupy, and more latency is involved in such accesses. This example presumes that code that follows in the program conditionally manipulates these arrays on their respective hypernodes, but that some sharing of the array data between hypernodes also occurs.

A C example that allocates memory in the same fashion follows. Again we assume that this code will always run on a two-hypernode subcomplex.

```
static near_shared double *xns, *yns;int node_id;
.
.
.
/* the following code must run node-parallel on a 2-node
   subcomplex:                                          */
#pragma _CNX loop_parallel(nodes, ivar = i), loop_private(node_id)
for(i=0;i<num_nodes();i++) {
  node_id = my_node();
  if(node_id == 0)
    xns = (double *)memory_class_malloc(sizeof(double)*1000,
                                   NEAR_SHARED_MEM);
  else
    yns = (double *)memory_class_malloc(sizeof(double)*1000,
                                   NEAR_SHARED_MEM);
.
.
.
}
```

Note that in C, the pointer class is assigned with the type declaration, and the data class is assigned by the `memory_class_malloc` function.

Recall that `near_shared` data objects have a single virtual address used by all hypernodes, and, when accessed, a single physical address (on the allocating hypernode). In the above examples, the pointers used to access the `near_shared` arrays were of the same memory class as the arrays (by default in Fortran and by explicit typing in C). The allocations are not thread-parallel, so a single thread on each hypernode will allocate its respective array. If at some point later in the program the code goes thread-parallel, all threads on a given hypernode will be able to access the arrays via their `near_shared` pointers.

However, if you wish to allocate `near_shared` memory from within a thread-parallel structure, the `near_shared` pointer can present a problem when the `near_shared` data space is actually allocated: all threads will be allocating the space using the same shared pointer, so each thread's `ALLOCATE` will reset the pointer. In C the pointer class is directly controllable, but in Fortran the *memory_class*_`POINTER` directive must be used to assign a pointer of a different class to the array.

Table 5 covers acceptable pointer classes to use for `near_shared` data. While a certain combination may be allowed, it may not

make sense for the task at hand. In the following Fortran example, the pointer to ZNS is assigned the THREAD_PRIVATE memory class, because ZNS is being allocated in a thread-parallel structure. This causes each thread to allocate its own near_shared copy of ZNS on its hypernode.

```
      REAL*8 ZNS(:)
      ALLOCATABLE(ZNS)
C$DIR NEAR_SHARED(ZNS)
C$DIR THREAD_PRIVATE_POINTER(ZNS)
      .
      .
      .
C THE FOLLOWING CODE MUST RUN THREAD-PARALLEL
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, NUM_THREADS
        ALLOCATE(ZNS(1000))
        .
        .
        .
      ENDDO
```

Here, thread-parallelism is achieved using the LOOP_PARALLEL(THREADS) directive, which is discussed further in Chapter 4, "Basic shared-memory programming." The hypernode on which the ALLOCATE statement executes allocates a near_shared copy of ZNS for each thread. Each thread running on that hypernode can then reference its copy of ZNS via the thread_private pointers provided by the THREAD_PRIVATE_POINTER directive on ZNS. If the thread-parallel section of code shown is also running node-parallel, each hypernode running it will allocate as many near_shared ZNS arrays as it has threads. For example, if 8 threads are running on each of 2 hypernodes, 16 physical and virtual copies of ZNS will be created. However, because of the thread-private pointers, these near_shared copies will lose their direct-accessibility; accesses by any thread other than the allocating thread is only possible in C and will require sharing of the thread_private pointers.

A similar C example follows:

```
static thread_private double *zns;
.
.
.
/* the following code must run thread-parallel   */
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0;i<num_threads();i++)
  zns = (double *)memory_class_malloc(sizeof(double)*1000,
                                      NEAR_SHARED_MEM);
.
.
.
```

This example allocates memory in the same manner as the Fortran example.

When only one parallel hypernode in a group of hypernode-parallel tasks needs a private array, the `near_shared` class can be allocated dynamically from within the task in question, a `near_shared` pointer can be used to provide low-latency access to the array.

The following Fortran code shows a parallel allocation for a single parallel hypernode:

```
      REAL*8 NODE_SCRATCH(:)
      ALLOCATABLE(NODE_SCRATCH)
C$DIR NEAR_SHARED(NODE_SCRATCH)
      .
      .
      .
C$DIR BEGIN_TASKS(NODES), TASK_PRIVATE(I)
      DO I = 1, N
        A(I) = B(I) + C(I)
      ENDDO
C$DIR NEXT_TASK
      CALL TSUB(X,Y)
      .
      .
      .
C$DIR NEXT_TASK
      ALLOCATE(NODE_SCRATCH(1000))
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, 1000
        NODE_SCRATCH(I) = Z(I)
      ENDDO
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, 999
        Z(I) = NODE_SCRATCH(I+1)
        .
        .
        .
      ENDDO
      DEALLOCATE(NODE_SCRATCH)
C$DIR END_TASKS
      .
      .
      .
```

Here, the final task in the list allocates the temporary near_shared work array NODE_SCRATCH, uses it, and deallocates it. The compiler will thread-parallelize both loops within this task because of the LOOP_PARALLEL directives on them, and they will both be able to access the NODE_SCRATCH array with minimal latency. Because this array is near_shared, it can also be accessed by any other hypernode, though that does not happen in this example. Using LOOP_PARALLEL to manually parallelize loops is further discussed in Chapter 6, "Advanced shared-memory programming."

An analogous C example follows:

```
static near_shared double *node_scratch;
.
.
.
#pragma _CNX begin_tasks(nodes), task_private(i)
for (i=0; i<n; i++) {
  a[i] = b[i] + c[i];
}
#pragma _CNX next_task
tsub(x,y);
.
.
.
#pragma _CNX next_task
node_scratch = (double *)memory_class_malloc(1000*sizeof(double),
                                             NEAR_SHARED_MEM);
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0; i<1000; i++) {
  node_scratch[i] = z[i];
}
#pragma _CNX loop_parallel(threads, ivar = i)
for(i=0; i<999; i++) {
  z[i] = node_scratch[i+1];
   .
   .
   .
}
free(node_scratch);
#pragma _CNX end_tasks
.
.
.
```

far_shared

Because far_shared memory is distributed across all hypernodes in the subcomplex, it is best dynamically allocated in nonparallel structures. Allocating far_shared memory in a thread- or node-parallel structure would create multiple copies of the requested far_shared array, one for each allocating thread or hypernode in the structure. This replication increases the amount of memory, both physical and virtual, used by the process, and is of questionable utility.

Consider the following Fortran example:

```
      REAL*8 XFS(:)
      ALLOCATABLE(XFS)
C$DIR FAR_SHARED(XFS)
      .
      .
      .
C THE FOLLOWING CODE SHOULD BE EXECUTED IN
C A NONPARALLEL STRUCTURE:
      ALLOCATE(XFS(10000))
```

Because the `far_shared` data in this example is allocated outside of parallel code, the default `far_shared` pointer is suitable; the array is allocated once, and there is no danger of parallel allocations resetting the pointer. When the `ALLOCATE` statement executes, 80,000 bytes of virtual space is allocated. When the array is accessed, this maps to 80,000 bytes of physical memory, which is distributed, in an approximately round-robin manner, by 4-kbyte pages to each hypernode in the subcomplex.

A similar C example follows:

```
static far_shared double *xfs;
.
.
.
/* the following code should run in a nonparallel
   structure:                                           */
xfs = (double *)memory_class_malloc(sizeof(double)*10000,
                                    FAR_SHARED_MEM);
```

This example allocates memory in the same manner as the Fortran example.

Keep in mind that even dynamically allocated `far_shared` memory is distributed across all hypernodes in the subcomplex, not just across the hypernodes on which your program can spawn threads. Therefore, if your program is constrained to a subset of the hypernodes available in its subcomplex, you should use the `block_shared` or `near_shared` memory classes to avoid placing data on unused hypernodes.

`far_shared` is the default memory class for all data not otherwise assigned a class.

block_shared

block_shared memory is specifically provided for dynamic allocation by a process running on multiple hypernodes. It is ideal for array-manipulating loops that will parallelize across hypernodes, with each hypernode computing chunks of contiguous elements of the arrays. Using block_shared memory for such arrays distributes the chunks across the hypernodes on which the program is running; work can then be distributed so that each hypernode accesses the elements that reside on it most frequently. If necessary, hypernodes can still directly access each other's block_shared data.

As with far_shared data, this block_shared data allocation should take place outside of parallel structures; in this case, the compiler will automatically distribute the data evenly across the hypernodes of the subcomplex. Allocating block_shared memory from parallel structures will cause multiple copies (one per parallel thread) of the block_shared data to be created, a situation that wastes memory and is of questionable utility.

Consider the following Fortran example:

```
      REAL*8 XBS(:), YBS(:)
      ALLOCATABLE(XBS, YBS)
C$DIR BLOCK_SHARED(XBS,YBS)
      .
      .
      .
C THE FOLLOWING CODE SHOULD BE EXECUTED IN
C A NONPARALLEL STRUCTURE:
      ALLOCATE(XBS(10240), YBS(7680))
```

Here, 81,920 bytes of virtual space is requested for XBS, and 61,440 bytes is requested for YBS. Assuming this code is running on a 4-hypernode subcomplex, these block_shared arrays will have their physical pages divided equally, in contiguous chunks, among the 4 hypernodes. Recall that the page size is 4,096 bytes, so XBS occupies 20 pages (81,920/4,096 = 20). The number of hypernodes, 4, is an integral divisor of 20, giving 5 pages per hypernode. So the first 5 pages of XBS are physically mapped to hypernode 0 (yet still accessible via their virtual addresses from any other hypernode), the second 5 pages go to hypernode 1, the third 5 pages to hypernode 2, and the last 5 pages to hypernode 3.

The 61,440 bytes of YBS, on the other hand, occupy 15 pages. 4 does not integrally divide 15, so the compiler automatically increases the size of YBS just enough to allow the number of hypernodes to integrally divide its pages. YBS becomes an 8,192 element array; it now occupies 65,536 bytes or 16 pages of memory. These 16 pages are divided up so that the first 4 pages

map to hypernode 0 and so on, with the last 4 mapping to hypernode 3.

Any hypernode-parallel code that follows the above allocation should be written such that the portion running on a particular hypernode accesses the array elements resident on that hypernode. For example, the code running on hypernode 2 should make most frequent use of `XBS(5121:7680)` and `YBS(3841:5760)`. The "Accessing hypernode-local block_shared elements" section, which follows, discusses how to determine which elements of a `block_shared` array are on a given hypernode.

A similar C example follows:

```
static near_shared double *xbs, *ybs;
.
.
.
/* the following code should be run in a nonparallel
   section of code:                                     */
xbs = (double *)memory_class_malloc(sizeof(double)*10240,
                                    BLOCK_SHARED_MEM);
ybs = (double*)memory_class_malloc(sizeof(double)*7680,
                                    BLOCK_SHARED_MEM);
```

This example allocates and distributes `xbs` and `ybs` as `block_shared` arrays exactly as the Fortran example did, including the automatic resizing of `ybs`. `near_shared` pointers are used to access the arrays.

When allocated, `block_shared` arrays are distributed across all hypernodes available to your program; you cannot constrain the number of hypernodes that the arrays occupy. This makes the `block_shared` class especially suitable for programs whose data sets scale with the number of available hypernodes. If there is any question as to whether your `block_shared` arrays will occupy enough pages to efficiently use the number of hypernodes available to your program, you should roughly compute the number of pages the array in question is likely to occupy. If this number is not at least equal to the number of hypernodes that will typically be available to the program, you can still use the `block_shared` class, but your program might waste some memory by expanding the array to occupy at least one page per hypernode.

While you cannot constrain the allocation of `block_shared` memory from within your program, you can constrain the number of hypernodes on which your program runs by using the mpa(1) utility or the `max_threads loop_parallel` attribute. `block_shared` memory is only allocated on the hypernodes on

which your program can spawn threads, so if your program is constrained in this way, block_shared memory is preferable to far_shared memory, which is distributed across all hypernodes in the subcomplex.

Because block_shared data elements each have a unique virtual and physical address (there is no virtual address replication as with the node_private class), their size is limited by the 4-Gbyte virtual address space limit. Keep this in mind if your data set size scales with the number of available hypernodes. If your program needs to surpass this limit, it can do so using the node_private class as described in the section "Advanced shared-memory example" on page 246.

Accessing hypernode-local block_shared elements

Determining the range of block_shared array element indexes located on a given hypernode is easily computable given the page size, number of hypernodes, element size, and number of elements. The following Fortran function takes these parameters along with the current hypernode number as arguments and returns the base index for the elements on the current hypernode:

```
      INTEGER FUNCTION MIN_NODE_ELT(PGSZ,NN,ELTSZ,NELTS,CUR_NODE)
      INTEGER PGSZ, NN, ELTSZ, NELTS, CUR_NODE
      INTEGER NUM_PGS, PGS_PER_NODE
      NUM_PGS = 1 + (NELTS*ELTSZ - 1)/PGSZ
      PGS_PER_NODE = 1 + (NUM_PGS - 1)/NN
C     ADJUST MIN_NODE_ELT BY +1 TO COMPENSATE FOR ARRAY INDEXING
C     FROM 1:
      MIN_NODE_ELT = (CUR_NODE*PGS_PER_NODE*PGSZ/ELTSZ) + 1
      RETURN
      END
```

This Fortran example assumes the array is indexed from 1.

The analogous C function, which assumes indexing begins at 0, is shown below.

```
int min_node_elt(int pgsz,int nn,int eltsz,int nelts,int cur_node)
{
  int num_pgs, pgs_per_node;
  num_pgs = 1 + (nelts*eltsz - 1)/pgsz;
  pgs_per_node = 1 + (num_pgs - 1)/nn;
  return (cur_node*pgs_per_node*pgsz/eltsz);
}
```

The following Fortran example shows one way in which MIN_NODE_ELT can be used in a hypernode-parallel loop so that each hypernode accesses only its local array elements. This example assumes that the number of pages occupied by XBS is

large enough to efficiently exploit all available hypernodes, and that the 4-Gbyte virtual address limit is not surpassed.

```
      REAL*8 XBS(:)
      ALLOCATABLE(XBS)
C$DIR BLOCK_SHARED(XBS)
      .
      .
      .
C **NO PARALLELISM**
      ALLOCATE(XBS(ARSZ))
      NN = NUM_NODES()
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J, MN, MIN_ELT, MAX_ELT)
      DO I = 1, NN          ! DO ON EACH NODE:
        MN = MY_NODE()      ! GET CURRENT NODE NUMBER
C       GET MIN AND MAX ELEMENT NUMBERS FOR CURRENT NODE:
        MIN_ELT = MIN_NODE_ELT(PGSZ,NN,8,ARSZ,MN)
        MAX_ELT = MIN_NODE_ELT(PGSZ,NN,8,ARSZ,MN+1) - 1
C       GO THREAD PARALLEL ON CURRENT NODE:
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = MIN_ELT, MAX_ELT ! LOOP OVER LOCAL ELEMENTS
          XBS(J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the array XBS is allocated in serial code. When the program goes node-parallel, each iteration calls MY_NODE to get its hypernode ID (which ranges from 0..NUM_NODES), then uses this ID in the following calls to MIN_NODE_ELT to determine the minimum and maximum indexes of the hypernode-local elements of XBS. Each hypernode then computes its elements of XBS in a thread-parallel loop that iterates over only the resident elements of XBS.

The analogous C code follows:

```
static near_shared double *xbs;
.
.
.
/********************** no parallelism **********************/
xbs = (double *)memory_class_malloc(sizeof(double)*arsz,
                                    BLOCK_SHARED_MEM);
nn = num_nodes();
#pragma _CNX loop_parallel(nodes, ivar = i)
#pragma _CNX loop_private(j, mn, min_elt, max_elt)
for(i=0; i<nn; i++) { /* do on each node:                    */
  mn = my_node();     /* get current node number             */
  /* get min and max element numbers for current node:       */
  min_elt = min_node_elt(pgsz,nn,sizeof(double),arsz,mn);
  max_elt = min_node_elt(pgsz,nn,sizeof(double),arsz,mn+1) - 1;
  /* go thread parallel on current node:                     */
#pragma _CNX loop_parallel(threads, ivar = j)
  for(j=min_elt; j<=max_elt; j++) { /* loop over local elts */
    xbs[j] = ...
    .
    .
    .
  }
}
```

Another easy way to access hypernode-local elements is to add a dimension of size 1..num_nodes() to your block_shared array when you allocate it.

The following Fortran example shows such an allocation:

```
      REAL*8 ABS(::)
      ALLOCATABLE(ABS)
C$DIR BLOCK_SHARED(ABS)
      .
      .
      .
C **NO PARALLELISM**
      ALLOCATE(ABS(N,NUM_NODES()))
      .
      .
      .
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, NUM_NODES()
C$DIR   LOOP_PARALLEL(THREADS)
        DO J = 1, N
          ABS(J,I) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

Here, N is chosen so that N*NUM_NODES() is equal to or greater than the total number of ABS elements required. We assume that the original problem (before it is rewritten for parallelization) does not require a two-dimensional array, and that the second dimension is provided only to allow each parallel hypernode to easily index its local elements. Inside the J loop, the I index into ABS ensures that each parallel hypernode accesses its local elements automatically.

Advanced shared-memory programming

6

Most of the manual parallelization techniques discussed in Chapter 4, "Basic shared-memory programming," allow you to take advantage of the compilers' automatic dependence checking and data privatization. The examples that used the LOOP_PRIVATE and TASK_PRIVATE directives and pragmas are exceptions to this; in these cases, manual privatization was required, but it was done on a loop-by loop basis. Only the simplest data dependences were handled in Chapter 4.

This chapter is concerned with manual parallelizations that use the program-wide memory classes discussed in Chapter 5, "Memory classes," and that handle multiple and ordered data dependences.

Before we can discuss specific examples of such parallelization, however, we must introduce the remaining underlying concepts and available functions.

Parallel information functions

Several SPP Series intrinsics are available to provide information regarding the parallelism or potential parallelism of your program. These are all integer functions, available in both 4- and 8-byte lengths, and can appear in executable statements anywhere an integer expression is legal. The 8-byte versions, which are suffixed with `_8`, are typically only used in Fortran programs in which the default data lengths have been changed using the `-cfc`, `-p8` or similar compiler options. When default integer lengths are modified via compiler options in Fortran, the correct intrinsic is automatically chosen regardless of which is specified. These versions expect 8-byte input arguments and return 8-byte values.

Note

All C code examples presented in this chapter assume that the line

```
#include <spp_prog_model.h>
```

appears above the C code presented. This header file contains the necessary type and function definitions.

The subsections that follow describe these functions.

Number of processors

These functions return the total number of processors on which the process has initiated threads. These threads are not necessarily active.

In Fortran, these functions have the following forms:

```
INTEGER NUM_PROCS()
INTEGER*8 NUM_PROCS_8()
```

In C, they have the following forms:

```
int num_procs(void);
long long num_procs_8(void);
```

`num_procs` can be used to dimension automatic and adjustable arrays in Fortran, and may be used in C or Fortran to dynamically specify array dimensions and allocate storage.

Number of threads

These functions return the total number of threads the process creates at initiation, regardless of how many hypernodes the threads occupy, and regardless of how many are idle or active. They are typically used to manually define thread-parallel loops which may span hypernodes.

In Fortran, these functions have the following forms:

```
INTEGER NUM_THREADS()
INTEGER*8 NUM_THREADS_8()
```

In C, they have the following forms:

```
int num_threads(void);
long long num_threads_8(void);
```

The return value will only differ from `num_procs` if threads are oversubscribed.

Number of hypernodes

These functions return the number of hypernodes on which the process is running. They can be used to dimension automatic and adjustable arrays in Fortran, and can be used in both C and Fortran to dynamically specify array dimensions and allocate storage.

In Fortran, these functions have the following forms:

```
INTEGER NUM_NODES()
INTEGER*8 NUM_NODES_8()
```

In C, they have the following forms:

```
int num_nodes(void);
long long num_nodes_8(void);
```

Number of threads on current hypernode

These functions return the number of the calling process's threads running on the hypernode from which the function is called. This number can vary from one hypernode to another depending on subcomplex configurations, usage of manual parallelization directives, and the number of processors installed on each hypernode.

In Fortran, these functions have the following forms:

```
INTEGER NUM_NODE_THREADS()
INTEGER*8 NUM_NODE_THREADS_8()
```

In C, they have the following forms:

```
int num_node_threads(void);
long long num_node_threads_8(void);
```

Thread ID

When called from parallel code these functions return the spawn thread ID of the calling thread, in the range 0..*nst*-1, where *nst* is the number of threads in the current spawn context (the number of threads spawned by the last parallel construct). Use them when you wish to direct specific tasks to specific threads inside parallel constructs.

In Fortran, these functions have the following forms:

```
INTEGER MY_THREAD()
INTEGER*8 MY_THREAD_8()
```

In C, they have the following forms:

```
int my_thread(void);
long long my_thread_8(void);
```

When called from code that is not running parallel due to compiler parallelism, `cps_ppcall()`, or `cps_ppcalln()` (for example, serial code or code that is parallel due to asymmetric CPSlib calls), these functions return 0.

Hypernode ID

These functions return the logical hypernode ID of the hypernode on which the calling thread is running, in the range 0..num_nodes()-1. Use them when you wish to direct specific tasks to specific hypernodes inside parallel constructs.

In Fortran, these functions have the following forms:

```
INTEGER MY_NODE()
INTEGER*8 MY_NODE_8()
```

In C, they have the following forms:

```
int my_node(void);
long long my_node_8(void);
```

Logical hypernode IDs range from 0..*n*-1, where *n* is the number of available hypernodes in the subcomplex. Logical IDs are assigned in the order in which your program occupies the subcomplex. The hypernode that your program's thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers. Because SPP-UX starts a program on the least-loaded hypernode, mapping of logical hypernode IDs to physical hypernodes can differ between programs due to load balancing; thus two programs running on the same subcomplex are unlikely to address identical hypernodes with identical logical IDs.

Logical hypernode IDs have no correlation to physical hypernode IDs, which are unique for each hypernode at the machine level.

Level of parallelism

These functions return a value representing the level of parallelism of the calling process.

In Fortran, these functions have the following forms:

```
INTEGER LEVEL_OF_PARALLELISM()
INTEGER*8 LEVEL_OF_PARALLELISM_8()
```

In C, they have the following forms:

```
int level_of_parallelism(void);
long long level_of_parallelism_8(void);
```

The return value is one or a sum of the values shown in Table 6. In C, these values are `#defined` as symbolic constants in `spp_prog_model.h`.

Table 6 Levels of parallelism

Function return value	C symbolic constant name	Meaning
0	CPS_PL_NONE	Not parallel
1	CPS_PL_PARALLEL	Asymmetric thread active
2	CPS_PL_NODE	Node-parallelism
4	CPS_PL_NTHREAD	Thread-parallelism within a hypernode
8	CPS_PL_THREAD	Single-dimensional thread-parallelism

As an example of how these can be summed, assume the return value is 6. This means the process is two-dimensionally parallel; it first went parallel across hypernodes, and within the current hypernode it went parallel again on the threads of the hypernode. This differs from a return value of 8, which means the process went one-dimensionally thread-parallel, and occupies all available threads on all available hypernodes with no nested parallelism.

The valid sum values are: 3,5,6,7, and 9.

A return value of 1, or a sum including 1, means an asymmetric thread is active in the calling program. Asymmetric parallelism is currently only supported by the Compiler Parallel Support Library. Refer to Appendix D, "Compiler Parallel Support Library," for more information.

Stack memory type

These functions return a value representing the memory class that the current thread stack is allocated from. The thread stack holds all the procedure-local arrays and variables not manually assigned a class. The thread stack is created in near_shared memory by default, but this can be changed via the mpa(1) utility.

In Fortran, these functions have the following forms:

```
INTEGER MEMORY_TYPE_OF_STACK()
INTEGER*8 MEMORY_TYPE_OF_STACK_8()
```

In C, they have the following forms:

```
int memory_type_of_stack(void);
long long memory_type_of_stack_8(void);
```

These functions return one of the values described in Table 7.

Table 7 Stack type return values

Function return value	C symbolic constant name	Stack memory type
4	FAR_SHARED_MEM	far_shared
3	NEAR_SHARED_MEM	near_shared
2	NODE_PRIVATE_MEM	node_private

Thread IDs and nested parallelism

As discussed in Chapter 4, "Basic shared-memory programming," you can manually parallelize nested loops and tasks to exploit up to two dimensions of parallelism. If you choose to do this, the first dimension must be node-parallel and the second must be thread-parallel. If thread-parallelism is exploited first, no dimensions are left; it is a programming error to attempt to spawn node-parallelism from within a thread-parallel construct. However, single-dimensional thread-parallel code can exploit all the threads on a subcomplex, even if they span hypernodes.

If you attempt to spawn thread-parallelism from within a thread-parallel construct, the compiler will ignore your directives, and your inner parallel construct will simply run serially.

Thread ID assignments

Chapter 3, "Compiler optimizations," discusses how programs are initiated as a collection of threads, one per available processor, and how all but thread 0 are idle until parallelism is encountered. We will now discuss the details of how threads are spawned and assigned IDs.

When a process begins, the threads created to run it have unique *kernel* thread IDs. Thread 0, which runs all the serial code in the program, has kernel thread ID 0; the rest of the threads have unique but unspecified kernel thread IDs at this point. The `num_threads()` intrinsic will return the number of threads created, regardless of how many are active when it is called.

When thread 0 encounters parallelism, it *spawns* some or all of the threads created at program start. This means it causes these threads to go from idle to active, at which point they begin working on their share of the parallel code. All available threads are spawned by default, but this can be changed using various compiler directives.

If the parallel structure is thread-parallel, then `num_threads()` threads will be spawned, subject to user-specified limits. At this point, kernel thread 0 becomes *spawn* thread 0, and the spawned threads are assigned spawn thread IDs ranging from 0..`num_threads()-1` (this range begins at what used to be kernel thread 0). If you manually limit the number of spawned threads, these IDs will range from 0 to one less than your limit. If you attempt to spawn thread-parallelism within an already thread-parallel structure, the thread attempting to spawn will acquire spawn thread ID 0. If all threads attempt to spawn thread-parallelism in this manner, they will all become spawn thread 0, each in a unique context.

If the parallel structure is node-parallel, then `num_nodes()` threads will be spawned, one per available hypernode, subject to user-specified limits. Again, kernel thread 0 becomes spawn thread 0, and in this case, the spawn thread IDs range from 0..`num_nodes()-1`, subject to user limits as described above.

If thread-parallelism is then encountered within this node-parallelism, `num_node_threads()` threads will be spawned on the hypernode or hypernodes encountering the thread-parallelism. These spawned threads will have spawn thread IDs, which are specific to the hypernode they are running on, ranging from 0..`num_node_threads()-1`, with spawn thread ID 0 belonging to the initial thread that executes the spawn. `num_node_threads()` may return a different value on each hypernode when called from node-parallel code.

Note that, with nested parallelism, a node-parallel thread that encounters a thread-parallel construct becomes spawn thread 0 on that hypernode regardless of its previous spawn thread ID. When this thread exits the thread-parallel construct, it returns to its previous spawn thread ID. The `my_thread()` intrinsic function returns the caller's spawn thread ID, which depends on the level of parallelism.

Synchronization tools

The compiler cannot automatically parallelize loops containing dependences, but a rich set of directives, pragmas and data types is available to help you manually parallelize such loops by synchronizing (and, if necessary, ordering) access to the code containing the dependence. These directives can also be used to synchronize dependences in parallel tasks. They allow you to efficiently exploit parallelism in structures that would otherwise be unparallelizable.

Gates and barriers

Gates allow you to restrict execution of a block of code to a single thread. They can be allocated, locked, unlocked and deallocated via the functions described in the "Synchronization functions" section, or they can be used with the ordered or critical section directives, which automate the locking and unlocking functions.

Barriers block further execution until all executing threads reach the barrier.

Gates and barriers use dynamically allocatable variables, declared using compiler directives in Fortran and using data type statements in C. They may be initialized and referenced only by

passing them as arguments to the functions discussed in the following "Synchronization functions" section.

In C, gates and barriers are declared using the `gate_t`, `gate8_t`, `barrier_t` and `barrier8_t` data type statements, which have the following forms:

```
gate_t namelist
gate8_t namelist
barrier_t namelist
barrier8_t namelist
```

where *namelist* is a comma-delimited list of one or more gate or barrier names, as appropriate. `gate8_t` and `barrier8_t` are used to declare 8-byte gate and barrier variables. The other declarations declare default-size variables.

In C, gates and barriers should appear only in definition and declaration statements, and as formal and actual arguments.

In Fortran, gates and barriers are declared using the `GATE` and `BARRIER` compiler directives, which have the following forms:

```
C$DIR GATE(namelist)
C$DIR BARRIER(namelist)
```

where *namelist* is a comma-delimited list of one or more gate or barrier names, as appropriate. These declare variables of the appropriate size; separate 4- and 8-byte versions are not needed in Fortran. No other type declarations are necessary for these variables; the compiler directives alone are sufficient.

In Fortran, gates and barriers can only appear:

- In `COMMON` statements (statement must precede `GATE` directive/`BARRIER` directive)
- In `DIMENSION` statements (statement must precede `GATE` directive/`BARRIER` directive)
- In preceding type statements
- As dummy arguments
- As actual arguments

Gate and barrier types override other types declared using the same names prior to the gate/barrier declaration. Once a variable is declared as a gate or barrier, it cannot be redeclared as another type. Gates and barriers cannot be `EQUIVALENCE`d. If you place gates or barriers in `COMMON`, the `COMMON` block declaration must precede the `GATE` directive/`BARRIER` directive, and the `COMMON` block should contain only gates or only barriers. Arrays of gates or barriers must be dimensioned using `DIMENSION` statements. The `DIMENSION` statement must precede the `GATE` directive/`BARRIER` directive.

Synchronization functions

The C and Fortran allocation, deallocation, lock and unlock functions provided for use with gates and barriers are listed here. 4- and 8-byte versions are provided; the 8-byte Fortran functions are primarily for use with compiler options that change the default data size to 8 bytes (e.g., `-cfc`, `-p8`, `-pd8`). You must be consistent in your choice of versions—memory allocated using an 8-byte function must be deallocated using an 8-byte function.

Examples of using these functions are presented and explained in the "Synchronizing code" section, which follows.

Allocation functions

These functions allocate memory for a gate or barrier. When first allocated, gate variables are unlocked.

The Fortran gate and barrier allocation functions have the following declarations:

```
INTEGER FUNCTION ALLOC_GATE(gate)
INTEGER*8 FUNCTION ALLOC_GATE_8(gate)
INTEGER FUNCTION ALLOC_BARRIER(barrier)
INTEGER*8 FUNCTION ALLOC_BARRIER_8(barrier)
```

Where *gate* and *barrier* are the gate or barrier variables, as appropriate. These variables must be declared as described in the "Gates and barriers" section of this chapter.

In C, the functions have the following declarations:

```
int alloc_gate(gate_t *gate_p);
long long alloc_gate_8(gate8_t *gate_p);
int alloc_barrier(barrier_t *barrier_p);
long long alloc_barrier_8(barrier8_t *barrier_p);
```

Where *gate_p* and *barrier_p* are pointers of the indicated type, which have been previously declared as described in the "Gates and barriers" section of this chapter.

Deallocation functions

These functions free the memory assigned to the specified gate or barrier variable.

The Fortran gate and barrier deallocation functions have the following declarations:

```
INTEGER FUNCTION FREE_GATE(gate)
INTEGER*8 FUNCTION FREE_GATE_8(gate)
INTEGER FUNCTION FREE_BARRIER(barrier)
INTEGER*8 FUNCTION FREE_BARRIER_8(barrier)
```

Where *gate* and *barrier* are the previously-declared gate or barrier variables, as appropriate.

In C, the functions have the following declarations:

```
int free_gate(gate_t *gate_p);
long long free_gate_8(gate8_t *gate_p);
int free_barrier(barrier_t *barrier_p);
long long free_barrier_8(barrier8_t *barrier_p);
```

Where *gate_p* and *barrier_p* are previously-declared pointers of the indicated type.

Always free gates and barriers when you are done using them.

Locking functions

These functions acquire a gate for exclusive access. If the gate cannot be immediately acquired, the calling thread waits for it. The conditional locking functions, which are prefixed with `COND_` or `cond_`, acquire a gate if doing so does not require a wait. If the gate is acquired, the functions return 0; if not, they return -1.

The Fortran gate locking functions have the following declarations:

```
INTEGER FUNCTION LOCK_GATE(gate)
INTEGER*8 FUNCTION LOCK_GATE_8(gate)
INTEGER FUNCTION COND_LOCK_GATE(gate)
INTEGER*8 FUNCTION COND_LOCK_GATE_8(gate)
```

Where *gate* is a gate variable.

In C, the functions have the following declarations:

```
int lock_gate(gate_t *gate_p);
long long lock_gate_8(gate8_t *gate_p);
int cond_lock_gate(gate_t *gate_p);
long long cond_lock_gate_8(gate8_t *gate_p);
```

Where *gate_p* is a pointer of the indicated type.

Unlocking functions

These functions release a gate from exclusive access. Gates are typically released by the thread that locks them, unless a gate was locked by thread 0 in serial code, in which case it might be unlocked by a single different thread in a parallel construct.

The Fortran gate unlocking functions have the following declarations:

```
INTEGER FUNCTION UNLOCK_GATE(gate)
INTEGER*8 FUNCTION UNLOCK_GATE_8(gate)
```

Where *gate* is a gate variable.

In C, the functions have the following declarations:

```
int unlock_gate(gate_t *gate_p);
long long unlock_gate_8(gate8_t *gate_p);
```

Where *gate_p* is a pointer of the indicated type.

Wait functions

These functions use a barrier to cause the calling thread to wait until the specified number of threads call the function, at which point all threads are released from the function simultaneously.

The Fortran barrier wait functions have the following declarations:

```
INTEGER FUNCTION WAIT_BARRIER(barrier,nthr)
INTEGER*8 FUNCTION WAIT_BARRIER_8(barrier,nthr)
```

Where barrier is a *barrier* variable of the indicated type and *nthr* is the number of threads calling the routine.

In C, the functions have the following declarations:

```
int wait_barrier(barrier_t *barrier_p,const int *nthr);
long long wait_barrier_8(barrier8_t *barrier_p,const long long *nthr);
```

Where *barrier_p* is a pointer of the indicated type and *nthr* is a pointer referencing the number of threads calling the routine.

A barrier variable can be used in multiple calls to the `wait` function, as long as the programmer ensures that two such barriers are not simultaneously active. It is also the programmer's responsibility to ensure that *nthr* reflects the correct number of threads.

`sync_routine` directive and pragma

Among the most basic optimizations performed by the Convex SPP Series compiler is code motion, which is described in Chapter 3, "Compiler optimizations." This optimization can move some code across routine calls. If the routine call is to a synchronization function that the compiler cannot identify as such, and the code moved must execute on a certain side of it, this movement can cause wrong answers.

The compiler is aware of all synchronization functions presented in this chapter and in Chapter 4, "Basic shared-memory programming," and will not move code across them when they appear directly in code. However, if the synchronization function is hidden in a user-defined routine, the compiler has no way of knowing about it and may move code across it.

Anytime you call synchronization functions indirectly via your own routines (or directly via CPSlib), you must identify your routines with a `sync_routine` directive or pragma.

In Fortran, `sync_routine` has the following form:

```
C$DIR SPP SYNC_ROUTINE (routinelist)
```

In C, it has the following form:

```
#pragma _CNX SPP sync_routine (routinelist)
```

Where *routinelist* is a comma-delimited list of synchronization routines.

`sync_routine` is only effective for the listed routines that lexically follow it in the routine in which it appears.

Consider the following Fortran example:

```
      INTEGER MY_LOCK, MY_UNLOCK
C$DIR GATE(LOCK)C$DIR SYNC_ROUTINE(MY_LOCK, MY_UNLOCK)
      .
      .
      .
      LCK = ALLOC_GATE(LOCK)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        LCK = MY_LOCK(LOCK)
        .
        .
        .
        SUM = SUM + A(I)
        LCK = MY_UNLOCK(LOCK)
      ENDDO
      .
      .
      .
      INTEGER FUNCTION MY_LOCK(LOCK)
C$DIR GATE(LOCK)
      LCK = LOCK_GATE(LOCK)
      MY_LOCK = LCK
      RETURN
      END

      INTEGER FUNCTION MY_UNLOCK(LOCK)
C$DIR GATE(LOCK)
      LCK = UNLOCK_GATE(LOCK)
      MY_UNLOCK = LCK
      RETURN
      END
```

Here, MY_LOCK and MY_UNLOCK are user functions that call the LOCK_GATE and UNLOCK_GATE intrinsics. The SYNC_ROUTINE directive prevents the compiler from moving code across the calls to MY_LOCK and MY_UNLOCK.

Such a programming technique might be used to implement code that is portable across several parallel architectures that support critical sections using different syntax; MY_LOCK and MY_UNLOCK could simply be modified to call the correct locking and unlocking functions.

An analogous C example follows:

```
#include <spp_prog_model.h>
main() {
  int i, lck;
  gate_t lock;
#pragma _CNX sync_routine(mylock, myunlock)
   .
   .
   .
  lck = alloc_gate(&lock);
#pragma _CNX loop_parallel(ivar=i)
  for(i=0; i<n; i++) {
    lck = mylock(&lock);
    .
    .
    .
    sum = sum+a[i];
    lck = myunlock(&lock);
  }
}

int mylock(gate_t *lock) {
  int lck;
  lck = lock_gate(lock);  return lck;
}
int myunlock(gate_t *lock) {
  int lck;
  lck = unlock_gate(lock);
  return lck;
}
```

`sync_routine` is also useful when CPSlib routines are used for synchronization. Refer to Appendix D, "Compiler Parallel Support Library," for more information.

loop_parallel (ordered)

The `loop_parallel(ordered)` directive and pragma was briefly introduced in Chapter 4, "Basic shared-memory programming." It is designed to be used with ordered sections (which are discussed in the next section) to execute loops with ordered dependences in loop order. It accomplishes this by parallelizing the loop so that consecutive iterations are initiated on separate processors, in loop order. While `loop_parallel(ordered)` guarantees starting order, it does not guarantee ending order, and it provides no automatic synchronization. To avoid wrong answers, you *must* manually synchronize dependences using the ordered section directives, pragmas, or the synchronization intrinsics.

Consider the following Fortran example:

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 100
         .
         . !CODE CONTAINING ORDERED SECTION
         .
      ENDDO
```

Or the analogous C code:

```
#pragma _CNX loop_parallel(ordered, ivar = i)
for(i=0;i<100;i++) {
   .
   . /* code containing ordered section */
   .
}
```

Assume that the body of this loop contains code that is parallelizable except for an ordered data dependence (otherwise there is no need to order the parallelization). This dependence is isolated using directives described in the next section. Also assume that 8 threads, numbered 0..7, are available to run the loop in parallel. Each thread would then execute code equivalent to the following:

```
DO I = (my_thread()+1), 100, num_threads()
   ...
ENDDO
```

Figure 23 illustrates the idea.

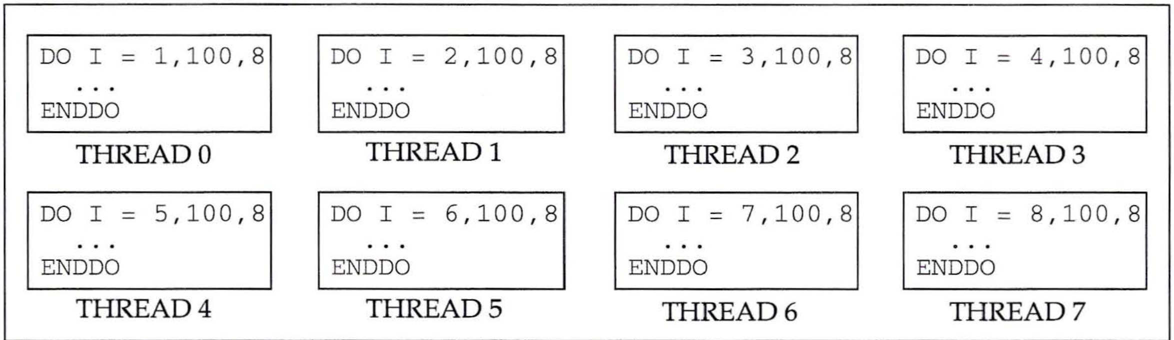


Figure 23 Ordered parallelization

Here, thread 0 executes first, followed by thread 1, and so on; each thread starts its iteration after the preceding iteration has started. A manually-defined ordered section will prevent one thread from executing the code in the ordered section until the previous thread exits the section, so thread 0 cannot enter the section for iteration 9 until thread 7 exits it for iteration 8. Obviously, this is only efficient if the loop body contains enough code to keep a thread busy until all other threads start their consecutive iterations, thus taking advantage of parallelism. You may find the `max_threads` attribute helpful when fine-tuning `loop_parallel(ordered)` loops to fully exploit their parallel code.

Examples of synchronizing `loop_parallel(ordered)` loops are given in the "Synchronizing code" section.

Critical and ordered sections

As discussed in Chapter 4, "Basic shared-memory programming," critical sections allow you to synchronize simple, nonordered dependences.

The critical_section and end_critical_section directives and pragmas are used to specify critical sections. In Fortran, these directives have the following form:

```
C$DIR CRITICAL_SECTION[(gate)]
      ...
C$DIR END_CRITICAL_SECTION
```

In C, these pragmas have the following form:

```
#pragma _CNX critical_section[(gate)]
...
#pragma _CNX end_critical_section
```

Where *gate* is an optional gate variable used for access to the critical section. *gate* must be appropriately declared as described in the "Gates and barriers" section of this chapter.

The gate variable is required when synchronizing access to a shared variable from multiple parallel tasks. When a gate variable is specified, it must be allocated (using the alloc_gate intrinsic) outside of parallel code prior to use. If no gate is specified, the compiler creates a unique gate for the critical section. When a gate is no longer needed, it should be deallocated using the free_gate function.

Critical sections must be entered through the critical_section and exited through the end_critical_section directive or pragma. They must not contain branches to outside the section. The two directives must appear in the same procedure, but they do not have to be in the same procedure as the parallel construct in which they are used; that is, the directives can exist in a procedure which is called in parallel.

Ordered sections, discussed in detail here for the first time, allow you to synchronize dependences that must execute in iteration order.

The ordered_section and end_ordered_section directives and pragmas are used to specify critical sections within manually-defined, ordered loop_parallel loops only. In Fortran, these directives have the following form:

```
C$DIR ORDERED_SECTION(gate)
      ...
C$DIR END_ORDERED_SECTION
```

In C, these pragmas have the following form:

```
#pragma _CNX ordered_section(gate)
      ...
#pragma _CNX end_ordered_section
```

Where *gate* is a required gate variable that must be allocated and, if necessary, unlocked prior to invocation of the parallel loop containing the ordered section. *gate* must be appropriately declared as described in the "Gates and barriers" section of this chapter.

Ordered sections must be entered through the ordered_section and exited through the end_ordered_section directive or pragma; they cannot contain branches to outside the section. Ordered sections are subject to the same control flow rules as critical sections.

Use critical and ordered sections with care, as they add synchronization overhead to your program. They should only be used when the amount of parallel code is significantly larger than the amount of code containing the dependence.

Synchronizing code

Code containing dependences can be parallelized by synchronizing the way the parallel tasks access the dependence. This can be done manually using the gates, barriers and synchronization functions, or semiautomatically using critical and ordered sections.

Critical sections

The critical section example shown in "Critical sections" on page 146 isolates a single critical section in a loop, so the critical_section directive does not require a gate. In this case, the critical section directives automate allocation, locking, unlocking and deallocation of the needed gate. Multiple dependences and dependences in manually-defined parallel tasks can be handled when user-defined gates are used with the directives.

Consider the following Fortran example:

```
      REAL GLOBAL_SUM
C$DIR FAR_SHARED(GLOBAL_SUM)
C$DIR GATE(SUM_GATE)
      .
      .
      .
      LOCK = ALLOC_GATE(SUM_GATE)
C$DIR BEGIN_TASKS
      CONTRIB1 = 0.0
      DO J = 1, M
        CONTRIB1 = CONTRIB1 + FUNC1(J)
      ENDDO
      .
      .
      .
C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM = GLOBAL_SUM + CONTRIB1
C$DIR END_CRITICAL_SECTION
      .
      .
      .
C$DIR NEXT_TASK
      CONTRIB2 = 0.0
      DO I = 1, N
        CONTRIB2 = CONTRIB2 + FUNC2(J)
      ENDDO
      .
      .
      .
C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM = GLOBAL_SUM + CONTRIB2
C$DIR END_CRITICAL_SECTION
      .
      .
      .
C$DIR END_TASKS
      LOCK = FREE_GATE(SUM_GATE)
```

Here, both parallel tasks must access the shared GLOBAL_SUM variable, which is assigned a function of itself. To ensure that GLOBAL_SUM is only updated by one task at a time, it is placed in a critical section. The critical sections both reference the SUM_GATE variable; this variable is unlocked on entry into the parallel code (gates are always unlocked when they are allocated). When one task reaches the critical section, the CRITICAL_SECTION directive automatically locks SUM_GATE. The END_CRITICAL_SECTION directive unlocks SUM_GATE on

exit from the section. Because access to both critical sections is controlled by a single gate, the sections must execute one at a time.

An analogous C example follows:

```
static far_shared float global_sum;
static gate_t sum_gate;
.
.
.
lock = alloc_gate(&sum_gate);
#pragma _CNX begin_tasks
contrib1 = 0.0;
for(j=0;j<m;j++)
  contrib1 = contrib1 + func1(j);
.
.
.
#pragma _CNX critical_section(sum_gate)
global_sum = global_sum + contrib1;
#pragma _CNX end_critical_section
.
.
.
#pragma _CNX next_task
contrib2 = 0.0;
for(i=0;i<n;i++)
  contrib2 = contrib2 + func2(j);
.
.
.
#pragma _CNX critical_section(sum_gate)
global_sum = global_sum + contrib2;
#pragma _CNX end_critical_section
.
.
.
#pragma _CNX end_tasks
lock = free_gate(&sum_gate);
```

Gated critical sections are also useful in loops containing multiple critical sections, when there are dependences between the critical sections. If no dependences exist between the sections, gates are not needed, as the compiler will automatically supply a unique gate for every critical section lacking a gate.

Consider the following Fortran example:

```
      REAL ABSUM
C$DIR FAR_SHARED(ABSUM)
C$DIR GATE(GATE1)
      LOGICAL ADJB(...)
      .
      .
      .
      LOCK = ALLOC_GATE(GATE1)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        A(I) = B(I) + C(I)
C$DIR   CRITICAL_SECTION(GATE1)
        ABSUM = ABSUM + A(I)
C$DIR   END_CRITICAL_SECTION
        IF(ADJB(I)) THEN
          B(I) = C(I) + D(I)
C$DIR     CRITICAL_SECTION(GATE1)
          ABSUM = ABSUM + B(I)
C$DIR     END_CRITICAL_SECTION
        ENDIF
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(GATE1)
```

Here, the shared variable ABSUM must be updated after A(I) is assigned and again if B(I) is assigned. Access to ABSUM must be guarded by the same gate to ensure that two threads do not attempt to update it at once. The critical sections protecting the assignment to ABSUM must explicitly name this gate, or the compiler will choose unique gates for each section, potentially resulting in incorrect answers. Note that there must be a substantial amount of parallelizable code outside of these critical sections to make parallelizing this loop cost-effective.

An analogous C example follows:

```
static far_shared float absum;
static gate_t gate1;
int adjb[...];
.
.
.
lock = alloc_gate(&gate1);
#pragma _CNX loop_parallel(ivar = i)
for(i=0;i<n;i++) {
  a[i] = b[i] + c[i];
#pragma _CNX critical_section(gate1)
  absum = absum + a[i];
#pragma _CNX end_critical_section
  if(adjb[i]) {
    b[i] = c[i] + d[i];
#pragma _CNX critical_section(gate1)
    absum = absum + b[i];
#pragma _CNX end_critical_section
  }
  .
  .
  .
}
lock = free_gate(&gate1);
```

Ordered sections

Like critical sections, ordered sections do the work of locking and unlocking a specified gate to isolate a section of code in a loop. However, they also ensure that the enclosed section of code executes in the same order as the iterations of the ordered parallel loop that contains it. Once a given thread passes through an ordered section, it cannot enter again until all other threads have passed through in order. This ordering is difficult to implement without using the ordered section directives or pragmas.

Note

You must use a `loop_parallel(ordered)` directive or pragma to parallelize any loop containing an ordered section.

Consider the following Fortran code, which contains a backward loop-carried dependence on the array A that would normally inhibit parallelization.

```
DO I = 2, N
   . ! PARALLELIZABLE CODE...
   .
   .
   A(I) = A(I-1) + B(I)
   . ! MORE PARALLELIZABLE CODE...
   .
   .
ENDDO
```

To simplify illustration, we will stick to Fortran, but an analogous C example could similarly be parallelized.

Assuming that the dependence shown is the only one in the loop, and that a significant amount of parallel code exists elsewhere in the loop, we can isolate the dependence and parallelize the loop as shown in the following example:

```
C$DIR GATE(LCD)
      LOCK = ALLOC_GATE(LCD)
      .
      .
      .
      LOCK = UNLOCK_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 2, N
         . ! PARALLELIZABLE CODE...
         .
         .
C$DIR    ORDERED_SECTION(LCD)
         A(I) = A(I-1) + B(I)
C$DIR    END_ORDERED_SECTION
         . ! MORE PARALLELIZABLE CODE...
         .
         .
      ENDDO
      LOCK = FREE_GATE(LCD)
```

The loop is now parallelized in the manner described in the section "`loop_parallel (ordered)`" on page 221, and the ordered section containing the `A(I)` assignment will execute in iteration order, ensuring that the value of `A(I-1)` used in the assignment is always valid. Assuming this loop runs on 4 threads, the synchronization of statement execution between threads is illustrated in Figure 24.

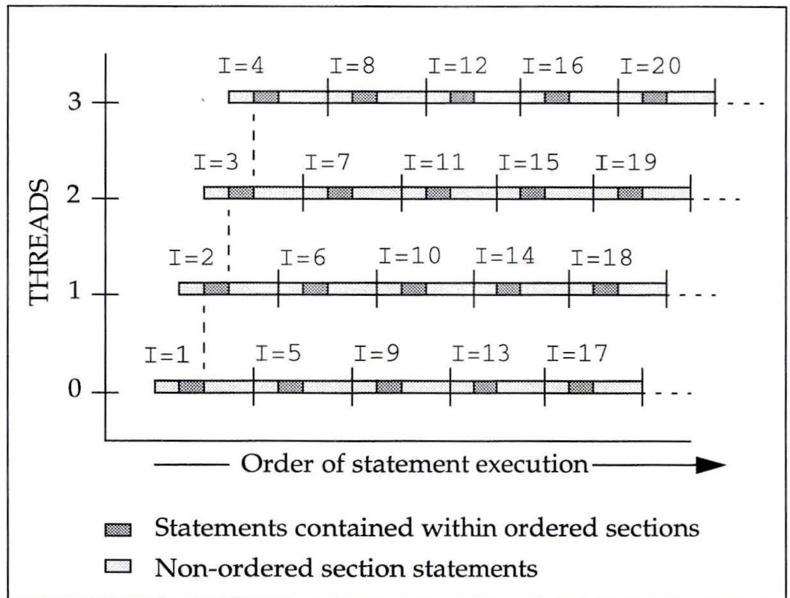


Figure 24 `LOOP_PARALLEL(ORDERED)` synchronization

As shown by the dashed lines between initial iterations for each thread, one ordered section must be done before the next is allowed to begin execution. Once a thread exits an ordered section, it cannot reenter it until all other threads have passed through in sequence. Overlap of nonordered statements, represented as lightly shaded boxes, allows all threads to proceed fully loaded, with only brief idle periods on 1, 2, and 3 at the beginning of the loop, and on 0, 1, and 2 at the end.

Limitations

Each thread in a parallel loop containing an ordered section must pass through the ordered section once and only once on every iteration of the loop. If you execute an ordered section conditionally, you must execute it in all possible branches of the condition; if the code contained in the section is not valid for some branches, you can insert a blank ordered section, as shown in the following Fortran example:

```
C$DIR GATE (LCD)
      .
      .
      .
      LOCK = ALLOC_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, N
        .
        .
        .
        IF (Z(I) .GT. 0.0) THEN
C$DIR     ORDERED_SECTION(LCD)
C         HERE'S THE BACKWARD LCD:
          A(I) = A(I-1) + B(I)
C$DIR     END_ORDERED_SECTION
        ELSE
C         HERE IS THE BLANK ORDERED SECTION:
C$DIR     ORDERED_SECTION(LCD)
C$DIR     END_ORDERED_SECTION
        ENDIF
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(LCD)
```

Here, no matter which path through the `IF` statement the loop takes, it must pass through the ordered section, even though the `ELSE` section is empty. This allows the compiler to properly synchronize the ordered loop. Note that again, we assume a substantial amount of parallel code exists outside the ordered sections, to offset the synchronization overhead.

An analogous C example follows:

```
static gate_t lcd;
.
.
.
lock = alloc_gate(&lcd);
#pragma _CNX loop_parallel(ordered,ivar = i)
for(i=0;i<ni++) {
  .
  .
  .
  if(z[i] > 0.0) {
#pragma _CNX ordered_section(lcd)
    a[i] = a[i-1] + b[i]; /* backward lcd */
#pragma _CNX end_ordered_section
  } else {
#pragma _CNX ordered_section(lcd)
    /* here is the blank ordered section */
#pragma _CNX end_ordered_section
  }
  .
  .
  .
}
lock = free_gate(&lcd);
```

Ordered sections within nested loops can create similar, but more difficult to recognize, problems. Consider the following Fortran example (gate manipulation is omitted for brevity):

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 99
        DO J = 1,M
          .
          .
          .
C$DIR     ORDERED_SECTION(ORDGATE)
          A(I,J) = A(I+1,J)
C$DIR     END_ORDERED_SECTION
          .
          .
          .
        ENDDO
      ENDDO
```

Recall that once a given thread has passed through an ordered section, it cannot reenter it until all other threads have passed through in order. This is only possible in the given example if the

number of available threads integrally divides 99 (the I loop limit). If not, deadlock results.

To see why, assume 6 threads, numbered 0 through 5, are running the parallel I loop. For I = 1, J = 1, thread 0 passes through the ordered section and loops back through J, stopping when it reaches the ordered section again for I = 1, J = 2. It cannot enter until threads 1 through 5 (which are executing I = 2 through 6, J = 1 respectively) pass through in sequence. This is not a problem, and the loop proceeds through I = 96 in this fashion in parallel. However, for I > 96, all 6 threads are no longer needed. In a single loop nest this would not pose a problem; the leftover 3 iterations would be handled by threads 0 through 2; when thread 2 exited the ordered section it would hit the ENDDO and the I loop would terminate normally. But in this example, the J loop isolates the ordered section from the I loop, so thread 0 executes J = 1 for I = 97, loops through J and waits during J = 2 at the ordered section for thread 5, which has gone idle, to complete. Threads 1 and 2 similarly execute J = 1 for I = 98 and I = 99, and similarly wait after incrementing J to 2. The entire J loop must terminate before the I loop can terminate, but the J loop can never terminate because the idle threads 3, 4, and 5 never pass through the ordered section. Deadlock results.

The analogous C code looks like this:

```
#pragma _CNX loop_parallel(ordered,ivar = i)
for(i=0;i<99;i++) {
  for(j=0;j<m;j++) {
    .
    .
    .
#pragma _CNX ordered_section(ordgate)
    a[i][j] = a[i+1][j];
#pragma _CNX end_ordered_section
    .
    .
    .
  }
}
```

To handle this problem, you can expand the ordered section to include the entire J loop, as shown in the following Fortran example:

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1, 99
C$DIR   ORDERED_SECTION(ORDGATE)
        DO J = 1,M
          .
          .
          .
          A(I,J) = A(I+1,J)
          .
          .
          .
        ENDDO
C$DIR   END_ORDERED_SECTION
      ENDDO
```

In this approach, each thread executes the entire J loop each time it enters the ordered section, allowing the I loop to terminate normally regardless of the number of threads available.

The analogous C code follows:

```
#pragma _CNX loop_parallel(ordered,ivar = i)
for(i=0;i<99;i++) {
#pragma _CNX ordered_section(ordgate)
  for(j=0;j<m;j++) {
    .
    .
    .
    a[i][j] = a[i+1][j];
    .
    .
    .
  }
#pragma _CNX end_ordered_section
}
```

Another approach is to manually interchange the I and J loops, as shown in the following example:

```
      DO J = 1, M
C$DIR   LOOP_PARALLEL(ORDERED)
        DO I = 1, 99
          .
          .
          .
C$DIR     ORDERED_SECTION(ORDGATE)
          A(I,J) = A(I+1,J)
C$DIR     END_ORDERED_SECTION
          .
          .
          .
        ENDDO
      ENDDO
```

Here, the I loop is parallelized on every iteration of the J loop. Again, the ordered section is not isolated from its parent loop, so the loop can terminate normally. This example has added benefits; here, elements of A are accessed more efficiently, and this nest can be further optimized by parallelizing the J loop across hypernodes by using a LOOP_PARALLEL(NODES) directive.

The analogous C code follows:

```
#pragma _CNX loop_parallel(ordered,ivar = i)
for(j=0;j<99;j++) {
  for(i=0;i<m;i++) {
    .
    .
    .
#pragma _CNX ordered_section(ordgate)
    a[i][j] = a[i+1][j];
#pragma _CNX end_ordered_section
    .
    .
    .
  }
}
```

Manual synchronization

Ordered and critical sections allow you to isolate dependences in a structured, semiautomatic manner. The same isolation can be accomplished manually using the functions discussed in the "Synchronization functions" section of this chapter.

Recall our simple critical section example:

```
C$DIR LOOP_PARALLEL
      DO I = 1, N   ! LOOP IS PARALLELIZABLE
        .
        .
        .
C$DIR   CRITICAL_SECTION
        SUM = SUM + X(I)
C$DIR   END_CRITICAL_SECTION
        .
        .
        .
      ENDDO
```

As shown, this example is easily implemented using critical sections. It can be manually implemented in Fortran as shown below.

```
C$DIR GATE(CRITSEC)
      .
      .
      .
      LOCK = ALLOC_GATE(CRITSEC)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        .
        .
        .
        LOCK = LOCK_GATE(CRITSEC)
        SUM = SUM + X(I)
        LOCK = UNLOCK_GATE(CRITSEC)
        .
        .
        .
      ENDDO
      LOCK = FREE_GATE(CRITSEC)
```

As shown, the manual implementation requires declaring, allocating and deallocating a gate, which must be locked on entry into the critical section using the `LOCK_GATE` function and unlocked on exit using `UNLOCK_GATE`.

An analogous manual C implementation follows:

```
static gate_t critsec;
.
.
.
lock = alloc_gate(&critsec);
#pragma _CNX loop_parallel(ivar = i)
for(i=0;i<n;i++) {
   .
   .
   .
   lock = lock_gate(&critsec);
   sum = sum + x[i];
   lock = unlock_gate(&critsec);
   .
   .
   .
}
lock = free_gate(&critsec);
```

Using synchronization functions to implement critical and ordered sections generally requires more work, and the compiler will not check such constructs for errors as thoroughly as it will check directive-delimited sections. However, because the functions are unstructured, they can be used to create more complex critical regions than are supported by the directives.

Consider the following Fortran example:

```
C$DIR GATE(TASK1, TASK2)
      .
      .
      .
      LOK1 = ALLOC_GATE(TASK1)
      LOK2 = ALLOC_GATE(TASK2)
      LOK1 = LOCK_GATE (TASK1)   ! LOCKING HERE PREVENTS 3RD TASK
      LOK2 = LOCK_GATE (TASK2)   ! FROM ACCESSING A OR B BEFORE
                                 ! FIRST TWO TASKS ASSIGN THEM
C$DIR BEGIN_TASKS(ORDERED)   ! TASK ONE:
      DO I = 1, N
        A(I) = 2.0 * A(I) + C(I) ! A GETS ASSIGNED IN THIS TASK
      ENDDO
      LOK1 = UNLOCK_GATE (TASK1) ! A CAN NOW BE ACCESSED
C$DIR NEXT_TASK    ! TASK TWO:
      DO J = 1, N
        B(J) = C(J) * SIN (X(J)) ! B GETS ASSIGNED IN THIS TASK
      ENDDO
      LOK2 = UNLOCK_GATE (TASK2) ! B CAN NOW BE ACCESSED
C$DIR NEXT_TASK     ! TASK THREE:
      DO K = 1, N   ! COMPUTE P IN PARALLEL WITH A AND B:
        P(K) = EXP (3.0 * SQRT (Q(K))) / ATAN (R(K))
      ENDDO
      LOK1 = LOCK_GATE (TASK1)    ! WAIT FOR UNLOCK IN TASK 1
      LOK1 = UNLOCK_GATE (TASK1)  ! WHEN LOCK IS ATTAINED,UNLOCK
      LOK2 = LOCK_GATE (TASK2)    ! WAIT FOR UNLOCK IN TASK 2
      LOK2 = UNLOCK_GATE (TASK2)  ! WHEN ATTAINED, UNLOCK
      DO L = 1, N                 ! WHEN WE HAVE BOTH LOCKS,
        D(L) = P(L) * B(L) / A(L) ! COMPUTE D
      ENDDO
C$DIR NEXT_TASK    ! TASK FOUR:
      DO M = 1, N                 ! NO DEPENDENCES IN THIS TASK
        Y(M) = X(M) * Y(M) / Z(M) ! Y CAN BE COMPUTED WITH
      ENDDO                       ! A, B, P, D
C$DIR END_TASKS
      LOK1 = FREE_GATE(TASK1)
      LOK2 = FREE_GATE(TASK2)
```

Here, the BEGIN_TASKS(ORDERED) directive guarantees that the following parallel tasks begin in lexical order (ending order is indeterminate). Ordered sections, however, cannot be used with parallel tasks. To order the dependence in task 3 of this code, where the computation of D assumes that A, B and P are fully computed, we must use manually locked and unlocked gates.

Tasks 1, 2, 4 and the K loop of task 3 can all run in parallel. To allow this while postponing execution of the L loop in task 3 until A, B

and P are computed, we allocate and lock two gates, named TASK1 and TASK2, before the parallel tasks begin. TASK1 remains locked until A is computed, at which point it is unlocked; TASK2 remains locked until B is computed, and is then unlocked. The I and J loops are free to run in parallel, along with the K loop in task 3 and the M loop in task 4, since none of these loops depend on each other's gates. The L loop in task 3, however, cannot begin execution until task 3 can lock both TASK1 and TASK2 gates. Task 3 immediately unlocks these gates because it does not need them; their purpose was to force it to wait until A, B and P were computed. Once task 3 has acquired and relinquished both locks, the L loop is free to run.

A similar C example follows:

```
static gate_t task1, task2;
.
.
.
lok1 = alloc_gate(&task1);
lok2 = alloc_gate(&task2);
lok1 = lock_gate(&task1); /* locking here prevents 3rd task   */
lok2 = lock_gate(&task2); /* from accessing a or b before     */
                          /* first two tasks assign them      */
#pragma _CNX begin_tasks(ordered) /* task 1: */
for(i=0;i<n;i++)
  a[i] = 2.0 * a[i] + c[i];  /* a gets assigned in this task */
lok1 = unlock_gate(&task1);  /* a can now be accessed        */
#pragma _CNX next_task            /* task 2: */
for(j=0;j<n;j++)
  b[j] = c[j] * sin(x[j]);   /* b gets assigned in this task */
lok2 = unlock_gate(&task2);   /* b can now be accessed        */
#pragma _CNX next_task            /* task 3: */
for(k=0;k<n;k++)          /* compute p in parallel with a and b */
  p[k] = exp(3.0*sqrt(q[k]))/atan(r[k]);
lok1 = lock_gate(&task1);   /* wait for unlock in task 1      */
lok1 = unlock_gate(&task1); /* when lock is attained, unlock  */
lok2 = lock_gate(&task2);   /* wait for unlock in task 2      */
lok2 = unlock_gate(&task2); /* when attained, unlock          */
for(l=0;l<n;l++)            /* when we have both locks,       */
  d[l] = p[l] * b[l]/a[l];  /* compute d                      */
#pragma _CNX next_task            /* task 4: */
for(m=0;m<n;m++)            /* no dependences in this task   */
  y[m] = x[m] * y[m]/z[m];  /* y can be computed with a,b,p,d */
#pragma _CNX end_tasks
lok1 = free_gate(&task1);
lok2 = free_gate(&task2);
```

Another advantage of manually-defined critical sections is the ability to conditionally lock them. This allows the task that wishes to execute the section to proceed with other work if the lock cannot be acquired. This construct is useful, for example, in situations where one thread is performing I/O for several other parallel threads; while a processing thread is reading from the input queue, the queue is locked, and the I/O thread can move on to do output. While a processing thread is writing to the output queue, the I/O thread can do input. This allows the I/O thread to keep as busy as possible while the parallel computational threads execute their (presumably large) computational code. This situation is illustrated in the following Fortran example. Task 1 performs I/O for the 7 other tasks, which perform parallel computations by calling the THREAD_WRK subroutine.

```
      COMMON INGATE,OUTGATE,STOPGATE,COMPBAR,DONEGATE
C$DIR GATE (INGATE, OUTGATE, STOPGATE, DONEGATE)
C$DIR BARRIER (COMPBAR)
      REAL DIN(:), DOUT(:)        ! I/O BUFFERS FOR TASK 1
      ALLOCATABLE DIN, DOUT       ! THREAD 0 WILL ALLOCATE
      REAL QIN(1000,1000), QOUT(1000,1000) ! SHARED I/O QUEUES
      INTEGER NIN/0/,NOUT/0/ ! QUEUE ENTRY COUNTERS
C     CIRCULAR BUFFER POINTERS:
      INTEGER IN_QIN/1/,OUT_QIN/1/,IN_QOUT/1/,OUT_QOUT/1/
        COMMON /DONE/ DONEIN, DONECOMP
      LOGICAL DONECOMP, DONEIN
C                           SIGNALS FOR COMPUTATION DONE AND INPUT DONE
      LOGICAL COMPDONE, INDONE
C                           FUNCTIONS TO RETURN DONECOMP AND DONEIN
      LOGICAL INFLAG, OUTFLAG  ! INPUT READ AND OUTPUT WRITE FLAGS
C$DIR THREAD_PRIVATE (INFLAG,OUTFLAG) ! ONLY NEEDED BY TASK 1
C                                         (WHICH RUNS ON THREAD 0)
        IF (NUM_THREADS() .LT. 8) STOP 1
        IN = 10
        OUT = 11
      LOCK = ALLOC_GATE(INGATE)
      LOCK = ALLOC_GATE(OUTGATE)
      LOCK = ALLOC_GATE(STOPGATE)
      IBAR = ALLOC_BARRIER(COMPBAR)
      LOCK = LOCK_GATE(STOPGATE)
      DONECOMP = .FALSE.
C$DIR BEGIN_TASKS                  ! TASK 1 STARTS HERE
      INFLAG = .TRUE.
      DONEIN = .FALSE.
      ALLOCATE(DIN(1000),DOUT(1000)) ! ALLOCATE LOCAL BUFFERS
```

```
      DO WHILE(.NOT. INDONE() .OR.. .NOT. COMPDONE() .OR. NOUT .GT. 0)
C                       DO TILL EOF AND COMPUTATION DONE AND OUTPUT DONE
        IF(NIN.LT.1000.AND.(.NOT.COMPDONE()) .AND.(.NOT. INDONE())) THEN
C                       FILL QUEUE
          IF (INFLAG) THEN  ! FILL BUFFER FIRST:
            READ(IN, IOSTAT = IOS) DIN  ! READ A RECORD; QUIT ON EOF
            IF(IOS .EQ. -1) THEN
              DONEIN = .TRUE. ! SIGNAL THAT INPUT IS DONE
              INFLAG = .TRUE.
            ELSE
              INFLAG = .FALSE.
            ENDIF
          ENDIF
C SYNCHRONOUSLY ENTER INTO INPUT QUEUE:
C         BLOCK QUEUE ACCESS WITH INGATE:
          IF (COND_LOCK_GATE(INGATE) .EQ. 0 .AND. .NOT. INDONE()) THEN
            QIN(:,IN_QIN) = DIN(:)  ! COPY INPUT BUFFER INTO QIN
            IN_QIN=1+MOD(IN_QIN,1000)  ! INCREMENT INPUT BUFFER PTR
            NIN = NIN + 1  ! INCREMENT INPUT QUEUE ENTRY COUNTER
            INFLAG = .TRUE.
            LOCK = UNLOCK_GATE(INGATE)  ! ALLOW INPUT QUEUE ACCESS
          ENDIF
        ENDIF
C SYNCHRONOUSLY REMOVE FROM OUTPUT QUEUE:
C         BLOCK QUEUE ACCESS WITH OUTGATE:
          IF (COND_LOCK_GATE(OUTGATE) .EQ. 0) THEN
           IF (NOUT .GT. 0) THEN
            DOUT(:)=QOUT(:,OUT_QOUT)  ! COPY OUTPUT QUE INTO BUFFR
            OUT_QOUT=1+MOD(OUT_QOUT,1000) ! INCREMENT OUTPUT BUFR PTR
            NOUT = NOUT - 1  ! DECREMENT OUTPUT QUEUE ENTRY COUNTR
            OUTFLAG = .TRUE.
           ELSE
            OUTFLAG = .FALSE.
           ENDIF
            LOCK = UNLOCK_GATE(OUTGATE)  ! ALLOW OUTPUT QUEUE ACCESS
            IF (OUTFLAG) WRITE(OUT) DOUT  ! WRITE A RECORD
          ENDIF
      ENDDO
C                               TASK 1 ENDS HERE
```

```
C$DIR NEXT_TASK                    ! TASK 2:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                    ! TASK 3:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                    ! TASK 4:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                    ! TASK 5:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                    ! TASK 6:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                    ! TASK 7:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
C$DIR NEXT_TASK                    ! TASK 8:
      CALL THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      IBAR = WAIT_BARRIER(COMPBAR,7)
      DONECOMP = .TRUE.
C$DIR END_TASKS
        END
```

Before looking at the THREAD_WRK subroutine we'll examine these parallel tasks, especially task 1, the I/O server.

Task 1 does all the I/O required by all the tasks; the other 7 tasks do computational work, receiving their input from and sending their output to task 1's queues. Conditionally-locked gates control task 1's access to one section of code that fills the input queue and one that empties the output queue. Task 1 works by first filling an input buffer; the code that does this does not require gate protection because no other tasks attempt to access the input buffer array. The section of code where the input buffer is copied into the input queue, however, must be protected by gates to prevent any threads from trying to read the input queue while it is being filled.

Similarly, if a task acquires a lock on the input queue, task 1 cannot fill it until the task is done reading from it. When task 1 cannot get a lock to access the input queue code, it goes on and tries to lock the output queue code. If it gets a lock here, it can copy the output queue into the output buffer array and relinquish the lock; it can then proceed to empty the output buffer. If another task is writing to the output queue, task 1 loops back and begins the entire process over again. When the end of the input file is reached, task 1 is finished. Note that the task loops on DONEIN (via

INDONE()), which is initially false. When input is exhausted, DONEIN is set to true, signalling all tasks that there is no more input.

The INDONE() function references DONEIN, forcing a memory reference. If DONEIN were referenced directly, the compiler might optimize it into a register and consequently not detect a change in its value.

So task 1 has four main jobs to do:

1. Read input into input buffer—no other tasks access the input buffer, so this can be done in parallel regardless of what other tasks are doing, as long as the buffer needs filling.
2. Copy input buffer into input queue—the other tasks read their input from the input queue; therefore it can only be filled when no computational task is reading it. This section of code is protected by the INGATE gate. It can run in parallel with the computational portions of other tasks, but only one task can access the input queue at a time.
3. Copy output queue into output buffer—the output queue is where other tasks write their output, so it can only be emptied when no computational task is writing to it. This section of code is protected by the OUTGATE gate. It can run in parallel with the computational portions of other tasks, but only one task can access the output queue at a time.
4. Write out output buffer—no other tasks access the output buffer, so this can be done in parallel regardless of what the other tasks are doing.

Now we will look at the subroutine THREAD_WRK, which tasks 2-7 call to perform computations.

```
      SUBROUTINE
     > THREAD_WRK(NIN,NOUT,QIN,QOUT,IN_QIN,OUT_QIN,IN_QOUT,OUT_QOUT)
      INTEGER NIN,NOUT
      REAL QIN(1000,1000), QOUT(1000,1000) ! SHARED I/O QUEUES
      INTEGER OUT_QIN, OUT_QOUT
      COMMON INGATE,OUTGATE,COMPBAR,DONEGATE
C$DIR GATE(INGATE, OUTGATE)
      REAL WORK(1000)      ! LOCAL THREAD PRIVATE WORK ARRAY
        LOGICAL OUTFLAG, INDONE
        OUTFLAG = .FALSE.
C$DIR THREAD_PRIVATE (WORK) ! EVERY THREAD WILL CREATE A COPY

      DO WHILE(.NOT. INDONE() .OR. NIN.GT.0 .OR. OUTFLAG)
C                                  WORK/QOUT EMPTYING LOOP
       IF (.NOT. OUTFLAG) THEN   ! IF NO PENDING OUTPUT
C$DIR        CRITICAL_SECTION (INGATE)   ! BLOCK ACCESS TO INPUT QUE
          IF (NIN .GT. 0) THEN   ! MORE WORK TO DO
            WORK(:) = QIN(:,OUT_QIN)
            OUT_QIN = 1 + MOD(OUT_QIN, 1000)
            NIN = NIN - 1
            OUTFLAG = .TRUE.
C                           INDICATE THAT INPUT DATA HAS BEEN RECEIVED
          ENDIF
C$DIR        END_CRITICAL_SECTION
          .
          .  ! SIGNIFICANT PARALLEL CODE HERE USING WORK ARRAY
          .
       ENDIF
       IF (OUTFLAG) THEN   ! IF PENDING OUTPUT, MOVE TO OUTPUT QUEUE
C AFTER INPUT QUEUE IS USED IN COMPUTATION, FILL OUTPUT QUEUE:
C$DIR   CRITICAL_SECTION (OUTGATE) ! BLOCK ACCESS TO OUTPUT QUEUE
        IF(NOUT.LT.1000) THEN   ! IF THERE IS ROOM IN THE OUTPUT QUEUE
          QOUT(:,IN_QOUT) = WORK(:) ! COPY WORK INTO OUTPUT QUEUE
          IN_QOUT =1+MOD(IN_QOUT,1000) ! INCREMENT BUFFER PTR
          NOUT = NOUT + 1 ! INCREMENT OUTPUT QUEUE ENTRY COUNTER
          OUTFLAG = .FALSE.   ! INDICATE NO OUTPUT PENDING
        ENDIF
C$DIR   END_CRITICAL_SECTION
       ENDIF
      ENDDO ! END WORK/QOUT EMPTYING LOOP
      END   ! END THRD_WRK
```

```
      LOGICAL FUNCTION INDONE()
C THIS FUNCTION FORCES A MEMORY REFERENCE TO GET THE DONEIN VALUE
      LOGICAL DONEIN
      COMMON /DONE/ DONEIN, DONECOMP
      INDONE = DONEIN
      END

      LOGICAL FUNCTION COMPDONE()
C THIS FUNCTION FORCES A MEMORY REFERENCE TO GET THE DONECOMP VALUE
      LOGICAL DONECOMP
      COMMON /DONE/ DONEIN, DONECOMP
      INDONE = DONECOMP
      END
```

Notice that the gates are accessed through COMMON blocks, and that each thread that calls this subroutine will allocate a thread_private WORK array.

This subroutine contains a loop that tests INDONE(). The loop copies the input queue into the local WORK array, then does a presumably significant amount of computational work that has been omitted for simplicity (because the computational work is the main code that executes in parallel, if there is not a large amount of it, the overhead of setting up these parallel tasks and critical sections cannot be justified). The loop encompasses this computation, and also the section of code that copies the WORK array to the output queue. This construct allows final output to be written after all input has been used in computation.

To avoid accessing the input queue while it is being filled or accessed by another thread, the section of code that copies it into the local WORK array is protected by a critical section. This section must be unconditionally locked; the computational threads cannot do something else until they receive their input. Once the input queue has been copied, THREAD_WRK can perform its large section of computational code in parallel with whatever the other tasks are doing. After the computational section is finished, another unconditional critical section must be entered so that the results can be written to the output queue. This prevents two threads from accessing the output queue at once.

Problems like this require performance testing and tuning to achieve optimal parallel efficiency. Variables such as the number of computational threads and the size of the I/O queues can be adjusted with experience to yield the best processor utilization.

Advanced shared-memory example

We will now consider a more detailed Fortran example that is manually parallelized in two dimensions and makes use of barrier synchronization and block_shared arrays, two topics which are difficult to illustrate in brief examples.

Consider the following Fortran code:

```
REAL*8 A(800,800,800), B(800,800,800)
   .
   .
   .
DO K = 1,800
  DO J = 1, 800
    DO I = 1, 800
      B(I,J,K) = ...
      .
      .
      .
    ENDDO
  ENDDO
ENDDO
   .
   .
   .
DO K = 2,799
  DO J = 2, 799
    DO I = 2, 799
      A(I,J,K)=A(I,J,K)+1./6.*(B(I,J+1,K)+B(I,J-1,K)+
>       B(I+1,J,K)+B(I-1,J,K)+B(I,J,K+1)+B(I,J,K-1))
      .
      .
      .
    ENDDO
  ENDDO
ENDDO
```

While this code contains several opportunities for parallelization, each of its arrays occupies nearly 4 Gbytes. The two arrays together with any additional arrays or variables required by the program therefore require far more virtual memory than is available to the process.

To get around this, we must somehow increase the virtual address space available to the process. We can do this by requiring that the process run on a minimum number of hypernodes, and allocating sections of A and B in node_private memory on the individual hypernodes that will compute them. Because these sections will only be accessible to the hypernodes on which they reside, any elements that must be shared must be copied into shared memory. We will use block_shared memory for this function. Recall that

`block_shared` arrays are distributed in chunks across hypernodes, allowing fastest access from the hypernode on which the section resides (like `node_private`), but also allowing other hypernodes to access the data. This will provide optimal array-access efficiency.

The following implementation uses manually-partitioned arrays and two-dimensional manual parallelization to solve this example as efficiently as possible. It assumes that performance testing indicates that a minimum of 4 hypernodes is required to adequately provide the necessary amounts of virtual memory.

```
      REAL*8 A, B
C     EACH NODE WILL ALLOCATE A 1/NUM_NODES-SIZE SECTION OF EACH ARRAY:
      ALLOCATABLE A(:,:,:),B(:,:,:)
C$DIR NODE_PRIVATE(A,B)
C     REQUIRED FOR DYNAMIC NODE_PRIVATE ALLOCATED BY A SINGLE THRD:
C$DIR FAR_SHARED_POINTER(A,B)
      REAL*8 BLEFT, BRIGHT ! EACH NODE COMMUNICATES ITS ENDPLANES
      ALLOCATABLE BLEFT(:,:,:),BRIGHT(:,:,:)! TO OTHER NODES VIA
C$DIR BLOCK_SHARED(BLEFT, BRIGHT)            ! BLEFT/BRIGHT
      INTEGER NODE_ID, nodeNS     ! EACH NODE'S ID AND ARRAY SIZE
      INTEGER K, J, I, KS, KE     ! INDUCTION VARIABLES
C$DIR NODE_PRIVATE(NODE_ID, nodeNS, KS, KE)
C$DIR THREAD_PRIVATE(K, J, I, LBAR2)
C$DIR BARRIER(BARRALL) ! TO PREVENT USING B BEFORE IT IS ASSIGNED
      INTEGER CEILING
      CEILING(X) = AINT(1.0+X)-AINT(1.0+AINT(X)-X)
      .
      .
      .
      PRINT*, "ENTER NUMBER OF HYPERNODES:"
      READ*, NN
      IF (NN .GE. 4) THEN           ! ENOUGH MEMORY TO SOLVE
C       NS IS THE MAX ARRAY SIZE NEEDED BY AT LEAST 1 NODE:
        NS = CEILING(800./FLOAT(NN)) ! SIZE OF A,B ARRAY SECTIONS
      ELSE
        STOP 'NOT ENOUGH MEMORY'    ! STOP IF NOT ENOUGH MEMORY
      ENDIF
      .
      .
      .
C     ALLOCATE EACH NODE'S ARRAYS;
C     B'S 3RD DIMENSION HAS ROOM FOR NEIGHBOR VALUES:
      ALLOCATE(A(800,800,NS),B(800,800,0:NS+1))
```

```
C     ALLOCATE BLOCK_SHARED BLEFT AND BRIGHT TO ALLOW EACH NODE
C     TO WRITE ENDPLANES LOCALLY, SHARE NEIGHBORS GLOBALLY:
      ALLOCATE(BLEFT(800,800,NN),BRIGHT(800,800,NN))
      LBAR1 = ALLOC_BARRIER(BARRALL) ! ALLOCATE BARRIER
C$DIR LOOP_PARALLEL(NODES)
      DO NODE = 1, NN          ! BEGIN NODE-PARALLEL STRUCTURE:
        NODE_ID=MY_NODE()+1    ! INDEX INTO SHARED ARRAY.
C       FIND EXACT SIZE OF ARRAYS TO BE COMPUTED ON EACH NODE:
        IF (NODE_ID .GT. 800 - NN * (NS-1)) THEN
          nodeNS=NS-1 ! HIGH NUMBERED NODES MIGHT DO ONE LESS.
        ELSE
          nodeNS = NS ! ALL OTHER NODES DO THE SAME SIZE.
        ENDIF
C       LOOP NEST 1:
C$DIR   LOOP_PARALLEL(THREADS) ! IN THRD-PARALLEL LOOP, EACH NODE
        DO K=1,nodeNS          ! COMPUTES ITS PIECE OF THE B ARRAY
          DO J = 1, 800
            DO I = 1, 800
              B(I,J,K) = ...
              .
              .
              .
            ENDDO
          ENDDO
        ENDDO ! END LOOP NEST 1
C       LOOP NEST 2:
C       NOW PUT EACH NODE'S B ENDPLANES IN SHARED ARRAYS:
        DO J = 1, 800
          DO I = 1, 800
C           CURRENT NODE'S RIGHT PLANE IS NEIGHBOR NODE'S LEFT:
            BLEFT(I,J,NODE_ID)=B(I,J,nodeNS)
C           CURRENT NODE'S LEFT IS NEIGHBOR NODE'S RIGHT:
            BRIGHT(I,J,NODE_ID)=B(I,J,1)
          ENDDO
        ENDDO ! END LOOP NEST 2
C       BARRIER ENSURES ALL ENDPLANES ARE COPIED BEFORE USED:
        LBAR2=WAIT_BARRIER(BARRALL,NN) ! ONE THREAD PER NODE CALLS
```

```
C       LOOP NEST 3:
        IF (NODE_ID .GT. 1) THEN  ! THIS NODE IS INTERIOR OR RTMOST
          DO J = 1, 800
            DO I = 1, 800
              B(I,J,0)=BLEFT(I,J,NODE_ID-1) ! GET LEFT ENDPLANE
            ENDDO
          ENDDO
        ENDIF
        IF(NODE_ID .LT. NN) THEN   ! THIS NODE IS INTERIOR OR LFTMOST
          DO J = 1, 800
            DO I = 1, 800
              B(I,J,nodeNS+1)=BRIGHT(I,J,NODE_ID+1) ! GET RT ENDPLANE
            ENDDO
          ENDDO
        ENDIF
C       GOT THE ENDPLANES, NOW COMPUTE ARRAY BOUNDS:
        IF (NODE_ID .EQ. 1) THEN        ! THIS NODE IS LEFTMOST
          KS = 2
          KE = nodeNS
        ELSEIF (NODE_ID .EQ. NN) THEN   ! THIS NODE IS RIGHTMOST
          KS = 1
          KE = nodeNS-1
        ELSE                            ! THIS NODE IS INTERIOR
          KS = 1
          KE = nodeNS
        ENDIF
C       NOW COMPUTE ARRAY A:
C       LOOP NEST 4:
C$DIR   LOOP_PARALLEL(THREADS)    ! GO THREAD-PARALLEL ON EACH NODE
        DO K = KS, KE
          DO J = 2, 1023
            DO I = 2, 1023
              A(I,J,K)=A(I,J,K)+1./6.*(B(I,J+1,K)+B(I,J-1,K)+
     >         B(I+1,J,K)+B(I-1,J,K)+B(I,J,K+1)+B(I,J,K-1))
              .
              .
              .
            ENDDO
          ENDDO
        ENDDO ! END LOOP NEST 4
      LBAR1 = FREE_BARRIER(BARRALL) ! DEALLOCATE BARRIER
      ENDDO                         ! END NODE-PARALLEL LOOP
```

Here, the A and B arrays, which are used in the bulk of the compute-intensive code and occupy the bulk of the process's virtual memory, are assigned the `node_private` class. They are allocated in serial code, so that when they are accessed, physical copies are created on each hypernode, and each hypernode

accesses its copy using the same array names. far_shared pointers are used to access A and B, as described in Chapter 5, "Memory classes."

If enough hypernodes are not available to handle the virtual memory requirements of A and B at allocation time, the program stops.

Note that B is allocated with an extra K dimension plane at each end so that the hypernodes can satisfy computational dependences involving these endplanes by sharing them.

B is computed in loop nest 1, and is then used in loop nest 4 to compute A. B must be entirely computed before computation of A can begin. Because the computation of A contains both positive and negative offset indexing of B, each hypernode must share its B endplanes using the extra endplanes allocated before computation of A can commence.

This sharing is accomplished through the BLEFT and BRIGHT block_shared arrays. Loop nest 2 copies the endplanes from B into these arrays, and, after the BARRALL barrier ensures that this copying is complete, loop nest 3 copies the endplanes from BLEFT and BRIGHT into the appropriate extended dimensions of B on the appropriate hypernodes. Note that BLEFT and BRIGHT are allocated based on the number of hypernodes, and indexed in their third dimension by NODE_ID; this ensures that the elements most often used on a hypernode will physically reside on that hypernode. This means copying the endplanes of B into these arrays can be done with minimal access latency. Copying the endplanes into neighboring hypernodes' copies of B in loop nest 3 requires more costly interhypernode memory accesses. However, if the problem is to run on multiple hypernodes, which it must to obtain the virtual memory it needs, some interhypernode communication is inevitable, and this code minimizes it to the extent possible.

After this copying takes place, each hypernode determines its index range in the third dimension of A and B, and A is computed in loop nest 4. Here, all accesses are to node_private memory, so access latency is minimal.

This problem can be solved in parallel in a number of ways, which include splitting A and B up across threads in thread_private memory, or placing BLEFT and BRIGHT in far_shared memory. The solution presented above is more versatile and efficient.

This problem can also be solved using message passing. See Chapter 7, "Message-passing programming," and the *Convex MPICH User's Guide for Exemplar Systems* or the *PVM/GSM User's Guide for Exemplar Systems* for more information.

Message-passing programming

7

This chapter presents a high-level overview of message-passing using MPI and PVM, briefly describes the different message-passing libraries available on Exemplar systems, and lists sources for more information.

A brief overview of the message-passing and shared-memory paradigms is given in Chapter 1, "Introduction."

Overview

Message passing is an approach to writing portable parallel programs. An application that uses message passing consists of several concurrent tasks, each with its own data, using messages to communicate with one another. Message passing requires the programmer to explicitly handle all parallelism and data distribution.

Message-passing programs are inherently parallel, and unless explicitly coordinated by message waiting, all processes execute independently. In a conventionally coded message-passing program, all variables are private to each process. So regardless of whether variables have been declared to be in any of the memory classes, no process can access the variables of any other process. Synchronization among the processes occurs explicitly through message passing.

Approaches to parallelism

Message-passing programs generally take one of two approaches to parallelism: the multiple-program multiple-data (MPMD) approach (also known as the manager/worker approach) or the single-program multiple-data (SPMD) approach.

In the MPMD approach, a set of computational worker processes perform work for one or more manager processes. This approach is generally used when little synchronization is required between worker processes.

In the SPMD approach, the program spawns several identical processes that perform the same work independently on different data sets. In this approach, synchronization is often required between processes. Exemplar systems are especially suited to this model because of their fast shared-memory communication, which minimizes synchronization delays.

Message passing on Exemplar systems

Exemplar versions of the MPI and PVM message-passing libraries are available in the Convex MPICH and PVM/GSM libraries, respectively. These libraries, which are discussed below, are accessible from both C and Fortran.

Convex MPICH

Convex MPICH is an implementation of version 1.1 of the Message-Passing Interface (MPI) standard and is derived from Argonne National Lab's implementation of MPI. It uses a proprietary low-level device optimized for the Exemplar shared-memory architecture but maintains a standard MPI user interface. Convex MPICH allows programmers to create and run Exemplar applications composed of one or more processes that interact using the MPI communication model. These applications are currently restricted to a single subcomplex. This restriction will be lifted in the future.

The default location on an Exemplar machine for Convex MPICH is the /usr/convex/mpich directory. Example programs are available in the /usr/convex/mpich/examples directory. For information on specific functions, including code examples, refer to the man page for the function in question. Man pages are stored in the directory /usr/convex/mpich/man. For more information on using Convex MPICH, refer to the *Convex MPICH User's Guide for Exemplar Systems* (DSW-493).

PVM/GSM

The PVM/GSM (Parallel Virtual Machine for Globally Shared Memory) message-passing library is a PVM implementation that is finely tuned for running on Exemplar systems. Although PVM/GSM runs only on Exemplar systems, it can work with other networked hosts—if those hosts are each running a PVM daemon that is compatible with Oak Ridge National Lab's version 3.3.10 of PVM.

The default location on an Exemplar machine for PVM/GSM is the /usr/convex/pvm directory. Example programs are available in the /usr/convex/pvm/examples directory. For information on specific functions, including code examples, refer to the man page for the function in question. Man pages are stored in the directory /usr/convex/pvm/man. For more information on using PVM/GSM, refer to the *PVM/GSM User's Guide for Exemplar Systems* (DSW-501).

Message-passing programming vs. shared-memory programming

The message-passing and shared-memory programming paradigms each have advantages and disadvantages. For example, programs that use message passing can be easily ported between architectures. In the Exemplar architecture, these programs benefit from low-latency interconnects that guarantee minimal overhead in parallel synchronization and data distribution. Using message passing, you can access all the virtual memory available on a system (4 Gbytes/process times the number of processes).

However, message-passing code generally requires more software overhead than parallel shared-memory code. Also, message passing is not thread safe; that is, messages can only be safely passed between single-threaded processes. The results of passing messages between processes—where one or more of the processes is multithreaded—are unpredictable. To make message passing thread safe, you must designate one thread in each process to send and receive messages.

The shared-memory style of programming is convenient because the compiler automatically optimizes parallel constructs that can be safely parallelized. In addition, the compiler allows you to manually tune (by using directives, pragmas, or CPSlib functions) those constructs that the compiler cannot parallelize. Also, the compiler's command-line options give you the ability to enable (or disable) many of the parallel optimizations on an individual basis.

However, shared-memory programming is restricted to the 4 Gbytes of virtual memory available to a single process, unless you use the `node_private` memory class. In addition, you must handle the synchronization in some cases. Although the directives, pragmas, and CPSlib functions allow you to get better performance from your programs, their functionality is not directly portable to other vendors' platforms. Other vendor's compilers will, however, ignore the Convex directives and pragmas. Also, the memory classes provided in the shared-memory programming paradigm are unique to Convex.

Given the benefits and drawbacks of the two paradigms, determining whether to use one over the other, or a combination of the two, should be done on a case-by-case basis.

Limits of optimization

8

This chapter discusses common optimization problems you may encounter when developing programs for SPP Series machines and presents some possible solutions. Optimization can remove instructions, replace them, and change the order in which they execute. In some cases, improper optimizations can cause unexpected or incorrect results or code that slows down at higher optimization levels.

In some cases, user error can cause similar problems in code that contains syntactically-correct constructs or directives that are used improperly.

If you encounter any of these problems, look for the following possible causes:

- Aliasing
- False cache line sharing
- Floating-point imprecision (roundoff error)
- Invalid subscripts
- Misused directives and options
- Misused memory classes
- Triangular loops
- Compiler limitations

Note

Compilers perform optimizations assuming that the source code being compiled is valid. Optimizations done on source that violates certain ANSI standard rules can cause the compilers to generate incorrect code.

Aliasing in Fortran

As described in the section "Inhibitors of localization" on page 94, an alias is an alternate name for some object. Fortran EQUIVALENCE statements, C pointers, and procedure calls in both languages can potentially cause aliasing problems. The examples presented in Chapter 3 concern aliasing problems that occur at optimization levels -O2 and above. However, code motion can also cause aliasing problems, at optimization levels -O1 and above.

Consider the following Fortran program:

```
PROGRAM ALIAS
INTEGER I
COMMON /DATA/I
I = 666
CALL CONFUSED(I)
END

SUBROUTINE CONFUSED(N1)
DO I = 1, 2
  N2 = 3 * (N1 + 1)
  CALL CALC(N2)
  WRITE(*,*)'Iteration:', I, ', n1 = ', N1
  WRITE(*,*)'Iteration:', I, ', n2 = ', N2
ENDDO
RETURN
END

SUBROUTINE CALC(N)
INTEGER K, N
COMMON /DATA/ K
K = N + 1
RETURN
END
```

In the subroutine CONFUSED, the compiler assumes that N1 is invariant. This is not true, as N1 is equivalent to I, which is in the DATA common block and therefore is manipulated by the CALC subroutine. The right side of the assignment to N2 appears to be loop invariant, so the compiler moves the assignment to N2 out of the loop at optimization levels -O1 and higher. When compiled at -O1 or above, the program produces incorrect answers.

The results of this program compiled and run at optimization levels -O0 and -O1 follow. Note that the answers are changed at optimization level -O1.

```
% fc -O0 alias.f -o O0.out

% O0.out

Iteration: 1, n1 = 2002
Iteration: 1, n2 = 2001
Iteration: 2, n1 = 6010
Iteration: 2, n2 = 6009

% fc -O1 alias.f -o O1.out

% O1.out

Iteration: 1, n1 = 2002
Iteration: 1, n2 = 2001
Iteration: 2, n1 = 2002
Iteration: 2, n2 = 2001
```

Aliasing in C

Because they frequently use pointers, C programs are especially susceptible to aliasing problems. The C compiler has two different algorithms for detecting potential aliases: a "worst-case" algorithm, which is used in backward-compatible (-pcc) mode, and an ANSI-C aliasing algorithm, which is used in the ANSI-C compatibility modes (-ext, and -std).

Worst-case algorithm

The worst-case aliasing algorithm views all pointer references as subscripts into a giant array that encompasses all of memory. This mythical array (known as *MEM* in the compiler's optimization report) includes all global variables and local variables whose address is taken using the address (&) operator. This results in the following conditions:

- Every pointer reference is a potential alias for every other pointer reference.
- Every pointer reference is a potential alias for every global variable (and vice versa).
- Every pointer reference is a potential alias for every local variable that is used with &.

ANSI algorithm

ANSI C provides stricter type-checking. This allows the C compiler to use a stricter algorithm that eliminates many potential aliases found by the worst-case algorithm. Instead of one giant `*MEM*` array, the ANSI-C algorithm uses a model with separate arrays for each base type (such as `int`, `float`, or `double`). Pointers and variables cannot alias with pointers or variables of a different base type.

The ANSI aliasing algorithm permits the compiler to parallelize the code shown below; the worst-case algorithm does not.

```
void alex1(a, ib, n)
float *a;
int *ib, n;
{
  int i;
  for (i=0; i<n; i++)
    a[i] = ib[i] + n;
}
```

Pointers `a` and `ib` point to different base types. The worst-case algorithm considers them to be aliased through `*MEM*`, but under ANSI C rules, no potential alias exists, and the loop is parallelized.

The ANSI C aliasing algorithm may not be safe if your program is not ANSI compliant. The non-ANSI-compliant code shown below allows two pointers of different base types to become aliased with one another. (The assignment to `fptr` makes the code non-ANSI C compliant.)

```
int array[100];
int *dummy;
float *fptr;

*dummy = array;  /* illegal pointer/integer
                    combination.          */
fptr = dummy;   /* operands of = point to
                   incompatible types.    */
```

Compiling this code generates the warning messages shown in the comments. (Compiling with `-d arg_ptr_ref=e` converts these warnings into error messages.)

Because of the ANSI standard violation, this code does not compile safely under the ANSI aliasing algorithm.

In the following example, function `caller` passes two `long int` pointers to `foo`, which expects one pointer to `long int` and one pointer to `short int`. Because `ia` and `ib` have different base

types, the ANSI C aliasing algorithm assumes that no alias between `*ia` and `*ib` can exist.

At optimization level `-O3` in ANSI mode, the compiler compiles this code (with a warning) and parallelizes the loop even though aliasing and a recurrence do exist. In `-pcc` mode, the code will not compile.

```
void foo(long int *ia, short int *fb)
{
  int i;
  for (i=0; i<500; i++) {
    ia[i] = 1;
    fb[i] = ia[i];
  }
}
caller()
{
  long int arra[600];
  foo(&arra[0], &arra[100]);
}
```

Specifying aliasing modes

To specify an aliasing mode, use one of the following options on the `cc` command line:

- `-alias cautious`
- `-alias standard`
- `-alias worst`

In `cautious` mode, the compiler uses the ANSI C aliasing algorithm. If the compiler finds constructs that could cause hidden aliasing, it switches to the worst-case aliasing algorithm. If you do not specify an aliasing mode, the compiler uses `cautious` mode by default.

In `standard` mode, the compiler always uses the ANSI-C aliasing algorithm.

In `worst` mode, the compiler uses the worst-case aliasing algorithm.

These and other C aliasing options are further discussed in Appendix B, "Optimization options."

Iteration and stop values

Aliasing a variable in an array subscript can make it unsafe for the compiler to parallelize a loop. Below are several situations that can prevent parallelization.

Using potential aliases as addresses of variables

In the following example, the code passes `&j` to `getval`; `getval` can use that address in any number of ways, including possibly assigning it to `iptr`. (Even though `iptr` is not passed to `getval`, `getval` might still access it as a global variable or through another alias.) This situation makes `j` a potential alias for `*iptr`.

```
void subex(iptr, n, j)
int *iptr, n, j;
{
   n = getval(&j,n);

   for (j--; j<n; j++)
      iptr[j] += 1;
}
```

This potential alias means that `j` and `iptr[j]` might occupy the same memory space for some value of `j`. The assignment to `iptr[j]` on that iteration would also change the value of `j` itself. The possible alteration of `j` prevents the compiler from safely parallelizing the loop. In this case, the optimization report says that no induction variable could be found for the loop, and the compiler does not parallelize the loop.

Avoid taking the address of any variable that will be used as the iteration variable for a loop. To parallelize the loop in `subex`, use a temporary variable `i` as shown in the following example:

```
void subex(iptr, n, j)
int *iptr, n, j;
{
   int i;
   n = getval(&j,n);
   i=j;
   for (i--; i<n; i++)
      iptr[i] += 1;
}
```

Using hidden aliases as pointers

In the next example, `ialex` takes the address of `j` and assigns it to `*ip`. Thus, `j` becomes an alias for `*ip` and, potentially, for `*iptr`. Assigned values to `iptr[j]` within the loop could alter the value of `j`. As a result, the compiler cannot use `j` as an induction variable and, without an induction variable, it cannot count the iterations of the loop. When the compiler cannot find the loop's iteration count, the compiler cannot parallelize the loop.

```
int *ip
void ialex(iptr)
int *iptr;{
  int j;
  *ip = &j;
  for (j=0; j<2048; j++)
     iptr[j] = 107;
}
```

To parallelize this loop, remove the line of code that takes the address of `j` or introduce a temporary variable.

Using a pointer as a loop counter

Compiling the following function, the compiler finds that `*j` is not an induction variable (because an assignment to `iptr[*j]` could alter the value of `*j` within the loop.) The compiler does not parallelize the loop.

```
void ialex2(iptr, j, n)
int *iptr;
int *j, n;
{
   for (*j=0; *j<n; (*j)++)
      iptr[*j] = 107;
}
```

Again, this problem can be solved by introducing a temporary iteration variable.

Aliasing stop variables

In the following code, the stop variable n becomes a possible alias for *iptr when &n is passed to foo. This means that n can be altered during the execution of the loop. As a result, the compiler cannot count the number of iterations and cannot parallelize the loop.

```
void salex(int *iptr, int n)
{
   int i;
   foo(&n);
   for (i=0; i < n; i++)
      iptr[i] += iptr[i];
   return;
}
```

To parallelize the affected loop, eliminate the call to foo, move the call below the loop (in which case flow-sensitive analysis takes care of the aliasing), or create a temporary variable as shown below:

```
void salex(int *iptr, int n)
{
   int i, tmp;
   foo(&n);
   tmp = n;
   for (i=0; i < tmp; i++)
      iptr[i] += iptr[i];
   return;
}
```

Because tmp is not aliased to iptr, the loop has a fixed stop value and the compiler parallelizes it.

Global variables

Potential aliases involving global variables cause optimization problems in many programs. The compiler cannot tell whether another function causes a global variable to become aliased.

The following code uses a global variable, n, as a stop value. Because n may have its address taken and assigned to ik outside the scope of the function, n must be considered a potential alias for *ik. The value of n, therefore, can be altered on any iteration of the loop. The compiler cannot determine the stop value and cannot parallelize the loop.

```
int n, *ik;
void foo(int *ik)
{
   int i;

   for (i=0; i<n; i++)
      ik[i]=i;
}
```

Using a temporary local variable solves the problem.

```
int n;
void foo(int *ik)
{
   int i,stop = n;

   for (i=0; i<stop; ++i)
      ik[i]=i;
}
```

If ik is a global variable instead of a pointer, the problem does not occur. Global variables do not cause aliasing problems except when pointers are involved. The following code will parallelize:

```
int n, ik[1000];
void foo()
{
   int i;

   for (i=0; i<n; i++)
      ik[i] = i;
}
```

Limits of optimization

False cache line sharing

False cache line sharing is a form of cache thrashing. It occurs whenever two or more threads in a parallel program are assigning different data items in the same cache line. This type of thrashing can happen on SPP 1000, SPP 1200, and SPP 1600 Series computers. This section discusses how to avoid false cache line sharing by restructuring the data layout and controlling the distribution of loop iterations among threads. To simplify explanations, only Fortran examples are given.

Consider the following example:

```
REAL*4 A(8)
DO I = 1, 8
  A(I) = ...
  .
  .
  .
ENDDO
```

Imagine eight threads, each executing one of the above iterations. Assume that `A(1)` is on a processor cache line (32-byte) boundary so that all eight elements are in the same cache line. Only one thread at a time can "own" the cache line, so not only is the above loop, in effect, run serially, but every assignment by a thread requires an invalidation of the line in the cache of its previous "owner." These problems would likely eliminate any benefit of parallelization.

Now consider this example:

```
REAL*4 B(100,100)
DO I = 1, 100
  DO J = 1, 100
    B(I,J) = ...B(I,J-1)...
  ENDDO
ENDDO
```

Imagine 8 threads working on the `I` loop in parallel (the `J` loop cannot be parallelized because of the dependence). Table 8 shows how the array maps to cache lines, assuming that `B(1,1)` is on a cache line boundary. Array entries that fall on cache line boundaries are in shaded cells.

Table 8 Initial mapping of array to cache lines

1, 1	1, 2	1, 3	1, 4	...	1, 99	1,100
2, 1	2, 2	2, 3	2, 4	...	2, 99	2,100
3, 1	3, 2	3, 3	3, 4	...	3, 99	3,100
4, 1	4, 2	4, 3	4, 4	...	4, 99	4,100
5, 1	5, 2	5, 3	5, 4	...	5, 99	5,100
6, 1	6, 2	6, 3	6, 4	...	6, 99	6,100
7, 1	7, 2	7, 3	7, 4	...	7, 99	7,100
8, 1	8, 2	8, 3	8, 4	...	8, 99	8,100
9, 1	9, 2	9, 3	9, 4	...	9, 99	9,100
10, 1	10, 2	10, 3	10, 4	...	10, 99	10,100
11, 1	11, 2	11, 3	11, 4	...	11, 99	11,100
12, 1	12, 2	12, 3	12, 4	...	12, 99	12,100
13, 1	13, 2	13, 3	13, 4	...	13, 99	13, 100
...	...	...	...	...	...	...
97, 1	97, 2	97, 3	97, 4	...	97, 99	97,100
98, 1	98, 2	98, 3	98, 4	...	98, 99	98,100
99, 1	99, 2	99, 3	99, 4	...	99, 99	99,100
100, 1	100, 2	100, 3	100, 4	...	100, 99	100,100

Array entries in shaded cells are on cache line boundaries.

The Convex compilers, by default, give each thread about the same number of iterations, assigning (if necessary) one extra iteration to the higher-numbered threads until all iterations are assigned to a thread. Table 9 shows the default distribution of the `I` loop across 8 threads.

Table 9 Default distribution of the `I` loop

Thread ID	Iteration range	Number of iterations
0	1-12	12
1	13-24	12
2	25-36	12
3	37-48	12
4	49-61	13
5	62-74	13
6	75-87	13
7	88-100	13

This distribution of iterations causes threads to share cache lines. For example, thread 0 assigns the elements `B(9:12,1)` and thread 1 assigns elements `B(13:16,1)` in the *same* cache line. In fact, every thread shares cache lines with at least one other thread; most share cache lines with two other threads. This type of sharing is called *false* because it is a result of the data layout and the compiler's distribution of iterations; it is *not* inherent in the algorithm itself. Therefore, it can be reduced or even removed by

1. Restructuring the data layout by aligning data on cache line boundaries
2. Controlling the iteration distribution

Aligning data to avoid false sharing

Because false cache line sharing is partially due to the layout of the data, one step in avoiding it is to adjust the layout. Typically, these adjustments are made by aligning data on cache line boundaries. (Aligning arrays generally improves performance; however, it can occasionally decrease performance.) The second step in avoiding false cache line sharing, which is covered in the next section, "Distributing iterations on cache line boundaries," is to adjust the distribution of loop iterations.

Aligning arrays on cache line boundaries

Note the assumption that in the previous example, array B starts on a cache line boundary. The methods below force arrays in Fortran to start on cache line boundaries:

- Using uninitialized COMMON blocks (blocks with no DATA statements) that are at least 64 bytes. These blocks start on 64-byte boundaries.
- Using ALLOCATE statements. These statements return addresses on 64-byte boundaries. (Applies only to parallel executables.)
- Using the command-line option -align cti or the directive ALIGN_CTI. This option/directive forces arrays of any size to start on CTIcache boundaries (64 bytes).
- Using the fc command-line option -align cache. This option forces arrays of any size to start on 32-byte boundaries.

The methods below force arrays in C to start on cache line boundaries:

- Using the functions malloc or memory_class_malloc. These functions return pointers on 64-byte boundaries. (Applies only to parallel executables.)
- Using uninitialized global arrays or structs that are at least 32 bytes. Such arrays and structs are aligned on 64-byte boundaries.
- Using the command-line option -align cti or the pragma align_cti. This option/pragma forces arrays of any size to start on CTIcache boundaries (64 bytes).
- Using uninitialized data of the external storage class in C that is at least 32 bytes. Data is aligned on 64-byte boundaries.

The alignment caused by the -align options and align_cti directive and pragma is performed by the compilers. However, the Convex loader performs all other data alignment. Therefore, you can use non-Convex compilers with the Convex loader and still get some types of the data alignment as described above.

Aligning multidimensional arrays on cache line boundaries

Multidimensional arrays can also be aligned on cache line boundaries. Recall that in the example from the beginning of the section:

```
REAL*4 B(100,100)
DO I = 1, 100
  DO J = 1, 100
    B(I,J) = ...B(I,J-1)...
  ENDDO
ENDDO
```

we assumed `B(1,1)` starts on a cache line boundary. However, because the iteration distribution caused alignment on cache line boundaries to vary from dimension to dimension, performance suffered. Choose a value *x* for the leftmost dimension (rightmost in C) in arrays so that *x* times the data size (in bytes) is an integral multiple of the CTIcache line size. (The code that follows gives an example of this idea.) Using such a value aligns data on CTIcache line boundaries at the same index points in all dimensions.

On SPP 1000, SPP 1200, and SPP1600 Series computers the cache lines sizes are:

- 32 bytes for processor caches
- 64 bytes for CTIcaches (network caches)

For general use, try to align everything on 64-byte boundaries. This alignment will help to eliminate false cache line sharing between processor caches and also between CTIcaches where the penalty for false sharing is even higher. Where possible, try to parameterize cache line size in anticipation of future systems that might have different cache line sizes.

Changing the example so that array B is aligned and padded (to 64 bytes), the leftmost dimension is now 112 instead of 100. (See the modified example below.) The number 112 is the smallest value *x* greater than 100 such that *x* times the data size (4 bytes) is an integral multiple of 64.

```
REAL*4 B(112,100)
COMMON /ALIGNED/ B
  DO I = 1, 100
    DO J = 1, 100
      B(I,J) = ...B(I,J-1)...
    ENDDO
  ENDDO
```

Note that loop limits have not changed. Placing B in a COMMON block has forced B(1,1) onto a cache line boundary (64 bytes), and changing the first dimension to 112 assures that B(1,2), B(1,3),..., B(1,100) all start on cache line boundaries. Table 10 shows how the restructured array maps to (processor) cache lines.

The next section, "Distributing iterations on cache line boundaries," explains how to make the compiler distribute iterations so that threads work on whole cache lines.

Table 10 Restructured mapping of array to cache lines

1, 1	1, 2	1, 3	1, 4	...	1, 99	1,100
2, 1	2, 2	2, 3	2, 4	...	2, 99	2,100
3, 1	3, 2	3, 3	3, 4	...	3, 99	3,100
4, 1	4, 2	4, 3	4, 4	...	4, 99	4,100
5, 1	5, 2	5, 3	5, 4	...	5, 99	5,100
6, 1	6, 2	6, 3	6, 4	...	6, 99	6,100
7, 1	7, 2	7, 3	7, 4	...	7, 99	7,100
8, 1	8, 2	8, 3	8, 4	...	8, 99	8,100
9, 1	9, 2	9, 3	9, 4	...	9, 99	9,100
10, 1	10, 2	10, 3	10, 4	...	10, 99	10,100
11, 1	11, 2	11, 3	11, 4	...	11, 99	11,100
12, 1	12, 2	12, 3	12, 4	...	12, 99	12,100
...	...	...	...	...	...	...
97, 1	97, 2	97, 3	97, 4	...	97, 99	97,100
98, 1	98, 2	98, 3	98, 4	...	98, 99	98,100
99, 1	99, 2	99, 3	99, 4	...	99, 99	99,100
100, 1	100, 2	100, 3	100, 4	...	100, 99	100,100
101, 1	101, 2	101, 3	101, 4	...	101, 99	101,101
102, 1	102, 2	102, 3	102, 4	...	102, 99	102,102
103, 1	103, 2	103, 3	103, 4	...	103, 99	103,103
104, 1	104, 2	104, 3	104, 4	...	104, 99	104,104
105, 1	105, 2	105, 3	105, 4	...	105, 99	105,105
106, 1	106, 2	106, 3	106, 4	...	106, 99	106,106
107, 1	107, 2	107, 3	107, 4	...	107, 99	107,107
108, 1	108, 2	108, 3	108, 4	...	108, 99	108,108
109, 1	109, 2	109, 3	109, 4	...	109, 99	109,109
110, 1	110, 2	110, 3	110, 4	...	110, 99	110,110
111, 1	111, 2	111, 3	111, 4	...	111, 99	111,112
112, 1	112, 2	112, 3	112, 4	...	112, 99	112,112

Array entries in shaded cells are on cache line boundaries.

Distributing iterations on cache line boundaries

Recall that the default iteration distribution causes thread 0 to work on iterations 1-12 and thread 1 to work on iterations 13-24, and so on. Even though the cache lines are aligned across the columns of the array (see Figure 10), we still need to change the iteration distribution. Use CHUNK_SIZE to change the distribution:

```
      REAL*4 B(112,100)
      COMMON /ALIGNED/ B
C$DIR PREFER_PARALLEL (CHUNK_SIZE=16)
      DO I = 1, 100
        DO J = 1, 100
          B(I,J) = ...B(I,J-1)...
        ENDDO
      ENDDO
```

You must specify a constant CHUNK_SIZE. However, the ideal would be to distribute work such that all but one thread works on the same number of whole cache lines, and the remaining thread works on any partial cache line. For example, given:

```
NITS  = number of iterations
NTHDS = number of threads
LSIZE = CTIcache line size in words (16 for 4-byte data;
          8 for 8-byte data;
          4 for 16-byte data)
```

the ideal CHUNK_SIZE would be:

```
CHUNK_SIZE = LSIZE * (1 + ( (1 + (NITS - 1) / LSIZE ) - 1 ) / NTHDS)
```

For the code above, these numbers are:

```
NITS  = 100
NTHDS = 8
LSIZE = 16 (aligns on CTIcache boundaries for 4-byte data)
```

```
CHUNK_SIZE = 16 * (1 + ( (1 + (100 - 1) / 16 ) - 1) / 8)
           = 16 * (1 + ( (1 +  6             ) - 1) / 8)
           = 16 * (1 + ( 6                        ) / 8)
           = 16 * (1 + 0                               )
           = 16
```

CHUNK_SIZE = 16 causes threads 0, 1, ..., 6 to execute iterations 1-16, 17-32, ..., 81-96, respectively. Thread 7 executes iterations 97-100. As a result there is no false cache line sharing, and parallel performance is greatly improved.

While you cannot specify the ideal CHUNK_SIZE for every loop, using

CHUNK_SIZE = *x*

where *x* times the data size (in bytes) is an integral multiple of 64 will eliminate false cache line sharing if the arrays are already properly aligned, as discussed earlier in this section. (The number 64 is used because the CTIcache line size is 64 bytes.)

Thread-specific array elements

Sometimes a parallel loop has each thread update a unique element of a shared array which is further processed by thread 0 outside the loop.

Consider the following Fortran example:

```
      REAL*4 S(8)
C$DIR LOOP_PARALLEL
      DO I = 1, N
         .
         .
         .
         S(MY_THREAD()) = ...     ! EACH THREAD ASSIGNS ONE ELEMENT OF S
         .
         .
         .
      ENDDO
C$DIR NO_PARALLEL
      DO J = 1, NUM_THREADS()
         = ...S(J)                ! THREAD 0 POST-PROCESSES S
      ENDDO
```

The problem here is that potentially all the elements of `S` are in a single cache line, so the assignments cause false sharing. One approach is to change the code to force the unique elements into different cache lines, as indicated below:

```
      REAL*4 S(8,8)
C$DIR LOOP_PARALLEL
      DO I = 1, N
        .
        .
        .
        S(1,MY_THREAD()) = ...   ! EACH THREAD ASSIGNS ONE ELEMENT OF S
        .
        .
        .
      ENDDO
C$DIR NO_PARALLEL
      DO J = 1, NUM_THREADS()
        = ...S(1,J)              ! THREAD 0 POST-PROCESSES S
      ENDDO
```

For multihypernode applications the dimensions should be `S(16,`*number_of_threads*`)` because the CTIcache line size is 64 bytes, and the data size is 4 bytes.

Limits of optimization

Scalars sharing a cache line

Sometimes parallel tasks will assign unique scalar variables that are in the same cache line, as in the following example:

```
      COMMON /RESULTS/ SUM, PRODUCT
C$DIR BEGIN_TASKS
      DO I = 1, N
        .
        .
        .
        SUM = SUM + ...
        .
        .
        .
      ENDDO
C$DIR NEXT_TASK
      DO J = 1, M
        .
        .
        .
        PRODUCT = PRODUCT * ...
        .
        .
        .
      ENDDO
C$DIR END_TASKS
```

This problem is similar to the example in the previous section ("Thread-specific array elements"), and can be avoided by padding enough space between the two scalar variables so that the variables are in separate CTIcache lines:

```
COMMON /RESULTS/ SUM, PAD(15), PRODUCT
```

where `PAD(15)` represents 60 bytes—`SUM` and `PAD(15)` together take up 64 bytes, forcing `PRODUCT` into the next CTIcache line. (`PAD(7)` would be adequate for single-node applications because the processor cache line size is 32 bytes.)

Working with unaligned arrays

The most common cache-thrashing complication using arrays and loops will be that arrays assigned within a loop are unaligned (and possibly unalignable) with each other. There are several possible causes for this:

- Arrays that are local to a routine are allocated on the stack. You must use `-align cti` or `-align cache` (available only in Fortran) to align these arrays.
- Array dummy arguments might be passed an element other than the first in the actual argument.
- Array elements might be assigned with different offset indexes.

Consider the following Fortran example:

```
COMMON /OKAY/ X(112,100)
        ...
CALL UNALIGNED (X(I,J))
        ...
SUBROUTINE UNALIGNED (Y)
REAL*4 Y(*)
    ! Y(1) PROBABLY NOT ON A CACHE LINE BOUNDARY
```

The address of `Y(1)` is unknown. However, if elements of `Y` are heavily assigned in this routine, it may be worthwhile to compute an alignment, given by the following formula:

```
LREM = LSIZE - ( ( MOD ( LOC(Y(1))-4, LSIZE*x) + 4) /x)
```

where

`LSIZE`
: is the appropriate cache line size in words

x
: is the data size for elements of `Y`

For this case, assume it is CTIcache line size (64 bytes) in single precision words (16 words). Note that

```
( ( MOD ( LOC(Y(1))-4, LSIZE*4) + 4) /4)
```

returns a value in the set 1, 2, 3, ..., `LSIZE`, so `LREM` is in the range 0 to 15.

Then an original loop such as:

```
DO I = 1, N
  Y(I) = ...
ENDDO
```

can be transformed to:

```
C$DIR NO_PARALLEL
      DO I = 1, MIN (LREM, N) ! 0 <= LREM < 16
        Y(I) = ...
      ENDDO
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 16)
      DO I = LREM+1, N
         ! Y(LREM+1) IS ON A CACHE LINE BOUNDARY
        Y(I) = ...
      ENDDO
```

The first loop takes care of elements from the first (if any) partial cache line of data. The second loop begins on a cache line boundary, and can be controlled with `CHUNK_SIZE` to avoid false sharing among the threads.

Working with dependences

Data dependences in loops prevent parallelization and prevent the elimination of false cache line sharing. If certain conditions are met, some performance gains can be achieved. For example, consider the following code:

```
COMMON /ALIGNED / P(128,128), Q(128,128), R(128,128)
REAL*4 P, Q, R
DO J =  2, 128
  DO I = 2, 127
    P(I-1,J) = SQRT (P(I-1,J-1) + 1./3.)
    Q(I   ,J) = SQRT (Q(I   ,J-1) + 1./3.)
    R(I+1,J) = SQRT (R(I+1,J-1) + 1./3.)
  ENDDO
ENDDO
```

Only the I loop can be parallelized. (Because of the dependence in the J loop, it cannot be parallelized.) It is impossible to distribute the iterations such that there will be no false cache line sharing in the above loop. If all loops that refer to these arrays always use the same offsets (which is unlikely) then you could make dimension adjustments that would allow a better iteration distribution. For example, the following would work well for 8 threads:

```
      COMMON /ADJUSTED/ P(128,128), PAD1(15), Q(128,128),
     > PAD2(15), R(128,128)

      DO J =  2, 128
C$DIR   PREFER_PARALLEL (CHUNK_SIZE=16)
        DO I = 2, 127
          P(I-1,J) = SQRT (P(I-1,J-1) + 1./3.)
          Q(I  ,J) = SQRT (Q(I  ,J-1) + 1./3.)
          R(I+1,J) = SQRT (R(I+1,J-1) + 1./3.)
        ENDDO
      ENDDO
```

Padding 60 bytes before the declarations of both Q and R causes the P(1,J), Q(2,J), and R(3,J) to be aligned on 64-byte boundaries for all J. Combined with a CHUNK_SIZE of 16, this causes threads to assign data to unique whole cache lines.

Often in real-world code you will find a mix of all the above problems in some CPU-intensive loops. You will not be able to avoid all false cache line sharing, but by careful inspection of the problems and careful application of some of the workarounds shown here, you will usually be able to significantly enhance performance of your parallel loops.

Floating-point imprecision

The compiler applies normal arithmetic rules to real numbers. It assumes that two arithmetically equivalent expressions produce the same numerical result.

Most real numbers cannot be represented exactly in digital computers. Instead, these numbers are rounded to a floating-point value that can be represented. When optimization changes the evaluation order of a floating-point expression, the results can change. Possible consequences of floating-point roundoff include program aborts, division by zero, address errors, and incorrect results.

In any parallel program, the execution order of the instructions will differ from the serial version of the same program. This can cause noticeable roundoff differences between the two versions. Running a parallel code under different machine configurations or conditions can also yield roundoff differences, because the

execution order can differ under differing machine conditions, causing roundoff errors to propagate in different orders between executions. Accumulator variables are especially susceptible to these problems.

Consider the following Fortran example:

```
C$DIR GATE(ACCUM_LOCK)
      LK = ALLOC_GATE(ACCUM_LOCK)
      .
      .
      .
      LK = UNLOCK_GATE(ACCUM_LOCK)
C$DIR BEGIN_TASKS, TASK_PRIVATE(I)
      CALL COMPUTE(A)
C$DIR CRITICAL_SECTION(ACCUM_LOCK)
      ACCUM = ACCUM + A
C$DIR END_CRITICAL_SECTION
C$DIR NEXT_TASK
      DO I = 1, 10000
        B(I) = FUNC(I)
C$DIR   CRITICAL_SECTION(ACCUM_LOCK)
        ACCUM = ACCUM + B(I)
C$DIR   END_CRITICAL_SECTION
        .
        .
        .
      ENDDO

C$DIR NEXT_TASK
      DO I = 1, 10000
        X = C(I) + D(I)
      ENDDO
C$DIR CRITICAL_SECTION(ACCUM_LOCK)
      ACCUM = ACCUM/X
C$DIR END_CRITICAL_SECTION
C$DIR END_TASKS
```

Here, three parallel tasks are all manipulating the real variable ACCUM, using real variables which have themselves been manipulated. Each manipulation is subject to roundoff error, so the total roundoff error here might be substantial. When the program runs in serial, the tasks execute in their written order, and the roundoff errors accumulate in that order. However, if the tasks run in parallel, there is no guarantee as to what order the tasks will run in, meaning the roundoff error will accumulate in a different order than it does during the serial run. Depending on machine conditions, the tasks may run in different orders during

different parallel runs also, potentially accumulating roundoff errors differently and yielding different answers.

An analogous C example follows:

```
static gate_t accum_lock;
lk = alloc_gate(&accum_lock);
.
.
.
lk = unlock_gate(&accum_lock);
#pragma _CNX begin_tasks, task_private(i)
compute(a);
#pragma _CNX critical_section(accum_lock)
accum = accum + a;
#pragma _CNX end_critical_section
#pragma _CNX next_task
for(i=0;i<10000;i++) {
  b[i] = func[i];
#pragma _CNX critical_section(accum_lock)
  accum = accum + b[i];
#pragma _CNX end_critical_section
.
.
.
}
#pragma _CNX next_task
for(i=0;i<10000;i++)
  x = c[i] + d[i];
#pragma _CNX critical_section(accum_lock)
accum = accum/x;
#pragma _CNX end_critical_section
#pragma _CNX end_tasks
```

Problems with floating-point precision can also occur when a program tests the value of a variable without allowing enough tolerance for roundoff errors. To solve the problem, adjust the tolerances to allow for greater roundoff errors or declare the variables to be of a higher precision (use the `double` type instead of `float` in C, or `REAL*8` rather than `REAL*4` in Fortran). It is always poor practice to test floating point numbers for exact equality.

Disabling underflow traps

By default, PA-RISC processor hardware traps floating point underflow when a floating point result is *tiny*. A floating point number is considered tiny if its exponent field is zero but its mantissa is nonzero (for more information, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*). This practice is extremely costly in terms of execution time and seldom provides any benefit. You can disable this behavior by passing the `+FPD` flag to the loader. This is done using the `-W` compiler option on either the `cc` or `fc` command line. The following example shows such an `fc` command line:

```
%fc -Wl,+FPD prog.f
```

This command line compiles the program `prog.f` and instructs the loader to disable floating point underflow. For more information on the `-W` option, refer to the *Fortran User's Guide* or the *C User's Guide*.

Invalid subscripts

An array reference in which *any* subscript falls outside declared bounds for that dimension is called an invalid subscript. Invalid subscripts are a common cause of answers that vary between optimization levels and programs that abort and dump core. Use the `fc` command-line option `-cs` (check subscripts) to check that each subscript is within its array bounds. See the fc(1) man page for more information.

Misused directives, pragmas, and options

Misused directives and pragmas are a common cause of wrong answers. For example, forcing parallelization of a loop containing a call is safe only if the called routine contains no dependences.

Do not assume that it is always safe to parallelize a loop whose data is safe to localize. You can safely localize loop data in loops that do not contain a loop-carried dependence (LCD) of the form shown in the following Fortran loop:

```
DO I = 2, M
  DO J = 1, N
    A(I,J) = A(I+IADD,J+JADD) + B(I,J)
  ENDDO
ENDDO
```

where one of IADD and JADD is negative and the other is positive. This is explained in detail in the section "Inhibitors of localization" on page 94.

You cannot safely parallelize a loop that contains any kind of LCD. This is discussed further in the section "Inhibitors of parallelization" on page 118.

The MAIN section of the Fortran program that follows initializes A, calls CALC, and outputs the new array values. In subroutine CALC, the indirect index used in A(IN(I)) introduces a potential dependence that prevents the compiler from parallelizing CALC's I loop.

```
PROGRAM MAIN
REAL A(1025)
INTEGER IN(1025)
COMMON /DATA/ A
DO I = 1, 1025
  IN(I) = I
ENDDO
CALL CALC(IN)
CALL OUTPUT(A)
END

SUBROUTINE CALC(IN)
INTEGER IN(1025)
REAL A(1025)
COMMON /DATA/ A
DO I = 1, 1025
  A(I) = A(IN(I))
ENDDO
RETURN
END
```

An analogous C example follows:

```
float arra[1025];

void calc(int in[])
{
  int i,j;

    for(i = 0; i <= 1024; i++)
      arra[i] = arra[in[i]];
}
main()
{
  int i,j,in[1025];

  for(i = 0; i <= 1024; i++)
    in[i] = i;
  calc(in);
  output(arra);
}
```

Because you know that `IN(I) = I`, you can use the `NO_LOOP_DEPENDENCE` directive, as shown below. This directive allows the compiler to ignore the apparent dependence and parallelize the loop at optimization level `-O3`.

```
      SUBROUTINE CALC(IN)
      INTEGER IN(1025)
      REAL A(1025)
      COMMON /DATA/ A
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, 1025
         A(I) = A(IN(I))
      ENDDO
      RETURN
      END
```

In C:

```
void calc(int in[])
{
  int i,j;
  #pragma _CNX no_loop_dependence(arra)
  for(i = 0; i <= 1024; i++)
    arra[i] = arra[in[i]]
}
```

Misused memory classes

While manually assigned memory classes can substantially boost performance when coupled with manual parallelization, assigning the wrong memory class to data can cause wrong answers and in some cases degrade performance. This section discusses some common misuses of memory classes.

Improper dynamic allocations

Dynamically allocating `thread_private` memory from serial code can give unexpected results if the memory is later accessed from parallel code. Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
      REAL*8 WRONGTP(:)
C$DIR THREAD_PRIVATE(WRONGTP)
      ALLOCATABLE WRONGTP
      .
      .
      .
C THE FOLLOWING ALLOCATE ONLY ALLOCATES
C WRONGTP(N) FOR THREAD 0:
      ALLOCATE(WRONGTP(N))
C$DIR LOOP_PARALLEL(THREADS, IVAR = I)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, NUM_THREADS()
        DO J = 1, N
          WRONGTP(J) = ... ! ONLY EXISTS FOR
          .                ! THREAD 0
          .
          .
        ENDDO
      ENDDO
```

Here, the array `WRONGTP` is allocated, but since the allocation takes place in serial code, which is run by thread 0, only thread 0 allocates the array. When other threads attempt to access the array in the `J` loop, it does not exist. To fix this, allocate the array inside the thread-parallel `I` loop, as discussed in Chapter 5, "Memory classes."

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!!                                  */
static thread_private double *wrongtp;
.
.
.
/* the following memory_class_malloc only allocates wrongtp
   for thread 0                                                  */
  wrongtp=(double *)memory_class_malloc(sizeof(double)*n,
                                       THREAD_PRIVATE_MEM);
#pragma _CNX loop_parallel(threads, ivar = i)
#pragma _CNX loop_private(j)
for(i=0;i<num_threads();i++) {
  for(j=0;j<n;j++) {
    wrongtp[j] = ... /* only exists for thread 0 */
    .
    .
    .
  }
}
```

In general, memory of classes other than `thread_private` should be dynamically allocated in serial code. Allocating `node_private`, `near_shared`, `far_shared` and `block_shared` memory from within parallel code will create wasteful redundant copies. Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!
      REAL*8 WRONGNP(:)
C$DIR NODE_PRIVATE(WRONGNP)
C$DIR FAR_SHARED_POINTER(WRONGNP)
      ALLOCATABLE(WRONGNP)
      .
      .
      .
      N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES, IVAR = I)
      DO I = 1, N
        ALLOCATE(WRONGNP(M))
        .
        .
        .
      ENDDO
```

Recall from Chapter 5, "Memory classes," that when a node_private array is allocated, a physical copy is created on each hypernode on which the program is running. Here, each loop iteration executes the ALLOCATE statement (or memory_class_malloc function in C), thus allocating N copies of the array. This is N-1 times more copies than are actually needed. To further complicate things, node_private arrays manipulated in parallel code must be accessed by shared pointers, which is why the Fortran example includes a far_shared_pointer statement. In the code above, this pointer would be overwritten every time the I loop executed the ALLOCATE statement (or memory_class_malloc function in C), meaning that only the final copy allocated would be accessible. Since the hypernodes' execution of the loop code is not perfectly synchronized, the actual memory accessed by WRONGNP(I) would vary depending on which hypernode was last to perform the allocation.

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!!                          */
static far_shared double *wrongnp;
.
.
.
n = numnodes();
#pragma _CNX loop_parallel(nodes, ivar = i)
for(i=0;i<n;i++) {
  wrongnp = (double *)memory_class_malloc(sizeof(double)*m,
                                          NODE_PRIVATE_MEM);
   .
   .
   .
}
```

While dynamically allocated near_shared, far_shared and block_shared arrays do not normally require special pointer types, they suffer from the same redundant-copy problem. Allocating any shared-memory arrays from within parallel code will create as many copies of the data as there are hypernodes (or threads) executing the ALLOCATE (or memory_class_malloc) statement. As with the node_private example above, the actual memory accessed will depend on which hypernode most recently executed the ALLOCATE statement. After all hypernodes have executed the ALLOCATE, the memory allocated by all but the last will be lost. Such lost arrays are not only unusable, they cannot be deallocated.

To avoid such redundancy problems, follow the allocation examples discussed in Chapter 5, "Memory classes," and only allocate memory from within parallel constructs as described there.

Incorrect array pointers

As mentioned in the previous section, sometimes it is necessary to access dynamically allocated arrays using pointers of different memory classes. For example, when accessing `node_private` arrays from node-parallel code, `far_shared` pointers must be used (refer to Chapter 5, "Memory classes"). Failing to do this will render the copies of the arrays on all but hypernode 0 inaccessible. Consider the following *incorrect* Fortran example:

```
C INCORRECT EXAMPLE FOLLOWS!!!!
      REAL*8 NONP(:)
C$DIR NODE_PRIVATE(NONP)
      ALLOCATABLE(NONP)
      .
      .
      .
      ALLOCATE(NONP(M))
      N = NUM_NODES
C$DIR LOOP_PARALLEL(NODES), LOOP_PRIVATE(J)
      DO I = 1, N
C$DIR LOOP_PARALLEL(THREADS)
        DO J = 1, M
          NONP(J) = ...
          .
          .
          .
        ENDDO
      ENDDO
```

While the `NONP` array is correctly allocated in serial code here, it is not explicitly given a shared pointer, so the arrays created will be accessed by the default `node_private` pointer. A physical copy of `NONP` will be created on every hypernode, but the `node_private` pointer by which these copies are accessed will only be initialized on hypernode 0, because it is the only hypernode executing the `ALLOCATE` statement (or `memory_class_malloc` in C). The contents of the (`node_private`) pointers on other hypernodes are uninitialized and therefore indeterminate. When, in the hypernode-parallel `J` loop, these other hypernodes attempt to access `NONP`, they will do so using the garbage contents of their uninitialized pointers, typically causing a runtime error.

An analogous C example follows:

```
/* INCORRECT EXAMPLE FOLLOWS!!!                              */
static node_private double *n0np;
.
.
.
n0np = (double *)memory_class_malloc(sizeof(double)*m,
                                     NODE_PRIVATE_MEM);
n = numnodes();
#pragma _CNX loop_parallel(nodes, ivar = i), loop_private(j)
for(i=0;i<n;i++) {
#pragma _CNX loop_parallel(threads, ivar = j)
  for(j=0;j<m;j++) {
    n0np[j] = ...
    .
    .
    .
  }
}
```

Chapter 5, "Memory classes," covers correct pointer/data combinations, and explains the situations in which non-default pointers should be used. To avoid uninitialized pointer problems such as the one described above, follow the recommendations of Chapter 5 carefully.

Hidden dependences

Improperly accessing a shared variable from parallel threads can create an unapparent dependence that can cause wrong answers.

Consider the following Fortran code:

```
      PROGRAM HOLDER
      REAL HOLD
C$DIR FAR_SHARED(HOLD)
C$DIR TASK_PRIVATE(X,Y)
C$DIR BEGIN_TASKS
      X = ...
      .
      .
      .
      CALL ADDHOLD(HOLD, X)
C$DIR NEXT_TASK
      Y = ...
      .
      .
      .
      CALL ADDHOLD(HOLD,Y)
C$DIR END_TASKS
      END

      SUBROUTINE ADDHOLD(HOLD,Z)
      REAL HOLD, Z
      HOLD = HOLD+Z
      END
```

Here, the `far_shared` variable HOLD is updated as a function of itself in the subroutine ADDHOLD, which is called from the potentially parallel tasks. If HOLD was updated within the tasks rather than in a subroutine, the dependence would be more obvious to the programmer, who may not have ready access to ADDHOLD's source.

Isolating the assignment to HOLD inside a critical section would allow the tasks to safely parallelize, whether the assignment took place in a subroutine or inside the tasks themselves.

An analogous C example follows:

```
void addhold(float *hold, float z) {
  *hold = *hold + z;
}
main() {
  static far_shared float hold;
  static task_private float x,y;
  #pragma _CNX begin_tasks
  x = ...;
  .
  .
  .
  addhold(&hold,x);
  #pragma _CNX next_task
  y = ...;
  .
  .
  .
  addhold(&hold,y);
  #pragma _CNX end_tasks
}
```

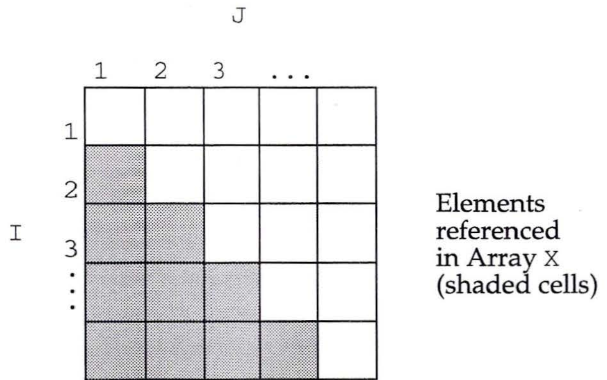
Always use caution when parallelizing a call to a procedure that passes the same shared variable from every thread.

Triangular loops

A *triangular loop* is a loop nest with an inner loop whose upper or lower bound (but not both) is a function of the outer loop's index. Examples of a lower triangular loop and an upper triangular loop are given below. (To simplify explanations, only Fortran examples are given in this section.)

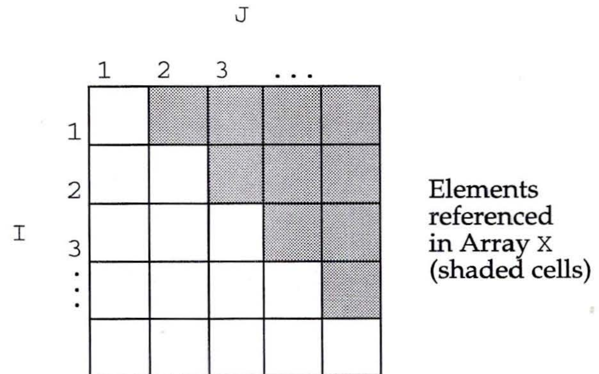
Lower triangular loop

```
DO J = 1, N
  DO I = J+1, N
    F(I) = F(I) + ... + X(I,J) + ...
```



Upper triangular loop

```
DO J = 1, N
  DO I = 1, J-1
    F(I) = F(I) + ... + X(I,J) + ...
```



While the compiler can usually auto-parallelize one of the outer or inner loops, there are typically performance problems in either case:

- If the outer loop is parallelized by assigning contiguous chunks of iterations to each of the threads, the load will be severely imbalanced. For example, in the lower triangular example above, the thread doing the last chunk of iterations does far less work than the thread doing the first chunk.
- If the inner loop is auto-parallelized, then on each outer iteration in the `J` loop, the threads are assigned to work on a different set of iterations in the `I` loop, thus losing access to some of their previously encached elements of `F` and thrashing each other's caches in the process.

By manually controlling the parallelization, you can greatly improve the performance of a triangular loop. Parallelizing the outer loop is generally more beneficial than parallelizing the inner loop. The next two sections ("Parallelizing the outer loop" and "Parallelizing the inner loop") explain how to achieve the enhanced performance.

Parallelizing the outer loop

Using directives you can control the parallelization of the outer loop in a triangular loop to optimize the performance of the loop nest.

For the outer loop, it is preferable to assign iterations to threads in a more balanced manner. The simplest method is to assign the threads one at a time using the attribute CHUNK_SIZE:

```
C$DIR PREFER_PARALLEL (CHUNK_SIZE = 1)
      DO J = 1, N
        DO I = J+1, N
          Y(I,J) = Y(I,J) + ...X(I,J)...
```

This causes each thread to execute in the following manner:

```
      DO J = MY_THREAD() + 1, N, NUM_THREADS()
        DO I = J+1, N
          Y(I,J) = Y(I,J) + ...X(I,J)...
```

where 0 <= MY_THREAD() < NUM_THREADS()

In this case, the first thread still does more work than the last, but the imbalance is greatly reduced. For example, assume N = 128 and there are 8 threads. Then the default parallel compilation would cause thread 0 to do J = 1 to 16, resulting in 1912 inner iterations, whereas thread 7 does J = 113 to 128, resulting in 120 inner iterations. With chunk_size = 1, thread 0 does 1072 inner iterations, and thread 7 does 1023.

The load can be balanced even more by unrolling the outer loop once. Each thread then executes in the following manner:

```
DO J = MY_THREAD() + 1, N, 2*NUM_THREADS()
  DO I = J+1, N
    Y(I,J) = Y(I,J) + ...X(I,J)...
      DO I=1+2*NUM_THREADS()-J,N
        Y(I,J) = Y(I,J)+...X(I,J)...
```

Parallelizing the inner loop

If the outer loop cannot be parallelized, parallelize the inner loop if possible. There are two issues to be aware of when parallelizing the inner loop:

- Cache thrashing

  Consider the parallelization of the following inner loop:

  ```
  DO J = I+1, N
    F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
  ```

  where `I` varies in the outer loop iteration.

  The default iteration distribution has each thread processing a contiguous chunk of iterations of approximately the same number as every other thread. The amount of work per thread is about the same; however, from one outer iteration to the next, threads work on different elements in `F`, resulting in cache thrashing.

- The overhead of parallelization

  If the loop cannot be interchanged to be outermost (or at least outermore), then the overhead of parallelization is compounded by the number of outer loop iterations.

Below is a scheme that assigns "ownership" of elements to threads on a cache line basis so that threads always work on the same cache lines and retain data locality from one iteration to the next. In addition, the `parallel` directive (see the section "`parallel [(`*attribute_list*`)]`" on page 325) is used to spawn threads just once. The outer, nonparallel loop is replicated on all processors, and the inner loop iterations are manually distributed to the threads.

Limits of optimization

```
C F IS KNOWN TO BEGIN ON A CACHE LINE BOUNDARY
      NTHD = NUM_THREADS()
      CHUNK = 8                   ! CHUNK * DATA SIZE (4 BYTES)
                                  !    EQUALS PROCESSOR CACHE LINE SIZE;
                                  !    A SINGLE THREAD WORKS ON CHUNK = 8
                                  !    ITERATIONS AT A TIME
      NTCHUNK = NTHD * CHUNK      ! A CHUNK TO BE SPLIT AMONG THE THREADS
      ...
C$DIR PARALLEL,PARALLEL_PRIVATE(ID,JS,JJ,J,I)
      ID = MY_THREAD() + 1       ! UNIQUE THREAD ID
      DO I = 1, N
        JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
     >         + (ID-1) * CHUNK + 1
        DO JJ = JS, N, NTCHUNK
          DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
            F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
          ENDDO
        ENDDO
      ENDDO
C$DIR END_PARALLEL
```

The idea is to assign a fixed ownership of cache lines of F and to assign a distribution of those cache lines to threads that keeps as many threads busy computing whole cache lines for as long as possible. Using CHUNK = 8 for 4-byte data makes each thread work on 8 iterations covering a total of 32 bytes—the processor cache line size. In general, set CHUNK equal to the smallest value that multiplies by the data size to give a multiple of 32 or 64 (the processor cache line size and the CTIcache line size, respectively). Smaller values of CHUNK keep most threads busy most of the time; however, setting CHUNK to obtain a multiple of 64 is better if the application is executing on more than one hypernode and consequently using the CTIcache.

When, because of the ever-decreasing work in the triangular loop, there are fewer cache lines left to compute than there are threads, threads gracefully drop out until there is only one thread left to compute those iterations associated with the last cache line. Compare this distribution to the default distribution that causes false cache line sharing (see the section "False cache line sharing" in this chapter) and consequent thrashing when all threads attempt to compute data into a few cache lines.

The scheme above maps a sequence of NTCHUNK-sized blocks over the F array. Within each block, each thread owns a specific cache line of data. The relationship between data, threads, and blocks of size NTCHUNK is shown in Figure 25.

NTCHUNK 1

CHUNKs of F	Associated thread
F(1) ... F(8)	thread 0
F(9) ... F(16)	thread 1
F(17) ... F(24)	thread 2
F(33) ... F(40)	thread 4
F(41) ... F(48)	thread 5
F(25) ... F(32)	thread 3
F(49) ... F(56)	thread 6
F(57) ... F(64)	thread 7

NTCHUNK 2

CHUNKs of F	Associated thread
F(65) ... F(72)	thread 0
F(73) ... F(80)	thread 1
F(81) ...	...

Figure 25 Data ownership by CHUNK and NTCHUNK blocks

CHUNK is the number of iterations a thread will work on at one time. The idea is to make a thread work on the same elements of F from one iteration of I to the next (except for those that are already complete). The scheme above causes thread 0 to do all work associated with the cache lines starting at F(1), F(1+NTCHUNK), F(1+2*NTCHUNK), and so on. Likewise, thread 1 does the work associated with the cache lines starting at F(9), F(9+NTCHUNK), F(9+2*NTCHUNK), and so on. Thus, if a thread assigns certain elements of F for I = 2, then it is certain that the same thread encached those elements of F in iteration I = 1, thus eliminating cache thrashing among the threads.

Examining the code

Having established the idea of assigning cache line ownership, consider the following Fortran example in more detail:

```
C$DIR PARALLEL,PARALLEL_PRIVATE(ID,JS,JJ,J,I)
        ID = MY_THREAD() + 1    ! UNIQUE THREAD ID
        DO I = 1, N
          JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
     >          + (ID-1) * CHUNK + 1
          DO JJ = JS, N, NTCHUNK
            DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)
              F(J) = F(J) + SQRT(A(J)**2 - B(I)**2)
            ENDDO
          ENDDO
        ENDDO
C$DIR END_PARALLEL
```

```
C$DIR PARALLEL, PARALLEL_PRIVATE(ID,JS,JJ,J,I)
```

The PARALLEL directive spawns threads, each of which begins executing the statements in the parallel region. Each thread has a private version of the variables ID, JS, JJ, J, and I.

```
ID = MY_THREAD() + 1    ! UNIQUE THREAD ID
```

This establishes a unique ID for each thread, in the range 1 to NUM_THREADS().

```
DO I = 1, N
```

All threads execute the I loop redundantly (instead of thread 0 executing it alone.)

```
JS = ((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
      + (ID-1) * CHUNK + 1
```

For a given value of I+1, the above line determines in which NTCHUNK the value I+1 falls, then assigns a unique CHUNK of it to each thread ID. Suppose that there are *ntc* NTCHUNKs, where *ntc* is approximately N/NTCHUNK. Then the expression:

```
(I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK)
```

returns a value in the range 1 to *ntc* for a given value of I+1. Then the expression:

```
((I+1 + NTCHUNK-1 - ID*CHUNK ) / NTCHUNK) * NTCHUNK
```

identifies the start of an NTCHUNK that contains I+1 or is immediately above I+1 for a given value of ID. (For the NTCHUNK that contains I+1, if the cache lines owned by a thread either contain I+1 or are above I+1 in memory, this expression returns this NTCHUNK. If the cache lines owned by a thread are below I+1 in this NTCHUNK, this expression

returns the next highest NTCHUNK. In other words, if there is no work for a particular thread to do in this NTCHUNK, then start working in the next one.)

`(ID-1) * CHUNK + 1`

identifies the start of the particular cache line for the thread to compute within this NTCHUNK.

`DO JJ = JS, N, NTCHUNK`

Each thread does a unique set of cache lines starting at its specific JS and continuing into succeeding NTCHUNKs until all the work is done.

`DO J = MAX (JJ, I+1), MIN (N, JJ+CHUNK-1)`

This does the work within a single cache line. If the starting index (I+1) is greater than the first element in the cache line (JS) then start with I+1. If the ending index (N) is less than the last element in the cache line, then finish with N.

More generally, notice that:

- Most of the "complicated" arithmetic is in outer loop iterations.
- Divides could be replaced, by the programmer, with shift instructions because they involve powers of two.
- If this application were to be run on a multihypernode subcomplex, choosing a chunk size of 16 for 4-byte data (using 64 bytes, the CTIcache line size) would be appropriate.

Compiler limitations

Compiler limitations can produce faulty optimized code when the source code contains:

- Reductions
- Different possible evaluation orderings
- Iterations by zero
- Nondeterminism of parallel execution
- Replaceable loop test variables
- Trip counts greater than $2^{31} - 1$ at optimization levels `-O2` and `-O3`
- Hidden ordered sections

Descriptions of, and methods for, avoiding the items listed above are in the following sections.

Reductions

Reductions are a special class of dependence that the compiler can parallelize. An apparent LCD can prevent the compiler from parallelizing a loop containing a reduction. The loop in the following Fortran example is not parallelized because of an apparent dependence between the references to `A(I)` on line 4 and the assignment to `A(JA(J))` on line 5. The compiler does not realize that the values of the elements of `JA` never coincide with the values of `I`, and so, assuming that they might, conservatively avoids parallelizing the loop.

```
DATA JA /11,12,13,14,15,16,17,18,19,20/
DO I = 1, 10
  DO J = I, 10
      A(I) = A(I) + B(J) * C(J)     !LINE 4
      A(JA(J)) = B(J) + C(J)        !LINE 5
  ENDDO
ENDDO
```

Note

In this example as well as the examples that follow, the apparent dependence becomes real if any of the values of the elements of `JA` are equal the values iterated over by `I`.

A `no_loop_dependence` directive or pragma placed before the `J` loop tells the compiler that the indirect subscript does not cause a true dependence. Because reductions are a form of dependence, this directive also tells the compiler to ignore the reduction on `A(I)`, which it would normally handle. Ignoring this reduction causes the compiler to generate incorrect code for the assignment on line 4; the apparent dependence on line 5 is properly handled

because of the directive. The resulting code runs fast but produces incorrect answers.

In the following analogous C example, the apparent dependence is between the reference to `a[i]` on line 4 and `a[ja[j]]` on line 5:

```
ja[] = {11,12,13,14,15,16,17,18,19,20};
for (i=0; i<10; i++)
  for (j=0; j<10; j++) {
    a[i] += b[j] * c[j];     /* line 4 */
    a[ja[j]] = b[j] + c[j]; /* line 5 */
  }
```

To solve this problem, distribute the `J` loop, isolating the reduction from the other statements, as shown in the following Fortran example:

```
      DATA JA/11,12,13,14,15,16,17,18,19,20/
      DO I = 1, 10
        DO J = I, 10
          A(I) = A(I) + B(J) * C(J)
        ENDDO
      ENDDO
C$DIR NO_LOOP_DEPENDENCE(A)
      DO I = 1, 10
        DO J = I, 10
          A(JA(J)) = B(J) + C(J)
        ENDDO
      ENDDO
```

And in C:

```
for (i=0; i<10; i++)
  for (j=i; j<10; j++)
    a[i] += b[j] * c[j];
#pragma _CNX no_loop_dependence(a)
for (i=0; i<10; i++)
  for (j=i; j<10; j++)
    a[ja[j]] = b[j] + c[j];
```

The apparent dependence is removed, and both loops can be optimized.

This problem occurs only if the reduction and the apparent dependence involve the same variable or array element. If the reduction and the apparent dependence involve different variables or array elements, as in the following Fortran example, both reduction and dependence are handled correctly without your intervention:

```
      DATA JD /6, 7, 8, 9, 10/
      DO I = 1, 5
C$DIR   NO_LOOP_DEPENDENCE(D)
        DO J = I, 5
          A(I) = A(I) + B(J) * C(J)
          D(JD(J)) = D(I) + B(J) + C(J)
        ENDDO
      ENDDO
```

In C:

```
jd[] = {6,7,8,9,10};
for (i=0; i<5; i++)
#pragma _CNX no_loop_dependence(d)
  for (j=0; j<5; j++){
    a[i] += b[j] * c[j];
    d[jd[j]] = d[i] + b[j] + c[j];
  }
```

Evaluation order

Assumptions the compiler makes about reordering code can sometimes cause answers to change at higher optimization levels. If this happens, use parentheses to force a specific order of evaluation.

Incrementing by zero

If the compiler parallelizes a loop that increments a variable by zero on each trip, the loop can produce incorrect answers or cause the program to abort. This error can occur when a variable used as an incrementation value is accidentally set to zero. If the compiler detects that the variable has been set to zero, the compiler does not parallelize the loop. If the compiler cannot detect the assignment, however, the symptoms described below occur. The following Fortran example shows three loops that increment by zero:

```
CALL SUB1(0)
.
.
.
SUBROUTINE SUB1(IZR)
DIMENSION A(100), B(100), C(100)
J = 1
DO I = 1, N
   B(I) = A(J)
   A(J) = C(I)
   J = J + IZR
ENDDO
DO I = 1, N, IZR      ! INCREMENT VALUE OF 0 IS
                      ! NON-STANDARD
   A(I) = B(I)
ENDDO
DO I = 1, N
   J = J + IZR
   B(I) = A(J)
   A(J) = C(I)
ENDDO
```

Because `IZR` is an argument passed to `SUB1`, the compiler does not detect that `IZR` has been set to zero. All three loops parallelize at `-O3`, but because of the zero increments, their runtime behavior cannot be reliably predicted. All three loops compile at `-O1`, but the second loop, which specifies the step as part of the `DO` statement (or as part of the `for` statement in C), will cause a runtime error. Runtime behavior of the other two loops cannot be predicted at `-O1`.

The analogous C code follows:

```
float a[100],b[100],c[100];

void sub1(int izr)
{
  int i,j = 1;

  for(i=0; i<100; i++){
    b[i] = a[j];
    a[j] = c[i];
    j += izr;
  }
  for(i=0; i<n; i+=izr)
    a[i] = b[i];
  for(i=0; i<n;i++) {
    j = j + izr;
    b[i] = a[j];
    a[j] = c[i];
  }
}

main()
{
  sub1(0);
}
```

Nondeterminism of parallel execution

In a parallel program, threads do not execute in a predictable or determined order. If you force the compiler to parallelize a loop when a dependence exists, the results are unpredictable and can vary from one execution to the next.

Consider the following Fortran example:

```
DO I = 1, N-1
  A(I) = A(I+1) * B(I)
  .
  .
  .
ENDDO
```

The compiler will not parallelize this code as written because of the dependence on `A(I)`. This dependence requires that the original value of `A(I+1)` is available for the computation of `A(I)`. If this code was parallelized, some values of `A` would be assigned by some processors before they were used by others, resulting in incorrect assignments. Because the results depend on the order in which statements execute, the errors are nondeterministic. The loop must therefore execute in iteration order to ensure that all values of `A` are computed correctly.

The analogous C code follows:

```
for(i=0;i<n-1;i++) {
  a[i] = a[i+1] * b[i];
  .
  .
  .
}
```

Loops containing dependences can sometimes be manually parallelized using the `LOOP_PARALLEL(ORDERED)` directive as described in Chapter 6, "Advanced shared-memory programming." Otherwise, unless you are sure that no loop-carried dependence exists, it is safest to let the compiler choose which loops to parallelize.

Large trip counts at `-O1` and above

When a loop is optimized at level `-O1` or above, its trip count must occupy no more than a signed 32-bit storage location. The largest positive value that can fit in this space is 2^{31} - 1 (2,147,483,647). If the compiler can determine that the trip count is larger than this at compile time, it will issue a warning. Loops with trip counts that cannot be determined at compile time but that exceed 2^{31} - 1 at runtime will yield wrong answers.

This limitation only applies at optimization levels `-O1` and above.

Loops with trip counts that overflow 32 bits can be optimized by manually strip mining the loop.

Hidden ordered sections

While it is legal and sometimes useful to place ordered sections in separate routines from their parent ordered loops, this practice can cause runtime deadlock in some situations.

Consider the following Fortran example:

```
      PROGRAM SEPMAIN
      REAL A(100)
      .
      .
      .
C$DIR BEGIN_TASKS
      CALL SUB1(A)
      .
      .
      .
C$DIR NEXT_TASK
      CALL SUBN
      .
      .
      .
C$DIR END_TASKS
      .
      .
      .
      END

      SUBROUTINE SUB1(A)
      REAL A(100)
C$DIR GATE(LOCK)
      LK = ALLOC_GATE(LOCK)
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 2, 100
        CALL SUB2(LOCK,A,I)
      ENDDO
      LK = FREE_GATE(LOCK)
      END

      SUBROUTINE SUB2(LOCK,A,I)
C$DIR GATE(LOCK)
      REAL A(100)
      INTEGER I
C$DIR ORDERED_SECTION(LOCK)
      A(I) = A(I-1)
C$DIR END_ORDERED_SECTION
      END
```

Here, the tasks in the main program go thread parallel by default, so when the loop in SUB1 is reached it cannot go parallel. However, because parallelism exists in the main program, the ordered section in SUB2 expects that it will be executed by all parallel threads. Only thread 0 is executing SUB1, since only the first ordered task calls it; thread 0 therefore runs the DO loop in

Limits of optimization

SUB1 and passes through the ordered section in SUB2. After this, the ordered section will wait for thread 1 to enter before allowing thread 0 back in on the next iteration of the I loop. Thread 1 never calls SUB1, so it never has the opportunity to enter the ordered section. This causes the program to deadlock in the ordered section.

The analogous C code follows:

```
void sub1(float *a) {
  static gate_t lock;
  int lk;
  lk = alloc_gate(&lock);
#pragma _CNX loop_parallel(ordered, ivar=1)
  for(i=1;i<100;i++)
    sub2(lock,a,i);
  lk = free_gate(&lock);
}

void sub2(gate_t lock, float *a, int i) {
#pragma ordered_section(lock)
  a[i] = a[i-1];
#pragma end_ordered_section
}

main() {
  float a[100];
  .
  .
  .
#pragma _CNX begin_tasks
  sub1(a[]);
  .
  .
  .
#pragma _CNX next_task
  subn();
  .
  .
  .
#pragma _CNX end_tasks
  .
  .
  .
}
```

If you encounter this kind of problem, try moving the ordered section into the same routine as its parent loop, or, if possible, add another dimension of parallelism so that the loop running the ordered section is running in parallel.

Potentially unsafe optimizations

9

This chapter describes optimizations that can potentially generate incorrect results. By default, the SPP Series Fortran and C compilers avoid performing these optimizations. You can enable potentially unsafe optimizations by specifying the `-uo` compiler option.

The `-uo` option enables these optimizations:

- Simple strength reductions
- Code motion
- Elimination of type conversions

Simple strength reduction

Chapter 3, "Compiler optimizations," describes how the compiler replaces slow operations with faster ones on the assumption that arithmetically equivalent expressions always yield the same results. However, reducing an expression such as `X/C` to `(1/C)*X` can be unsafe because it can increase roundoff error.

When you use the `-uo` option, the compiler replaces division operations with multiplication. If a possibility of overflow exists, however, the compiler does not perform this optimization.

Code motion

The compiler normally moves an invariant expression out of a loop only if the expression is located on a path to all loop exits. When you use -uo, the compiler can move an invariant expression out of a loop if the expression does not lie on a path to all loop exits.

In the following example, the invariant expression A=B/X is relocated only when the program is compiled with the -uo option:

```
DO I = 1, 100
  IF (X .NE. 0) THEN
    A= B/X
    AR(I) = A*C
  ELSE
    AR(I) = D*C
  ENDIF
ENDDO
```

Conversion elimination

Type conversions are costly in terms of machine cycles, and they can inhibit optimization. When you use the -uo option, the compiler eliminates costly type conversions by creating REAL induction variables that it then increments concurrently with the loop's INTEGER induction variables. Consider the following Fortran loop:

```
REAL A(100000)
.
.
.
DO I = 1, 100000
  A(I) = I
ENDDO
```

And its C equivalent:

```
float a[100000];

int foo()
{
  int I;

  for(I=0; I<100000; I++)
    a[I] = I;
}
```

Here, in absence of the -uo option, I must be converted to floating point on every iteration of the loop. With the -uo option the compiler avoids this costly operation by copying I into a REAL (or float) induction variable before entering the loop, then incrementing this variable by 1.0 on every iteration of the loop. At optimization level -O3, the compiler can then parallelize the following optimized Fortran loop:

```
     REAL_I = 1.0
     I = 1
10   A(I) = REAL_I
     REAL_I = REAL_I + 1.0
     I = I + 1
     IF (I .LE. 100000) GOTO 10
```

In C:

```
float a[100000];

int foo()
{
  int I;
  float REAL_I;

  for(I=0,REAL_I=0.0;I<100000;I++,REAL_I++)
    a[I] = REAL_I;
}
```

This optimization is considered potentially unsafe because the internal representation of real numbers is inexact, and this can lead to a significant accumulated error when REAL_I is incremented over the course of the loop.

Compiler directives and pragmas

A

This appendix presents an alphabetical list of all the Fortran directives and C pragmas that are supported by the Convex SPP Series compilers.

Overview

This appendix is intended to provide only a brief overview of the available directives and pragmas. More specific information and examples can be found elsewhere in this guide. The Fortran directives not supported as C pragmas are expressed as storage class extensions (`thread_private`, etc.) or as typedefs (`gate_t`, `barrier_t`, etc.), and are described in Chapter 5, "Memory classes," and Chapter 6, "Advanced shared-memory programming."

The form of an SPP Series Fortran compiler directive is:

`C$DIR [CSERIES|SPP]` *directive-specification*

The form of an SPP Series C pragma is:

`#pragma _CNX [CSERIES|SPP]` *directive-specification*

Where *directive-specification* is one of the directives/pragmas described in this chapter. `CSERIES` or `SPP` can optionally be included to indicate the target machine for the directive; this is useful if you are writing code that may be compiled on either C Series or Exemplar architectures, and wish to include directives or pragmas for both without having the compiler issue warnings for those that do not apply to the current machine.

Directive names are presented here in lower case; they may be specified in either case in both languages, but "`#pragma`" must always appear in lower case in C. In the sections that follow, *namelist* represents a comma-delimited list of names. These names can be variables, arrays, or `COMMON` blocks. In the case of a `COMMON` block, its name must be enclosed within slashes. The occurrence of a lower-case *n* is used to indicate a compile-time integer constant expression; values not determinable at compile time cannot be used. Occurrences of *gate_var* are for variables that have been, or are being, defined as gates. Any parameters that appear within square brackets (`[ ]`) are optional.

Directives and pragmas

Brief descriptions of the available directives and pragmas follow in this section. Where appropriate, cross references to chapters containing more detailed information are included.

`align_cti(`*namelist*`)`

Aligns the variables and arrays listed in *namelist* on CTIcache boundaries. This allows for more efficient data reuse.

`barrier(`*namelist*`)`

This Fortran directive is used to denote a list of variables, as given in *namelist*, that will be used as the synchronization variables for the barrier routines. This does not imply any synchronization in itself, it is simply defining the barrier variables. Note that in C, `barrier` is a typedef, rather than a pragma. For more information, refer to Chapter 6, "Advanced shared-memory programming."

begin_tasks [(*attribute_list*)]

This directive or pragma defines the beginning of a section (or sections; see next_task) of code that will be executed as an independent, parallel task. Each task is executed by a separate thread. begin_tasks must have an accompanying end_tasks in the same program unit.

The optional *attribute_list* can be any of the following legal combinations:

- (max_threads=*m*)
- (ordered)
- (nodes)
- (threads)
- (ordered,nodes)
- (ordered,threads)
- (ordered,max_threads=*m*)
- (nodes,max_threads=*m*)
- (threads,max_threads=*m*)
- (ordered,nodes,max_threads=*m*)
- (ordered,threads,max_threads=*m*)

Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for a complete discussion of parallel tasking.

`block_loop[(block_factor=`*n*`)]`

This directive or pragma is used to indicate a specific loop to block, and optionally, the block factor that will be used in the compiler's internal computation of loop nest based data reuse. In absence of the `block_factor` argument, this directive is useful for indicating which loop in a nest the compiler should block. Refer to Chapter 3, "Compiler optimizations," for more information on blocking.

`block_shared(`*allocatable_array_namelist*`)`

This Fortran directive is used to declare arrays as being of type `block-shared`. Block-shared arrays are sized to be an integral multiple of the page size. The pages of the array are distributed in same-size blocks across the hypernodes on which the process is executing in the subcomplex. If the user-specified size is not an integral multiple of page size × `num_nodes()` then the compiler will automatically round it up to meet this criterion. Refer to Chapter 5, "Memory classes," for more information.

`critical_section[(`*gate_var*`)]`

This defines the beginning of a code block in which only one thread may be executing at a time. The end of the code block must be indicated by an `end_critical_section` directive or pragma, which must appear in the same flow of control within the same program unit. The optional *gate_var* can be used to differentiate between parallel tasks. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

dynsel [(*trip_count=n*)]

This enables workload-based dynamic selection for the immediately following loop. *trip_count* represents either the thread_trip_count or node_trip_count attribute. If thread_trip_count = *n* is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the thread-parallel version is run. If node_trip_count = *n* is specified, the serial version of the loop is run if the iteration count is less than *n*; otherwise, the node-parallel version is run. *n* must be a compile-time constant.

end_critical_section

This directive or pragma is used to define the end of the critical section that was begun with the critical_section directive or pragma. critical_section and end_critical_section must appear as a pair. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

end_ordered_section

This directive or pragma is used to define the end of the ordered section that was begun with the ordered_section directive or pragma. ordered_section and end_ordered_section must appear as a pair. Refer to Chapter 6, "Advanced shared-memory programming," for more information on ordered sections.

end_parallel

Signifies the end of a parallel region. The parallel directive signifies the beginning of a parallel region. Refer to Chapter 4, "Basic shared-memory programming," for more information.

end_tasks

This is used to terminate the specification of parallel tasks indicated by begin_tasks and next_task. It must appear at the end of the last section of parallel code defined by these directives or pragmas. All of these must appear in the same program unit. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

far_shared(*namelist*)

This Fortran directive causes the compiler to place the data objects in *namelist* (i.e., variables, arrays, or COMMON blocks) into far_shared memory. far_shared memory is the most general form that is distributed on a page basis across the memories of all hypernodes in a subcomplex. The far_shared data objects of a process are addressable by all threads of that process. Refer to Chapter 5, "Memory classes," for more information on memory classes.

far_shared_pointer(*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in far_shared memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information on memory classes.

gate(*namelist*)

This Fortran directive defines a gate variable that will be subsequently used in a critical section, ordered section, or passed as an argument to the synchronization intrinsics. Note that in C, gate is a typedef, rather than a pragma. Refer to Chapter 6, "Advanced shared-memory programming," for more information.

`loop_parallel[(`*attribute_list*`)]`

This is an explicit instruction to the compiler to parallelize the immediately following loop. The loop iterations will be run in an indeterminate order unless the optional `ordered` attribute appears. The user is responsible for any required data privatization and loop synchronization, as described in Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming." The optional *attribute_list* can be any of the following combinations:

- `(chunk_size=`*n*`)`
- `(max_threads=`*m*`)`
- `(ordered)`
- `(nodes)`
- `(threads)`
- `(dist)`
- `(ordered,nodes)`
- `(ordered,threads)`
- `(ordered,dist)`
- `(nodes,chunk_size=`*n*`)`
- `(threads,chunk_size=`*n*`)`
- `(dist,chunk_size=`*n*`)`
- `(chunk_size=`*n*`,max_threads=`*m*`)`
- `(ordered,max_threads=`*m*`)`
- `(nodes,max_threads=`*m*`)`
- `(threads,max_threads=`*m*`)`
- `(dist,max_threads=`*m*`)`
- `(ordered,nodes,max_threads=`*m*`)`
- `(ordered,threads,max_threads=`*m*`)`
- `(ordered,dist,max_threads=`*m*`)`
- `(nodes,chunk_size=`*n*`,max_threads=`*m*`)`
- `(threads,chunk_size=`*n*`,max_threads=`*m*`)`
- `(dist,chunk_size=`*n*`,max_threads=`*m*`)`
- `(ivar = `*indvar*`)`

`ivar = `*indvar* is:

- Required for all loops in C and for `DO WHILE` and hand-rolled loops in Fortran
- Optional for Fortran `DO` loops
- Can be specified with any other attribute

Attributes may be listed in any order. Any attribute combinations other than those listed above will be flagged with a fatal error by the compilers.

Refer to Chapter 6, "Advanced shared-memory programming," for more information.

`loop_private(`*namelist*`)`

This is used to declare a list of variables and/or arrays private to the immediately following loop. To be loop private, the variables and/or arrays must be assigned before they are used on any iteration of the immediately following loop. These private data items are distinct from the shared items of the same name that exist outside the loop. No values may be carried into the loop by `loop_private` variables. Values assigned to `loop_private` variables on the final iteration may be saved into the shared variables of the same name if the `save_last` directive or pragma also appears on this loop. If `save_last` is not used, then the value of any shared variable declared to be `loop_private` is undefined at loop termination. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

`near_shared(`*namelist*`)`

When applied to static variables at compile-time, this Fortran directive will cause all pages of the data objects in *namelist* to be mapped to physical pages on hypernode 0. If applied to allocatable arrays, then the pages of such arrays will be mapped to physical pages on the hypernode of the allocating thread. `near_shared` data can be addressed by any thread of a process on any hypernode in the subcomplex but it is "closer" (in terms of access latency) to the threads on the hypernode that allocates the data. Refer to Chapter 5, "Memory classes," for more information on memory classes.

near_shared_pointer(*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in near_shared memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information on memory classes.

next_task

This starts a block of code following a begin_tasks block that will be executed as a parallel task. The end of the code block is marked by another next_task or by an end_tasks directive or pragma.

This directive must appear within a begin_tasks and end_tasks pair. There is no limit on the number of next_task directives that can appear. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

no_block_loop

This informs the compiler to perform no blocking on the immediately following loop. Generally the compiler will attempt to automatically block a nested loop in order to achieve optimal cache data reuse between loop iterations. Refer to Chapter 3, "Compiler optimizations," for more information on loop blocking.

no_distribute

This disables loop distribution for the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information on loop distribution.

no_dynsel

This disables workload-based dynamic selection for the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information on dynamic selection.

no_fuse

This disables loop fusion for the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information on loop fusion.

no_loop_dependence(*namelist*)

Informs the compiler that the arrays in *namelist* do not have any dependences for iterations of the immediately following loop. Use `no_loop_dependence` for arrays only; use `loop_private` to indicate dependence-free scalar variables.

This will cause the compiler to ignore any dependences that it perceives to exist. This can enhance the compiler's ability to optimize the loop, including the possibility of parallelization.

Refer to Chapter 3, "Compiler optimizations," and Chapter 7, "Limits of optimization," for more information.

no_parallel

This prevents the compiler from generating parallel code for the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information.

no_peel

This prevents the compiler from peeling the immediately following loop. Loop peeling involves removing tests that appear for the first and/or last iterations of a loop to outside the loop body. Refer to Chapter 3, "Compiler optimizations," for more information.

`no_promote_test`

This prevents the complier from performing test promotion on the immediately following loop. Refer to Chapter 3, "Compiler optimizations," for more information.

`no_side_effects` (*funclist*)

This informs the compiler that the functions appearing in *funclist* have no side effects wherever they appear lexically following the directive. Side effects include modifying a function argument, modifying a Fortran COMMON variable, performing I/O, or calling another routine that does any of the above. The compiler can sometimes eliminate calls to procedures that have no side effects.

`no_unroll`

This disables loop unrolling. Refer to the section "Loop unrolling" on page 80 for more information.

`no_unroll_and_jam`

This disables unrolling and jamming of loops. Refer to the section "Loop unroll and jam" on page 83 for more information.

`node_private` (*namelist*)

This Fortran directive causes the variables and arrays specified in *namelist* to be replicated in the physical memory of each hypernode on which the process is executing. Thus, while each data object has a single image in virtual memory, it maps to a different physical location on each hypernode. Process threads within a hypernode all share access to the copy on their hypernode and cannot access the copies on other hypernodes. Refer to Chapter 5, "Memory classes," for more information.

node_private_pointer(*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `node_private` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information.

opt_level (*level*)

This C pragma forces the optimization level to be the lower of the level supplied on the command line and the level supplied in *level* for all routines following the pragma, which must appear at file scope. Consider the following example:

```
#pragma _CNX opt_level(-O3)
func() {
.
.
.
}
#pragma _CNX opt_level(-O1)
```

If the file containing this code was compiled using the `-O2` command-line argument, the function `func` would be compiled at optimization level `-O2` and functions following the second `opt_level` pragma would be compiled at optimization level `-O1`. `opt_level` has no effect if it appears inside a function. No Fortran directive version of `opt_level` is available.

ordered_section(*gate_var*)

This defines the beginning of an ordered section. An ordered section is the same as a critical section (a code block in which only one thread may be executing at a time) with the additional restriction that the threads must pass through the ordered section in iteration order. The end of the code block must be indicated by an `end_ordered_section` directive or pragma. Ordered sections must appear within the control flow of a `loop_parallel(ordered)`. Refer to Chapter 6, "Advanced shared-memory programming," for more information.

`parallel[(attribute_list)]`

This signifies the beginning of a parallel region of code. All code up to the following `end_parallel` directive or pragma will be run on all available threads. No loop transformations, data privatization, or parallelization analysis will be performed by the compiler on the code in the region.

The optional *attribute_list* can be any of the following legal combinations:

- `(max_threads=m)`
- `(nodes)`
- `(threads)`
- `(nodes,max_threads=m)`
- `(threads,max_threads=m)`

Refer to Chapter 4, "Basic shared-memory programming," for more information.

`parallel_private(namelist)`

This declares a list of variables or arrays private to the immediately following parallel region. It serves the same purpose for parallel regions that `task_private` serves for tasks. The privatized variables and arrays will not carry their values beyond the `end_parallel` directive or pragma. Refer to Chapter 4, "Basic shared-memory programming," for more information.

`peel_all`

This has the same effect as the `peel` directive or pragma except that it permits the compiler to replicate code without bound. Normally peeling will only replicate a certain amount of code. This can slow the compile phase and might even cause internal compiler tables to overflow. Refer to Chapter 3, "Compiler optimizations," for more information.

`peel`

This allows the compiler to peel the loop immediately following the directive, expanding the code beyond the default conservative limit, but not without bound. Refer to Chapter 3 for more information on peeling. Refer to Chapter 3, "Compiler optimizations," for more information.

prefer_fuse

This specifies fusion for particular loops. A prefer_fuse directive or pragma must immediately precede the loop for which you want to enable fusion. If fusion is enabled for the program, prefer_fuse has no effect. If fusion is disabled for the program, prefer_fuse must be specified on two or more neighboring loops, and will override the -nfl option. Refer to Chapter 3, "Compiler optimizations," for more information.

prefer_parallel [(*attribute_list*)]

This instructs the compiler to parallelize the following loop but only if it is safe to do so. A loop is safe to parallelize if it has an iteration count determinable at runtime before loop invocation, and contains no LCDs, procedure calls, or I/O operations. Refer to Chapter 4, "Basic shared-memory programming," for more information.

- (chunk_size=*n*)
- (max_threads=*m*)
- (ordered)
- (nodes)
- (threads)
- (ordered,nodes)
- (ordered,threads)
- (nodes,chunk_size=*n*)
- (threads,chunk_size=*n*)
- (chunk_size=*n*,max_threads=*m*)
- (ordered,max_threads=*m*)
- (nodes,max_threads=*m*)
- (threads,max_threads=*m*)
- (ordered,nodes,max_threads=*m*)
- (ordered,threads,max_threads=*m*)
- (nodes,chunk_size=*n*,max_threads=*m*)
- (threads,chunk_size=*n*,max_threads=*m*)

Attributes may be listed in any order. Any attribute combinations other than those listed above will be flagged with a fatal error by the compilers.

promote_test_all

This instructs the compiler to promote tests out of the immediately following loop. The amount of code replication is not constrained in any way. Refer to Chapter 3, "Compiler optimizations," for more information.

promote_test

This instructs the compiler to promote tests out of the immediately following loop, expanding code beyond the default conservative limit but not without bound. Refer to Chapter 3, "Compiler optimizations," for more information.

row_wise(*array_namelist*)

In Fortran, the normal convention of storing arrays in column major order (left-most index varies fastest) can be switched for the arrays specified in *array_namelist*. Specifying `row_wise` for an array causes it to be stored in row major order which is consistent with the C language storage ordering.

save_last

This specifies that all variables named in an associated `loop_private(`*namelist*`)` or `task_private(`*namelist*`)` must have their last value saved into the "shared" variable of the same name, at loop or task termination. If `save_last` is not specified then the values in any privatized variables or arrays are indeterminate at loop termination. Refer to Chapter 6, "Advanced shared-memory programming," for more information.

scalar

This directive or pragma prevents the compiler from performing reordering transformations on the following loop; the compiler will not parallelize or data-localize a loop on which this directive appears.

`sync_routine`(*routinelist*)

This indicates to the compiler that the routines listed in *routinelist* are user-defined synchronization routines, so that the compiler will not attempt to move code across these routine calls. Use `sync_routine` anytime you hide a call to a compiler synchronization function inside another routine call, or anytime you use CPSlib functions for synchronization.

`sync_routine` is only effective for the listed routines that lexically follow it in the routine in which it appears.

`task_private`(*namelist*)

This will privatize the variables and arrays specified in *namelist* for each task specified in the immediately following `begin_tasks/end_tasks` block. The privatized variables and arrays will not carry their values beyond the `end_tasks` directive or pragma. Refer to Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," for more information.

`thread_private`(*namelist*)

This Fortran directive will cause the variables and arrays specified in *namelist* to be treated (by software convention) as being `thread_private`. `thread_private` data objects map to unique `node_private` addresses for each thread of a process. Refer to Chapter 5, "Memory classes," for more information.

`thread_private_pointer`(*alloc_var_name*)

This Fortran directive causes the compiler to place the (compiler-generated, hidden) pointer to the allocated object in `thread_private` memory, regardless of the memory class to which the object itself is allocated.

This directive applies only to Fortran 90 allocatable data objects. Refer to Chapter 5, "Memory classes," for more information.

`unroll[(unroll_factor=n)]`

This causes the immediately following loop to have its body replicated (*n* times if `unroll_factor` is specified) and unrolled in order to reduce loop overhead.

This only takes place if

- The loop is a scalar loop
- There is no internal branching
- The specified loop is innermost

Complete unrolling occurs if the loop count is less than 5. Otherwise only partial unrolling occurs. When used on a loop nest, this directive or pragma must be placed on the loop that ends up being innermost after compilation, or it will have no effect.

Refer to the section "Loop unrolling" on page 80 for more information.

`unroll_and_jam[(unroll_factor=n)]`

This causes one or more noninnermost loops in the immediately following nest to be partially unrolled (to a depth of *n* if `unroll_factor` is specified), then fuses the resulting loops back together. It must be placed on a loop that ends up being noninnermost after any compiler-initiated interchanges.

Refer to the section "Loop unroll and jam" on page 83 for more information.

Optimization options

B

This appendix lists the optimization options available for use with Convex SPP Series Fortran and C compilers and briefly describes each option. A complete list of all Fortran compiler options is available in Chapter 1, "Compiling programs," of the *Fortran User's Guide*. A complete list of all C compiler options is available in Chapter 2, "Compiler fundamentals," of the *C User's Guide*.

Optimization level and related options

The options listed in this section specify the level of optimization allowed.

`-no`

Machine instruction level scalar optimization. This option is the default.

`-O0`

Basic block level scalar optimization.

`-O1`

`-O0` optimizations plus program unit level scalar optimization and global register allocation.

`-O2`

`-O1` optimizations plus data localization.

`-O3`

`-O2` optimizations plus parallelization.

`-or` *table*

Specifies the contents of the optimization report. Values of *table* and the optimization reports they produce are shown in Table 11.

Table 11 Optimization report contents

Table value	Report contents
`all`	Loop table, privatization table and array table
`loop`	Loop table only (default)
`array`	Array table only
`private`	Loop table and privatization table
`none`	No report

For more information about the optimization report, refer to Appendix C, "Optimization report."

`-noautopar`

Disables parallelization for any loop not specifically parallelized by a `loop_parallel` or `prefer_parallel` directive or pragma. Loops preceded by these directives are parallelized; all other loops are not parallelized. This is effective only when specified with the `-O3` option.

`-nonodepar`

Disables hypernode-level parallelization. This prevents the compiler from implementing node-parallelism, but allows the implementation of both automatic and directive-specified thread-parallelism.

Cross compilation options

The options listed in this section allow a program to be optimized for a machine configuration that differs from the configuration of the machine the program is being compiled on.

`-tm` *target*

Specifies the target machine architecture for which compilation is to be performed. *target* takes the value `spp1000` to specify SPP1000 Series machines, `spp1200` for SPP1200 Series machines, or `spp1600` for SPP1600 Series machines. The default *target* value corresponds to the machine on you which you invoke the compiler.

`-cache` *n*

Instructs the compiler to assume a per-processor direct-mapped cache size of *n* kbytes. The compiler uses this information when determining a loop's blocking factor. Cache size defaults to the size of the cache on which the program is being compiled. This option may be useful when cross-compiling for future SPP Series machines which may have different cache sizes.

Loop replication options

The options listed in this section control loop replication optimizations.

`-mrl`

Raises the limit on replicated loops. This may significantly increase compile time. If you are not concerned with long compile times, you can use this option when the compiler issues advisories that indicate that loop replication is being limited for certain optimizations. Optimizations that replicate loops include `IF-DO` and `if-for` optimizations, dynamic selection, loop unrolling, unroll and jam, and loop blocking. This option is available only at optimization level `-O2` or `-O3`.

`-ur`

Causes the compiler to automatically find unrollable loops and unroll them. Loops with iteration counts determinable at compile time to be less than 5 are unrolled completely. Those with indeterminate iteration counts, or determinate counts of 5 or more, are partially unrolled. Only innermost loops can be unrolled. This option is the default, and is available only at optimization levels `-O2` or `-O3`.

`-urn` *n*

Enables loop unrolling with an unroll factor of *n*; *n* is the number of times to replicate the body of the loop.

`-nur`

Disables loop unrolling.

`-uj`

Enables the unroll and jam optimization. This optimization unrolls one or more noninnermost loops in a nest, then jams the resulting loops back together, greatly improving register usage. This option is the default, and is available only at optimization levels `-O2` and `-O3`. For more information, refer to the section "Loop unroll and jam" on page 83.

`-ujn` *n*

Enables unroll and jam with an unroll factor of *n*.

`-nuj`

Disables unroll and jam.

`-ds`

Enables workload-based dynamic selection. This optimization causes the compiler to generate parallel and serial versions of parallelizable loops whose iteration counts are unknown at compile time. At runtime, the loop's workload is compared to parallelization overhead, and the parallel version is run only if it is profitable to do so.

`-nds`

Disables dynamic selection.

Loop blocking options

The options listed in this section control the loop blocking optimization.

`-blockloop` *n*

Instructs the compiler to use a block factor of *n* and to automatically select which loops to block. In a loop nest containing *m* loops, *m*-1 loops will be blocked. If the `-blockloop` option is not used, the compiler will attempt to choose an optimal blocking factor based on the cache size and the amount of data being manipulated by the loop. Loop blocking is provided at optimization levels `-O2` and `-O3` unless the `-noblock` option is specified.

`-noblock`

Disables loop blocking for the sources being compiled. Because loop blocking occurs at optimization levels `-O2` and `-O3`, the `-noblock` option is effective only at these levels.

IF-DO and if-for optimization options

The options listed in this section control the degree of loop peeling and test promotion allowed.

`-nopeel`

Disallows loop boundary value peeling, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to the `-peel` and `-peelall` options described in this section.

`-noptst`

Disallows test promotion, which is enabled by default at optimization levels `-O2` and `-O3`. Refer to `-ptst` and `-ptstall` below.

`-peel`

Removes the first and/or last iterations of a loop when doing so removes conditional tests from the loop. This is done when the loop contains a test involving an explicit reference to the loop index variable that always evaluates to `.TRUE.` or `.FALSE.` for the first and/or last iteration. By default, the compiler peels boundary values and expands code up to a predetermined conservative limit. With the `-peel` option, this limit is increased and code expansion may become significant. `-peel` must be used with the `-O2` or `-O3` optimization options.

`-peelall`

Same as `-peel`, but allows code expansion without bound. For code containing large numbers of boundary value operations, this can greatly lengthen compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-peelall` must be used with the `-O2` or `-O3` optimization options.

`-ptst`

Causes a test to be promoted out of the loop that encloses it by replicating the containing loop for each branch of the test. By default, the compiler replicates code up to a predetermined conservative limit. The `-ptst` option increases this limit and can cause a noticeable increase in compile time. `-ptst` must be used with the `-O2` or `-O3` optimization options.

`-ptstall`

Same as `-ptst`, but allows code replication without bound. For loops containing large numbers of tests, this can significantly increase compile time and can increase the size of the code enough to exceed the limits of some of the compiler's internal tables. `-ptstall` must be used with the `-O2` or `-O3` optimization options.

Register use options

The options listed in this section control the extent to which registers are exploited.

`-gr`

Enables global register allocation. This is the default at optimization level `-O1` and above. Refer to the section "Global register allocation" on page 59.

`-ngr`

Disable global register allocation.

`-nga`

Disables global register allocation for arguments passed by reference. This is useful for programs that violate the ANSI standard by passing a constant as a procedure argument when the procedure may potentially write to the dummy argument. Refer to the section "Global register allocation" on page 59.

`-ngs`

Disables global register allocation for shared-memory variables that are visible to multiple threads. This option may help if a variable shared among parallel threads is causing wrong answers. Refer to the section "Global register allocation" on page 59 for more information.

`-br`

Allows the generation of load and store instructions that modify the base register directly, rather than requiring a separate instruction. This eliminates extra instructions in loops that perform address updates, improving performance. `-br` is on by default.

`-nbr`

Disable base register modification load/store instructions.

Data alignment options

The options listed below allow you to control data alignment. Be aware that aligning arrays generally enhances performance, but can occasionally degrade performance.

`-align cache` (Available only in Fortran)

Aligns arrays on processor cache line boundaries (32 bytes). See the following section, "Using `-align cache` and `-align cache_check`," for more information.

`-align cache_check` (Available only in Fortran)

Performs diagnostics of arrays; issues warnings when arrays are encountered that are not on processor cache line boundaries (32 bytes). See the following section, "Using `-align cache` and `-align cache_check`," for more information.

`-align cseries`

Causes the Fortran compiler to store `COMMON` blocks using "tight" packing (the ANSI standard). Tight `COMMON` block packing is the default on Convex C Series machines.

Rather than padding `COMMON` block data items to their natural boundaries (the default on SPP Series machines), this method stores them in memory contiguously, so partial-word items may cause other data items to align on unnatural boundaries. For instance, a `REAL*8` item may align on a 2- or 4-byte boundary instead of an 8-byte boundary. Note that this natural boundary alignment is in addition to the CTIcache boundary alignment discussed in the section "Data alignment" on page 21.

If you specify tight packing, you must take other measures to prevent runtime errors as described in the *Release Notice* for the compiler you are using.

`-align cti`

Aligns all arrays on CTIcache boundaries, increasing the efficiency of data reuse in multihypernode systems.

`-align spp`

The opposite of `-align cseries` and is the default on SPP Series machines.

For more information, refer to the "`COMMON` block packing" section of Chapter 5, "Specification statements," of the *Fortran Language Reference*.

Using `-align cache` and `-align cache_check`

Cache misses and false sharing can quickly degrade performance of a parallel application. Aligning the arrays that are referenced in parallel regions on cache line boundaries is one way to reduce these situations.

Use the `fc` command-line option `-align cache` to align arrays. Use the `fc` command-line option `-align cache_check` to perform array diagnostics without actually aligning any arrays.

To obtain more performance gains, use proper strip mining that takes account of the cache line size, in addition to array alignment.

`-align cache` (Available only in Fortran)

Aligns arrays on processor cache line boundaries (32 bytes). This option implies the `-align spp` option. The `-align cti` option overrides the `-align cache` option.

Using `-align cache` causes:

- Local arrays to be aligned on a cache line boundary in stack space or in heap space.
- Each *true array member* in a COMMON block to be aligned. (True array members are array members that are directly allocated storage space; members that are assigned storage through EQUIVALENCE statements are not aligned.)

Consider the following example:

```
REAL A,B(100),C(50)
EQUIVALENCE(B(51),C)
COMMON /COMB/A,B
```

Without the `-align cache` option, the array B is on offset 4 and array C is on offset 204, counting from the start of the /COMB/ block, which is on a 64-byte boundary because it is a COMMON block. With `-align cache`, the compiler inserts 28 bytes of padding between the variable A and array B so that B starts on a 32 byte offset. The C array then starts at a 232 byte offset.

If an EQUIVALENCE array overlaps two arrays that are adjacent in a COMMON block, padding may be unsafe. Consider the example below. If -align cache were used in compiling the example, padding would be inserted between arrays A and B. Because of the equivalence between arrays A and C, Array C would then contain some of that padding and would not contain all of array B.

```
REAL A(10),B(10),C(20)
EQUIVALENCE(A,C)
COMMON /COMB/A,B
```

In such a case, the compiler still continues padding but issues the following warning message:

```
Warning: cache alignment of array A and B in
common block COMB may cause incorrect program
execution due to equivalence with array C;
aligning on 32-byte boundary anyway due to
-align cache flag
```

If you use the -align cache option when compiling any file in your application, you must use the option when compiling all other files in your application. Failure to consistently use the -align cache option breaks interfaces between program modules. For example, suppose the programs foo.f and bar.f both reference the same COMMON block, but only foo.f is compiled with -align cache. COMMON members in foo.f will be padded, but members in bar.f will not be padded; the interface between foo.f and bar.f will be broken. The interface is also broken if the COMMON block name is referenced in C programs.

-align cache_check (Available only in Fortran)

Issues diagnostics when arrays are encountered that are not on a processor cache line boundary (32 bytes). Does not perform actual padding. This option implies the -align spp option.

Combining `-align` options

You can combine the following `-align` options:

- `cache`
- `cache_check`
- `cseries`
- `cti`
- `spp`

according to the table below:

Table 12 Combinations of `-align` options

	cache	cache_check	cseries	cti	spp
cache	OK	NR	NR	NR	OK
cache_check	NR	OK	NR	OK	OK
cseries	NR	NR	OK	NR	NR
cti	NR	OK	NR	OK	OK
spp	OK	OK	NR	OK	OK

In the table above, OK means the combination is acceptable; NR means the combination is not recommended or is not supported.

C aliasing options

Potential aliases occur more frequently in C than Fortran because of C's typically heavy use of pointers. This section covers the options supported by the Convex C compiler to help you deal with aliasing problems.

`-alias array_args`

Causes the compiler to assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but can allow greater optimization to occur. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to the elements of the formal parameter (for example, `formalParameter[10] = x[10]`).

The following source code illustrates a situation in which the `-alias array_args` option is useful:

```
void vaf(char []);
char global = 'A';

int main()
{
  char b[10];

  vaf(b);
  return(1);
}

void vaf(char array[]){
  int i;

  for(i=0; i<10; i++)
    array[i] += global;
}
```

The loop in the `vaf` function can be parallelized when the `-alias array_args` option is specified because the compiler can assume that the address of `global` is not the address of an element of `array`.

`-alias standard`

Performs aliasing based on assumptions permitted in ANSI C; pointers of one object type can only reference objects of that same type. For example, a pointer to an object of type `float` cannot reference an object of type `int`. Such assumptions permit additional optimizations. If this option is specified when such conditions do not exist, the resulting program may not function correctly.

`-alias cautious`

Tries to compile with `-alias standard`, but if inappropriate constructs are found, compiles with `-alias worst` instead. This is the default in the ANSI C-compatible modes.

`-alias worst`

Performs pointer aliasing based on the assumption that pointers can modify objects of any type. This is the default in the backward-compatible (`-pcc`) mode.

The default aliasing algorithm for ANSI C assumes that pointers of one type do not point to objects of another type. For example, `int` pointers cannot point to objects with type `float`.

`-alias [no_global | global]`

When `-alias no_global` is specified, the compiler assumes that globals (external variables) are not accessed via pointers. The default is `global`.

Compiling the following code with `-alias no_global` would be an *incorrect* use of the option because the value of the global is accessed through the pointer `gptr` as well as the variable `glob`.

```
int glob; /* wrong answers result if
             compiled with -alias
             no_global!!!          */
int *gptr;bar()
{
  gptr = &glob;
}

main()
{
  int tmp;

  bar();
  glob = 1;
  *gptr = 2;
  tmp = glob;
  printf("%d\n", tmp);
}
```

The program may print 1 when compiled with `-alias no_global`.

`-alias no_addr`

Indicates that address operands do not introduce aliases. This is useful when your program passes data by reference and you are sure that no aliases are produced. The default is `-alias addr`.

`-alias ptr_args`

Causes the compiler to assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but allows greater optimization to occur. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to variables identified by the formal parameter (for example, `formalParameter[10] + x[10]`).

`-alias restrict_args`

Causes the compiler to treat the code as though a `restrict` qualifier was applied to each pointer parameter. The `restrict` qualifier tells the compiler that a pointer provides exclusive access to the data object at a given memory location. When you use the `restrict` qualifier, you must access the object pointed to by a `restrict` pointer only through that pointer. If you reference the object by some other means, the compiler may incorrectly optimize code.

Other optimization options

This section lists optimization options that cannot be otherwise categorized.

`-il`

Instructs the Fortran compiler to prepare an intermediate language (`.fil`) file for a subprogram that is to be used for inline substitution. The `-il` option cannot be used with the `-c`, `-cs`, or `-S` options. Optimization levels are ignored. Refer to the *Fortran User's Guide,* Chapter 1, "Compiling programs."

`-is` *directory*

Instructs the Fortran compiler to attempt inline substitution of each subprogram for which there exists a `.fil` (intermediate-language file) file in the specified *directory*. This option must be repeated for each directory containing `.fil` files to be used for inline substitution. Refer to the *Fortran User's Guide,* Chapter 1, "Compiling programs."

`-mo`

Enables the generation of multi-op instructions, which is default behavior at all optimization levels. Multi-op instructions are single instructions that perform more than one operation, such as multiply/add.

`-nmo`

Disable multi-op instructions.

Options

`-nore`

SPP Series compilers compile all procedures for reentrancy by default (refer to the `-re` option). Fortran procedures can be compiled using the `-nore` compiler option, which causes nonreentrant compilation. Procedures compiled in this manner cannot be parallelized or specified in a `NO_SIDE_EFFECTS` directive or pragma.

Many Fortran compilers, including the Convex C Series Fortran compiler, save local procedure variables by default. If your program is expecting this behavior, default reentrant compilation may cause wrong answers. Compiling the offending procedures with `-nore` should correct the problem.

`-nsr`

Disables scalar replacement.

`-re`

Causes reentrant compilation, which is the default. Reentrant procedures store all local variables on the stack; no values can be carried from one invocation of the procedure to the next. This allows parallel procedure calls.

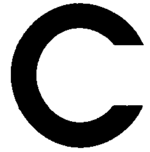
`-sr`

Enables scalar replacement. This is the default at optimization level `-O2` and above. Refer to the section "Optimization options" on page 37.

`-uo`

Performs potentially unsafe optimizations, for example, moving the evaluation of common subexpressions or invariant code from within conditionally executed code. This moved code may be executed unconditionally. Refer to Chapter 9, "Potentially unsafe optimizations."

Optimization report



Optimization report

This appendix provides a complete description of the optimization report produced by the Convex SPP Series C and Fortran compilers. When you compile a program with the `-O2` or `-O3` option, the compiler generates an optimization report for each program unit. The `-or` *table* option determines the report's contents based on the value of *table*, as shown in Table 13.

Table 13 Optimization report contents

Table value	Report contents
`all`	Loop table, privatization table and array table
`loop`	Loop table only (default)
`array`	Array table only
`private`	Loop table and privatization table
`none`	No report

Examples of optimization reports are given in the section "Examples" on page 355.

Loop table

The loop table lists the optimizations that were performed on each loop and, if appropriate, the reasons why a possible optimization was not performed. Loop nests are reported in the order in which they are encountered and separated by a blank line. A description of each column of the loop table follows:

`Line Num.`

Specifies the source line of the beginning of the loop, or of the loop from which it was derived. If the line number has two components separated by a hyphen, the second component is the distributed part (due to loop distribution).

`Id Num.`

Specifies a unique ID number for every loop. This ID number can then be referenced by other parts of the report. Both loops appearing in the original program source and loops created by the compiler are given loop ID numbers; loops created by the compiler are also enumerated in the `New Loops` column as described further on. No distinction between compiler-generated loops and loops that existed in the original source is made in the `Id Num` column; loops are assigned unique, sequential numbers as they are encountered.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop. If the variable is compiler-generated, its name is listed as `*VAR*`; if there is no iteration variable, it is listed as `*NONE*`. If the iteration variable has two parts separated by a colon, the second part is the inline substitution instance of that variable. If it consists of a truncated variable name followed by a colon and a number, the number is a reference to the variable name footnote table which appears after the loop table, analysis table, and test table in the report.

`Reordering Transformation`

Indicates which reordering transformations were performed. Reordering transformations are performed on loops and loop nests, and typically involve reordering and/or duplicating sections of code to facilitate more efficient execution. This column has one of the values shown in Table 14.

Table 14 Reordering transformations reported in opt. report

Value	Explanation
`Serial`	No reordering transformation was performed.
`PARALLEL`	The loop runs in thread-parallel mode.
`PAR-NODE`	The loop runs in node-parallel mode.
`Interchange`	Loop interchange was performed. The new loop order may be indicated under the `Optimizing/Special Transformation` column, as shown in Table 15.
`Dist`	Loop distribution was performed.
`DynSel`	Dynamic selection was performed. The numbers in the `New Loops` column correspond to the loops created; for reentrant parallel loops, these generally include a `PARALLEL` and a `Serial` version.
`Peel`	Loop peeling was performed. In addition to cases in which loops are peeled to remove invariant boundary-iteration assignments, this may appear for automatically parallelized loops when it is necessary to peel the last iteration in order to assign automatically privatized variables their last-iteration value.
`Promote`	Test promotion was performed.
`Unroll and Jam`	Loop unrolling and jamming was performed.
*	Appears at left of loop-producing transformation optimizations (distribution, dynamic selection, peeling, interchange, promotion).

Optimization report

`New Loops`

Specifies the loop ID number(s) for loops created by the compiler. These ID numbers are listed in the `Id Num.` column and can be referenced in other parts of the report; however, the loops they represent were not present in the original source code. In the case of loop fusion, the number in this column indicates the loop into which the current loop was fused.

`Optimizing/Special Transformation`

Indicates which, if any, optimizing transformations were performed. An optimizing transformation reduces the number of operations executed, or replaces operations with simpler operations. A special transformation allows the compiler to optimize code under special circumstances. When appropriate, this column has one of the values shown in Table 15.

Table 15 Optimizing/special transformations in opt. report

Value	Explanation
Unroll	The loop was completely or partially unrolled.
Reduction	The compiler recognized a reduction in the loop.
Pattern	The compiler recognized a special pattern in the loop.
Removed	The compiler removed the loop.
StripMine	The loop was strip mined.
Blocked	The loop was blocked.
Fused	The loop was fused.
Reentrant	Dynamic selection was performed for reentrancy.
Work	Workload-based dynamic selection was performed.
(*oldorder*) -> (*neworder*)	This may appear when `Interchange` is reported under `Reordering Transformation`. *oldorder* indicates the order of loops in the original nest; *neworder* indicates the new order. *oldorder* and *neworder* consist of user iteration variables presented in outermost to innermost order; if user variables are not available, line numbers are used. If the multiple nests begin on the same line, as with compiler-generated loops associated with Fortran 90 array assignments, this information is not reported.

Supplemental tables

The tables described in this section are included if necessary to provide information supplemental to the loop table.

Analysis table

If necessary, an analysis table is included in the optimization report to further elaborate on optimizations reported in the loop table. A description of each column of the analysis table follows:

`Line Num.`

Specifies the source line of the beginning of the loop.

`Id Num.`

References the ID number assigned to the loop in the loop table.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop, `*VAR*`, or `*NONE*`, as described in the "Loop table" section of this appendix.

`Analysis`

Indicates why a transformation or optimization was not performed, or additional information on what was done.

Test table

If any test promotion or removal optimizations were performed, a test table is included in the optimization report. A description of each column in the test table follows:

`Line Num.`

Specifies the source line number of the beginning of the `IF` test.

`Col. Num.`

Specifies the source column number of the beginning of the `IF` test.

`Test Transformation`

Specifies the transformation performed: either `TEST PROMOTED` or `TEST REMOVED`.

`Analysis`

Presents a further explanation of the transformation performed, including the source line number of the original loop from which the test was transformed, and (if applicable), in parentheses, the loop ID number of the compiler-generated loop from which the test was transformed. This ID will be missing if the loop was removed.

Privatization table

This table reports any user variables contained in a parallelized loop that are privatized by the compiler. Because the privatization table refers to loops, the loop table is automatically provided with it. A description of each column in the privatization table follows:

`Line Num.`

Specifies the source line of the beginning of the loop.

`Id Num.`

References the ID number assigned to the loop in the loop table.

`Iter. Var.`

Specifies the name of the iteration variable controlling the loop, `*VAR*`, or `*NONE*`, as described in the section "Loop table" on page 348.

`Priv. Var.`

Specifies the name of the privatized user variable. Compiler-generated variables that are privatized are not reported here.

`Privatization Information for Parallel Loops`

Provides more detail on the privatization performed. For example, this column may indicate that a variable was automatically privatized by the compiler and that its last assignment was peeled from the loop so that its final value could be saved for later use.

Variable name footnote table

Variable names that are too long to fit in the `Iter. Var.` columns of the other tables are truncated and followed by a colon and a footnote number. These footnotes are explained in the variable name footnote table. The headings in the variable name footnote table are explained below.

`Footnoted Iter. Var.`

Specifies the truncated variable name and its footnote number.

`User Variable Name`

Specifies the actual name of the variable as given by the user in the source code.

Array table

The array table lists array references that prevented optimization or array references on which special optimizations were performed. The array table contains the following information:

`Line Num.`

Specifies the source line on which the reference occurs.

`Var. Name`

Specifies the name of the array being referenced.

`Optimization`

Describes the optimizations, if any, performed on the array in question.

`Dependences`

If an array or memory dependence prevented optimization, this column shows the names of variables in the recurrence, in the form *name@linenumber*. If the reference could be to any memory location, it is in the form `*MEM*`*@linenumber*. If the reference is to a subprogram call, it is in the form `*CALL*`*@linenumber*.

Examples

The following Fortran examples enumerate the contents of the optimization report. In discussing the examples, loops are referred to by their ID numbers.

While only Fortran examples are given, analogous C code would produce similar optimization reports.

Example 1

Consider the following loop (line numbers are provided for reference):

```
1       SUBROUTINE EXAMPLE1(A, B, C, N)
2       REAL A(N,N), B(N,N), C(N)
3       DO ILOOPINDEX = 1, N
4         C(ILOOPINDEX) = ILOOPINDEX
5         DO JLOOPINDEX = 1, N
6           A(ILOOPINDEX,JLOOPINDEX) = B(JLOOPINDEX, ILOOPINDEX)
      >     + C(ILOOPINDEX)
7         ENDDO
8       ENDDO
9       END
```

The following output shows the optimization report generated by compiling the program `EXAMPLE1` at optimization level `-O3`.

```
% fc -O3 -or all example1.f

          Optimization for EXAMPLE1

Line      Id   Iter.     Reordering           New       Optimizing / Special
Num.      Num. Var.      Transformation       Loops     Transformation
-----------------------------------------------------------------------------
   3        1  ILOOPI:1 *Dist                 (2-3)     No Strip
   3-1      2  ILOOPI:1 *DynSel               (4-5)     Work Reentrant
   3        4  ILOOPI:1  PARALLEL                       Unroll

   3        5  ILOOPI:1  Serial                         Unroll

   3-2      3  ILOOPI:1 *DynSel               (6-7)     Work Reentrant
   3        6  ILOOPI:1 *Interchange          (8)        (ILOOPI:1 JLOOPI:2
                                                        JLOOPI:2) -> (JLOOPI:2
                                                        ILOOPI:1 JLOOPI:2)
   5        8  JLOOPI:2  PARALLEL                        Blocked Unroll
   3        9  ILOOPI:1  Serial

   3        7  ILOOPI:1 *Interchange          (10)       (ILOOPI:1 JLOOPI:2
                                                        JLOOPI:2) -> (JLOOPI:2
                                                        ILOOPI:1 JLOOPI:2)
   5       10  JLOOPI:2  Serial                          Blocked Unroll
   3       11  ILOOPI:1  Serial

Line      Id   Iter.   Analysis
Num.      Num. Var.
-----------------------------------------------------------------------------
   5        8  JLOOPI:2Loop blocked by 224 iterations
   5       10  JLOOPI:2Loop blocked by 224 iterations
   3        4  ILOOPI:1Replicated: partially unrolled with factor of 8
   3        5  ILOOPI:1Replicated: partially unrolled with factor of 8
   5        8  JLOOPI:2Replicated: partially unrolled with factor of 5
   5       10  JLOOPI:2Replicated: partially unrolled with factor of 5

Line      Id   Iter.    Priv.    Privatization Information
Num.      Num. Var.     Var.     for Parallel Loops
-----------------------------------------------------------------------------
   3        4  ILOOPI:1 ILOOPI:1 Loop induction variable privatized
   3        4  ILOOPI:1 ILOOPI:1 Loop induction variable privatized
   5        8  JLOOPI:2 JLOOPI:2 Loop induction variable privatized
   5        8  JLOOPI:2 ILOOPI:1 Reference privatized
```

Figure 26 Optimization report for Example 1

```
Footnoted   User Variable
Iter. Var.  Name
-----------------------------------------------------------------------
ILOOPI:1    ILOOPINDEX
JLOOPI:2    JLOOPINDEX

            Array References for Procedure EXAMPLE

Line   Var.    Optimi-    Dependences
Num.   Name.   zation
-----------------------------------------------------------------------
   6   C       SR Hoist
   6   C       SR Hoist
   6   C       SR Hoist
   6   C       SR Hoist
```

Figure 26 Optimization report for Example 1 (continued)

Optimization report

Loop table

EXAMPLE1's loop table reports the following:

- Loop number 1 is the loop that appears on line 3 of the source. It iterates over the variable ILOOPINDEX. It was distributed and two new loops, numbers 2 and 3, were created. Loop 1 was not strip mined.

  Note that according to the variable name footnote table, ILOOPINDEX is abbreviated as ILOOPI:1; similarly, JLOOPINDEX is abbreviated as JLOOPI:2. These names are used throughout the report to refer to these iteration variables.

- 3-1, the line number for loop number 2, tells us that it came from the loop at line 3 of the source, and that it is in the first distributed part of that distribution (this is indicated by the -1). Two loops, 4 and 5, are created to facilitate dynamic selection of loop 2. This dynamic selection is workload-based and is required for reentrancy. Loop 4 is the parallel version of loop 2, and loop 5 is the serial version. Both of these loops are unrolled.

- The line number 3-2 for loop 3 tells us that it is the second part of the distribution of the loop appearing on line 3 of the source (loop 2). Like loop 2, both workload-based and reentrant dynamic selection is performed on loop 3, resulting in the creation of loops 6 and 7.

- Loop 6 is interchanged; the `Optimizing/Special Transformation` column shows the loop nest order before and after interchange. There are two `JLOOPINDEX` loops before interchange as a result of the blocking that is reported for loop 8 (optimizations are not always reported in the order in which they are applied). The innermost `JLOOPINDEX` loop is interchanged to outside the `ILOOPINDEX` loop, and this creates loop 8.
- Loop 8 is the final parallel version of the now-outer `JLOOPINDEX` loop. It is blocked and unrolled. Loop 8 contains within it loop 9, which is the `ILOOPINDEX` loop after interchange.
- Loop 7 is interchanged just as loop 6 is, but instead of creating a parallel loop, it creates a serial loop, loop 10.
- Loop 10 is the final serial version of the now-outer `JLOOPINDEX` loop, and includes within it loop 11, which is the `ILOOPINDEX` loop after interchange.

Analysis table

`EXAMPLE1`'s analysis table reports the following:

- Loops 8 and 10 are blocked by 224 iterations.
- Loops 4 and 5 are partially unrolled with an unroll factor of 8.
- Loops 8 and 10 are partially unrolled with an unroll factor of 5.

Privatization table

`EXAMPLE1`'s privatization table reports the following:

- The induction variable `ILOOPINDEX` is privatized in loop 4. This is reported twice because the unrolling of loop 4 creates two loops, but only one of them is shown in the optimization report.
- The induction variable `JLOOPINDEX` is privatized in loop 8.
- The induction variable `ILOOPINDEX` is also privatized in loop 8 because the `ILOOPINDEX` loop is nested within the `JLOOPINDEX` loop in loop 8.

Array reference table

The array reference table indicates that the reference to the C array on line 6 was subject to scalar replacement (`SR`) and the resulting register load was hoisted out of the innermost loop. There are four entries because this optimization occurred for 4 loops (some of which were compiler-generated) during compilation.

Example 2

The following Fortran code provides an example of other transformations the compiler performs. (Line numbers are listed for reference.)

```
1   SUBROUTINE EXAMPLE2(A,N,ZERO,NEGATE,SUM)
2   REAL A(N), SUM
3   LOGICAL ZERO, NEGATE
4
5   SUM = 0.0
6   DO I = 1, N
7     SUM = SUM + A(I)
8     IF (ZERO) THEN
9       A(I) = 0
10    ELSE IF (NEGATE) THEN
11      A(I) = -A(I)
12    ENDIF
13    IF (I .EQ. 1 .OR. I .EQ. N) THEN
14      A(I) = -1
15    ENDIF
16  ENDDO
17  END
```

The following code example shows the optimization report generated by compiling the subroutine EXAMPLE2 above for parallelization:

```
% fc -O3 -or all example2.f

          Optimization for EXAMPLE2

Line       Id    Iter.      Reordering        New        Optimizing / Special
Num.       Num.  Var.       Transformation    Loops      Transformation
-----------------------------------------------------------------------------
    6        1   I         *Promote           (2-3)
    6        2   I         *Promote           (4-5)
    6        4   I         *Peel              (6)
    6        6   I         *DynSel            (7-8)      Work Reentrant
    6        7   I          PARALLEL                     Unroll

    6        8   I          Serial                       Unroll

    6        5   I         *Peel              (9)
    6        9   I         *DynSel            (10-11)    Work Reentrant
    6       10   I          PARALLEL                     Unroll

    6       11   I          Serial                       Unroll

    6        3   I         *Peel              (12)
    6       12   I         *DynSel            (13-14)    Work Reentrant
    6       13   I          PARALLEL                     Unroll

    6       14   I          Serial                       Unroll

Line       Id    Iter.    Analysis
Num.       Num.  Var.
-----------------------------------------------------------------------------
    6        7   I        Replicated: partially unrolled with factor of 8
    6        8   I        Replicated: partially unrolled with factor of 8
    6       10   I        Replicated: partially unrolled with factor of 8
    6       11   I        Replicated: partially unrolled with factor of 8
    6       13   I        Replicated: partially unrolled with factor of 8
    6       14   I        Replicated: partially unrolled with factor of 8

Line      Col.   Test              Analysis
Num.      Num.   Transformation
-----------------------------------------------------------------------------
    8     14     TEST PROMOTED     Test promoted out of loop on 6 (7)
   10     19     TEST PROMOTED     Test promoted out of loop on 6 (7)
   13     16     TEST REMOVED      Peeled first iteration of loop on 6 (13)
   13     16     TEST REMOVED      Peeled first iteration of loop on 6 (7)
   13     16     TEST REMOVED      Peeled first iteration of loop on 6 (10)
   13     30     TEST REMOVED      Peeled last iteration of loop on 6 (13)
   13     30     TEST REMOVED      Peeled last iteration of loop on 6 (7)
   13     30     TEST REMOVED      Peeled last iteration of loop on 6 (10)
```

Figure 27 Optimization report for Example 2

```
Line      Id   Iter.    Priv.    Privatization Information
Num.      Num. Var.     Var.     for Parallel Loops
------------------------------------------------------------------------------
    6       7  I        SUM      Scalar reduction privatized, value saved
    6       7  I        I        Loop induction variable privatized
    6       7  I        SUM      Scalar reduction privatized, value saved
    6       7  I        I        Loop induction variable privatized
    6      10  I        SUM      Scalar reduction privatized, value saved
    6      10  I        I        Loop induction variable privatized
    6      10  I        SUM      Scalar reduction privatized, value saved
    6      10  I        I        Loop induction variable privatized
    6      13  I        SUM      Scalar reduction privatized, value saved
    6      13  I        I        Loop induction variable privatized
    6      13  I        SUM      Scalar reduction privatized, value saved
    6      13  I        I        Loop induction variable privatized

             Array References for Procedure EXAMPLE2

Line   Var.    Optimi-    Dependences
Num.   Name.   zation
------------------------------------------------------------------------------
    7  A       SR Hoist
    7  A       SR Hoist
    7  A       SR Hoist
       .
       .
       .
```

Figure 27 Optimization report for Example 2 (continued)

Loop table

EXAMPLE2's loop table reports the following:

- There is only one loop in this example, and it appears at line 6 in the source. Loops 2 and 3, which are noted under New Loops on loop 1's line, are the result of promoting tests from the original loop; loops 4 and 5 are the result of promoting tests from loop 2.
- Loop 6 is created when loop 4 is peeled. Two loops, 7 and 8, are then created to facilitate workload-based and reentrant dynamic selection of loop 6. Loop 7 is the parallel version of loop 6, and is unrolled. Loop 8 is the serial version of loop 6; it is also unrolled.

Optimization report

- Loops 3 and 5 were both peeled. Peeling loop 5 produces loop 9; peeling loop 3 produces loop 12. Workload-based and reentrant dynamic selection is then applied to both resulting loops. This replicates loop 9 into loops 10 and 11. Loop 12 is replicated into loops 13 and 14.
- Loop 10 is the parallel version of loop 9, and loop 11 is the serial version. Loop 13 is the parallel version of loop 12, and loop 14 is the serial version.

After all the transformations are completed, six loops (ID numbers 7, 8, 10, 11, 13, and 14) remain in the program. These remaining loops can be easily spotted under the `Reordering Transformation` column, as they are the loops that are not marked with the "*" transformation indicator. Loops marked with this symbol no longer exist because they are replaced by the new loops indicated in the `New Loops` column.

Analysis table

The analysis table of this report gives details of the test promotions and peelings that are mentioned in the loop table. It reports that each of the loops remaining after all transformations are performed is unrolled with an unroll factor of 8.

Test table

EXAMPLE2's test table gives details of the test promotions and removals that are indicated in the loop table by the keyword Peel. The test table reports the following:

- The tests on lines 8 and 10 are promoted out of loop 7.
- The tests on line 13 are removed (the .OR. operator implies two tests) by peeling. This optimization is performed for every remaining parallel loop after all the transformations are completed (loops 7, 10, and 13), and it is reported separately for each loop.

Privatization table

For each loop, the table contains an entry indicating that the induction variable I was privatized, and a second entry indicating the privatization of a scalar reduction (involving the variable SUM). This indicates that SUM was accumulated separately in a private copy in each parallel thread, and these copies were automatically added together after the threads joined. Privatization is only performed in parallel loops, so this is only reported for the parallel loops that remain after all transformations are complete (loops 7, 10, and 13).

Array reference table

The references to array A on lines 7, 9 and 11 are subject to scalar replacement and hoisting, and this is reported for every loop which includes these lines, including compiler-generated copies (the report has been shortened for brevity).

Compiler Parallel Support Library

D

Introduction

The Convex Compiler Parallel Support Library (CPSlib) is a library of thread management and synchronization routines that can be used to control parallelism on Exemplar systems. Most programs can fully exploit their parallelism via higher-level devices such as automatic parallelization, compiler directives, and message-passing; CPSlib is provided for those few cases in which a lower-level interface is required. Using CPSlib requires you to manually control all aspects of parallelism, synchronization, and data partitioning.

This appendix includes a discussion of the forms of parallelism available via CPSlib, instructions for accessing CPSlib, a brief description of each of the routines included in CPSlib, and examples of common programming constructs as implemented using CPSlib routines. For further information, refer to the section 3 man pages for the routine in question, or to the cps(3) man page for an overview.

CPSlib supports two forms of parallelism: symmetric and asymmetric.

Symmetric parallelism

In symmetric parallelism, several threads execute the same instruction stream. Symmetric parallelism is typically used by the compilers to parallelize a loop; the description of parallelism given in Chapter 3, "Compiler optimizations," is a description of symmetric parallelism.

Symmetric parallel threads are spawned using `cps_ppcall()` or `cps_ppcalln()`, which, along with all CPSlib routines, are described in detail further on. These functions automatically spawn a given number of threads, which call a specified routine in

CPSlib

parallel. All parallel work must occur in the called routine. When the routine returns, `cps_ppcall()` or `cps_ppcalln()` automatically executes a join and the program proceeds in serial.

Figure 28 shows a `cps_ppcall()` that spawns two threads, each of which in turn spawns four threads. The arrows represent a thread's instruction flow; the numbers labeling the arrows indicate the spawn thread IDs. Kernel and spawn thread IDs are also discussed in the section "Thread ID assignments" on page 212.

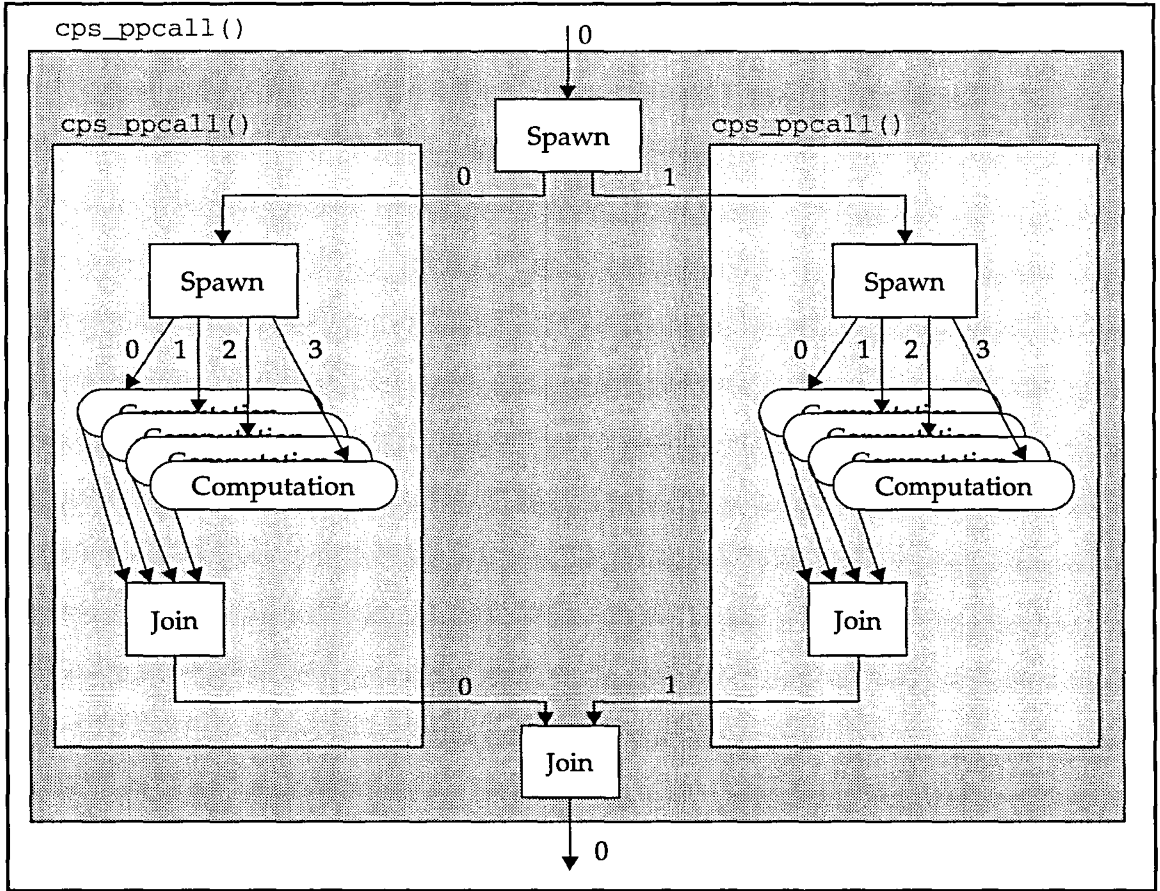


Figure 28 Symmetric parallelism

Shaded boxes represent operations hidden from the user by `cps_ppcall()`. As shown in the left branch of the first spawn, when a `cps_ppcall()` or `cps_ppcalln()` is processed, the parent thread is allocated to the computation as spawn thread ID 0; its kernel thread ID, which is uninteresting to the user, is unchanged. Additional peer threads in both spawned threads are spawned and assigned spawn thread IDs from one to the number of threads spawned minus one.

When the threads join, the original parent thread (spawn thread ID 0) leaves the join after all other threads that were spawned last have also joined. The join operation contains an implicit barrier, so that all threads must reach the join before the original thread continues.

Spawns and joins may thus be arbitrarily nested; however, each thread that was allocated as a result of a spawn must eventually join for correct program operation. The spawn thread ID has a scope between the immediately enclosing spawn/join pair; a single thread may change its spawn thread ID as the result of executing a spawn or join operation.

Asymmetric Parallelism

Asymmetric parallelism is used when each thread executes a different, independent instruction stream. Asymmetric threads are analogous to the UNIX `fork` system call construct; the threads are disjoint. Either the parent or the child may terminate first in any order. Either or both the parent and the child may spawn additional symmetric or asymmetric threads.

Asymmetric threads are spawned using the `cps_thread_create()` function and terminated using the `cps_thread_exit()` function. Asymmetric threads cannot join with their parent thread; they terminate separately. If you do not specifically call `cps_thread_exit()` to terminate an asymmetric thread, the thread will automatically terminate when the procedure called by `cps_thread_create()` successfully terminates.

Figure 29 shows an asymmetric thread tree. Asymmetrically spawned child threads do not have spawn thread IDs; they do, however, have unique kernel thread IDs, which are assigned in no particular order. The parent which executed the `cps_thread_create()` retains the same kernel thread ID it had before it spawned the child thread.

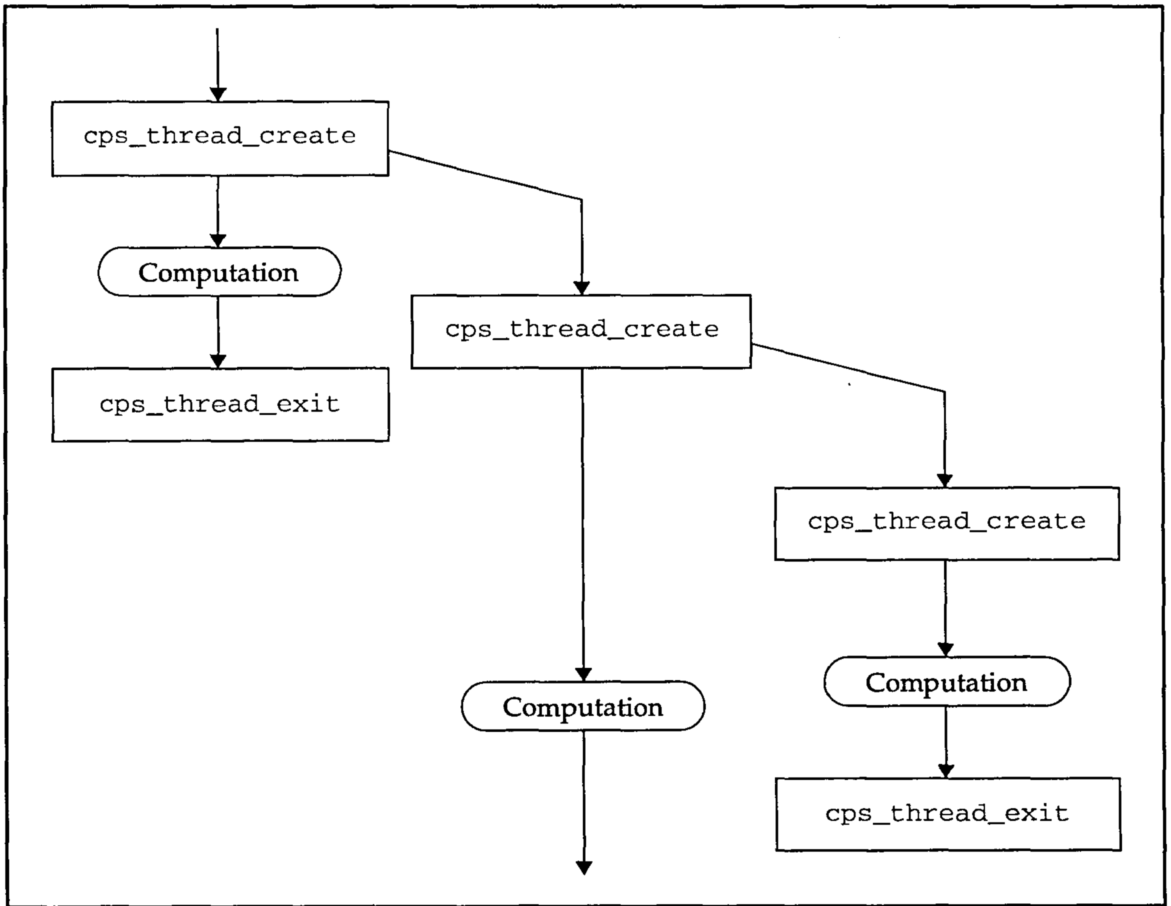


Figure 29 Asymmetric parallelism

You can spawn symmetric threads from asymmetric threads using `cps_ppcall` and `cps_ppcalln`. In this case, the parent thread retains its kernel thread ID and, along with the symmetric threads, receives a spawn thread ID. These symmetric threads must join before the asymmetric parent can exit; if an asymmetric parent attempts to exit while its symmetric children are still active, it will join instead.

Accessing CPSlib

C programs which use CPSlib functions must include the header file cps.h, as shown:

```
#include <cps.h>
```

To access errno symbolic constant values in C, you must also include the header file errno.h:

```
#include <errno.h>
```

C and Fortran share a common interface to CPSlib; the same library, libcps.a, allows access to the CPSlib functions from either language. This library is automatically linked when you compile your program with the Convex Exemplar fc or cc compiler drivers.

CPS library functions

CPSlib provides thread-management functions, high-level synchronization functions, and low-level synchronization functions. This section briefly describes each function and its arguments.

Default versions of the functions presented here return 4-byte values and take 4-byte arguments. 8-byte versions are also available; the names of these functions are suffixed with _8 (for example, the 8-byte version of cps_ppcall is cps_ppcall_8), and the functions take eight byte arguments. Refer to the appropriate man pages for more information.

All C examples presented here assume that cps.h is included in the program. Note that the NULL value in C is equivalent to 0 (zero) in Fortran.

Note

CPSlib routines are generally incompatible with system functions of the form cnx_*; mixing the two may cause wrong answers, deadlock or runtime errors. Mixing CPSlib routines and certain compiler directives may cause wrong answers at -O1 or higher; if this happens, lower the optimization level or refer to the section "Global register allocation" on page 59.

CPSlib

Thread-management functions

These functions allow you to spawn and join or terminate threads.

Symmetric thread functions

Symmetric threads are spawned, execute in parallel, and join via a single CPSlib function call. By definition, all parallel symmetric threads must execute to completion before the join operation can take place; therefore no exit or wait functions are provided.

cps_ppcall and cps_ppcalln

The `cps_ppcall` and `cps_ppcalln` functions allow you to spawn symmetrically parallel threads. In Fortran, these functions have the following forms:

```
INTEGER FUNCTION CPS_PPCALL(PARAMS, FUNC, ARG)
INTEGER PARAMS(4) !ELEMENTS ARE ANALOGOUS TO ELEMENTS OF params
                  !STRUCTURE IN C CODE
EXTERNAL FUNC
INTEGER FUNCTION CPS_PPCALLN(PARAMS,FUNC,n,ARG1,...,ARGn)
INTEGER PARAMS(4) !ELEMENTS ARE ANALOGOUS TO ELEMENTS OF params
                  !STRUCTURE IN C CODE
EXTERNAL FUNC
```

Because the subprogram `FUNC` is used as an actual argument in the function reference, it must be declared `EXTERNAL`.

In C:

```
typedef struct {
  int node;        /* node to place allocated threads on */
  int min;         /* minimum number of threads to allocate */
  int max;         /* maximum number of threads to allocate */
  int threadscope; /* thread scope attributes */
} spawn_sym_t;
int cps_ppcall(spawn_sym_t *params,void (*func)(void *),void *arg);
int cps_ppcalln(spawn_sym_t *params,void (*func)(void *),
              const int *n, void *arg1, ..., void *argn);
```

The elements `params->node` and `PARAMS(1)` control what hypernodes threads are allocated on. They can contain the hypernode ID of the hypernode on which to allocate the threads, or they can take one of the values shown in Table 16.

Table 16 params->node/PARAMS(1) values

C symbolic constant name	Value	Meaning
CPS_SAME_NODE	-1	Allocate threads on same hypernode as calling thread
CPS_ANY_NODE	-2	Allocate threads on any hypernode
CPS_DIFFERENT_NODE	-3	Allocate threads on different hypernode than that of the calling thread

params->min and PARAMS(2) specify the minimum number of threads to allocate; params->max and PARAMS(3) specify the maximum number of threads to allocate.

params->threadscope and PARAMS(4) control the creation strategy for new threads. Table 17 shows acceptable values for these parameters. Except where noted, the logical or of two or more of these values can be used.

Table 17 params->threadscope/PARAMS(4) values

C symbolic constant name	Value	Meaning
CPS_THREAD_PARALLEL	1	Allocate multiple threads per hypernode; mutually exclusive with CPS_NODE_PARALLEL
CPS_NODE_PARALLEL	2	Allocate one thread per hypernode; mutually exclusive with CPS_THREAD_PARALLEL
CPS_OVER_SUBSCRIBE	4	Allows multiple threads per CPU; if not set, no more than one thread per CPU can be started
CPS_IGNORE_STACKSCOPE	8	Allows the spawning of new threads which may not be able to validly address their parent's stack or thread_private data

cps_ppcall spawns the number of threads designated in the argument params and arranges for each thread, including the calling thread, to call the argument function func with the argument arg. After returning from func, each thread automatically joins. When all threads have joined, the parent continues executing the code following the cps_ppcall or cps_ppcalln.

cps_ppcalln is identical to cps_ppcall, except that it will pass *n* arguments to the function func, where *n* ranges from 0 to 256.

The thread that calls `cps_ppcall` or `cps_ppcalln` is considered the parent thread; if the call is successful, it will become spawn thread 0 prior to calling the function. Other threads spawned will be assigned increasing spawn thread IDs ranging from 1 to *m*-1, where *m* is the number of threads spawned.

After all of the threads return from `func`, the parent thread will restore its previous thread state and continue execution after `cps_ppcall` or `cps_ppcalln`.

If successful, these functions return the number of threads spawned including the parent. If an error occurs, they return -1 and, in C, `errno` is set as shown in Table 18.

Table 18 `errno` values for `cps_ppcall` and `cps_ppcalln`

`errno` value	Meaning
EAGAIN	The minimum number of threads could not be allocated
EINVAL	Either the hypernode specified by `params->node` (or `PARAMS(1)`) is not a valid logical hypernode ID, or the value of `params->min` (`PARAMS(2)`) or `params->max` (`PARAMS(3)`) is less than 0

Nested `cps_ppcall` and/or `cps_ppcalln` calls are permitted.

Note

Each thread spawned to run `func` receives a default local stack of 8 Mbytes. If `func` declares local variables that occupy more than 8 Mbytes, you must change this default using the `CPS_STACK_SIZE` environment variable. `CPS_STACK_SIZE` specifies the default stack size for spawned parallel functions in kbytes. It is read once at program startup; the spawn thread stack size cannot be changed during execution. Thread 0's default stack size is specified by SPP-UX; you can modify it via the mpa(1) utility.

For more information, refer to the cps_ppcall(3) man page.

Asymmetric thread functions

By definition, asymmetric threads execute independently of one another. Rather than join, asymmetric threads *terminate*; that is, since their execution is independent, one asymmetric thread has no need to know when another has completed, so no join is necessary or possible.

cps_thread_create and cps_thread_createn

The cps_thread_create and cps_thread_createn functions allow you to spawn asymmetrically parallel threads. In Fortran, these functions have the following form:

```
INTEGER FUNCTION CPS_THREAD_CREATE(NODE, FUNC, ARG)
INTEGER NODE
EXTERNAL FUNC
INTEGER FUNCTION CPS_THREAD_CREATEN(NODE, FUNC, n, ARG1, ..., ARGn)
INTEGER NODEEXTERNAL FUNC
```

Because the subprogram FUNC is used as an actual argument in the function reference, it must be declared EXTERNAL.

In C:

```
int cps_thread_create(const int *node, void (*func) (void *),
                      void *arg);

int cps_thread_createn(const int *node, void (*func) (void*),
                       const int *n, void *arg1, ..., void *argn);
```

cps_thread_create spawns a single asymmetric thread on the hypernode specified by node and arranges for the asymmetric thread to call the argument function func with the parameter arg. On return from func, the asymmetric thread will automatically call cps_thread_exit(); this function can also be manually called from within func to terminate the asymmetric thread.

node takes the same values as the node argument to cps_ppcall, which is described in the section "Symmetric thread functions" on page 370.

cps_thread_createn is identical to cps_thread_create, except that it will pass *n* arguments to the function func, where *n* ranges from 0 to 256.

If successful, these functions return the kernel thread ID of the newly spawned thread. No assumptions can be made about kernel thread IDs except that they are unique. Asymmetric threads do not have spawn thread IDs.

Caution

cps_thread_create and cps_thread_createn return immediately after initiating func, and both func and the parent thread execute in parallel until one terminates. If the parent thread manipulates arg after returning from cps_thread_create[n] but before func exits, you must ensure that this manipulation is synchronized between the parent and func, or the value of arg will be indeterminate.

If unsuccessful, these functions return -1 and, in C, set errno as shown in Table 19.

Table 19 errno values for cps_thread_create[n]

errno value	Meaning
EAGAIN	The asymmetric thread could not be allocated
EINVAL	node is not a valid hypernode ID in the program-assigned subcomplex

Nested calls to cps_thread_create and/or cps_thread_createn are permitted.

For more information, refer to the cps_thread_create(3) man page.

cps_thread_exit

This function terminates the asymmetric thread call made by cps_thread_create. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_THREAD_EXIT()
```

In C:

```
int cps_thread_exit(void);
```

Upon successful completion of func (specified in cps_thread_create), asymmetric threads spawned with cps_thread_create will automatically call cps_thread_exit. The function is provided for cases in which you wish to terminate an asymmetric thread before func normally returns.

If successful, cps_thread_exit does not return. If unsuccessful it returns -1 and, in C, sets errno to EINVAL if it was called from a symmetric thread.

For more information, refer to the cps_thread_create(3) man page.

cps_thread_wait

This function can be used to wait until all asymmetric threads have terminated, or to find out the number of active asymmetric threads. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_THREAD_WAIT(FLAG)
```

In C:

```
int cps_thread_wait(const int *flag);
```

If `flag` is set, this routine waits until all asymmetric threads in the program have terminated. If `flag` is not set, it returns the number of active asymmetric threads in the program.

`cps_thread_wait` cannot be called with `flag` set from an asymmetric thread because the active state of the calling thread will prevent the function from returning, resulting in deadlock.

To use `cps_thread_wait` in an asymmetric thread to wait for all child asymmetric threads to terminate, you must know how many asymmetric threads were spawned before the calling thread. With this information, you can construct a loop that calls `cps_thread_wait` with `flag` equal to 0 until the number of active threads is equal to the number of previously spawned threads plus 1 (the calling thread), as shown in the following Fortran example:

```
10     IWAIT = CPS_THREAD_WAIT(0) ! FIND NUMBER OF ASYM THREADS
       IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
       IF(IWAIT .GT. PREVTHRDS+1) GOTO 10  ! SPIN UNTIL ALL
                                           ! CHILDREN TERMINATE
```

Here, `CPS_THREAD_WAIT` returns the total number of asymmetric threads at line 10; the next line is a routine error trap, and the third line checks the returned number against the known number of threads that are not children of the calling thread. `PREVTHRDS` is a user-defined and user-incremented variable. The loop cannot terminate until the returned number indicates that all of the calling thread's children have terminated.

If unsuccessful, `cps_thread_wait` returns -1 and, in C, sets `errno` to `EDEADLK` if it was called from an asymmetric thread or the child of an asymmetric thread with `flag` set.

For more information, refer to the cps_thread_create(3) man page.

Thread information and attribute functions

These functions provide information on active parallel threads, the subcomplex configuration, and cps attributes. For information beyond that which follows, refer to the cps_info(3) man page.

cps_stid

This function returns the spawn thread ID of the calling thread. In Fortran `cps_stid` has the following form:

```
INTEGER FUNCTION CPS_STID()
```

In C:

```
stid_t cps_stid(void);
```

Spawn thread IDs range from 0..*n*-1, where *n* is the number of currently active symmetric threads in the current spawn context (refer to `cps_nsthreads`).

If unsuccessful, this function returns -1. Because asymmetric threads have no spawn thread IDs, `cps_stid()` returns -1 when called from an asymmetric thread.

cps_ktid

This function returns the kernel thread ID of the calling thread. In Fortran, `cps_ktid` has the following form:

```
INTEGER FUNCTION CPS_KTID()
```

In C:

```
tid_t cps_ktid(void);
```

Note that kernel threads IDs are generated with no regularity; they are simply unique IDs.

cps_nsthreads

This function returns the number of threads in the current cps *spawn context*. Each spawn establishes a spawn context and the number returned by `cps_nsthreads` can vary from spawn to spawn. For example, if you have a `cps_ppcall` that spawns 2 threads nested within a `cps_ppcall` that spawns 4 threads, `cps_nsthreads` returns 2 when called from the inner spawn context, and 4 when called from the outer spawn context.

In Fortran, `cps_nsthreads` has the following form:

```
INTEGER FUNCTION CPS_NSTHREADS()
```

In C:

```
int cps_nsthreads(void);
```

cps_plevel

This function can be used to determine the current level of parallelism. In Fortran, it has the following form:

```
INTEGER FUNCTION CPS_PLEVEL()
```

In C:

```
int cps_plevel(void);
```

The return value is a bit mask. The possible return values are a sum of those shown in Table 20.

Table 20 `cps_plevel` return values

C symbolic constant name	Value	Meaning
CPS_PL_NONE	0	No parallel threads
CPS_PL_PARALLEL	1	Asymmetric parallel threads are active
CPS_PL_NODE	2	Hypernode-parallel threads are active
CPS_PL_NTHREAD	4	Parallel threads are active on the current hypernode
CPS_PL_THREAD	8	Parallel threads are active across multiple hypernodes
CPS_PL_ASYMMETRIC	16	Current thread is an asymmetric thread or a child of one

cps_node_id

This function returns the logical ID of the hypernode on which the calling thread is executing. In Fortran, `cps_node_id` has the following form:

```
INTEGER CPS_NODE_ID()
```

In C:

```
int cps_node_id(void);
```

Logical hypernode IDs range from 0..*n*-1, where *n* is the number of available hypernodes in the subcomplex. Logical IDs are assigned in the order in which your program occupies the subcomplex. The hypernode that your program's thread 0 runs on is considered logical hypernode 0; any hypernodes it expands to later are assigned increasing logical ID numbers. Because SPP-UX starts a program on the least-loaded hypernode, logical hypernode IDs can differ between programs due to load balancing; thus two programs running on the same subcomplex are unlikely to address identical hypernodes with identical logical IDs.

Logical hypernode IDs have no correlation to physical hypernode IDs, which are unique for each hypernode at the machine level.

cps_node_cpus

This function returns the number of threads available to the caller on the hypernode on which it is running. In Fortran it has the following form:

```
INTEGER CPS_NODE_CPUS()
```

In C:

```
int cps_node_cpus(void);
```

Note that the return value represents the number of threads visible to the calling application, and does not necessarily indicate the total number of threads on the hypernode.

cps_node_nthreads

This function returns the number of active threads belonging to the calling application that are running on the hypernode on which the call is executing. In Fortran it has the following form:

```
INTEGER CPS_NODE_NTHREADS()
```

In C:

```
int cps_node_nthreads(void);
```

Active threads include threads spawned by cps_ppcall, cps_ppcalln, or cps_thread_create, as well as threads spawned automatically due to compiler-generated parallelism.

cps_is_parallel

This function returns 1 if the program has parallel code and can go parallel; otherwise it returns 0. In Fortran it has the following form:

```
INTEGER CPS_IS_PARALLEL()
```

In C:

```
int cps_is_parallel(void);
```

Note that the mpa utility can be used to modify attributes of the executable such as whether or not it is parallel. Refer to the mpa(1) man page for more information.

cps_complex_cpus

This function returns the total number of threads available to the application. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_CPUS()
```

In C:

```
int cps_complex_cpus(void);
```

cps_complex_nthreads

This function returns the number of active threads belonging to the calling application that are running on the subcomplex on which the call is executing. In Fortran, it has the following form:

```
INTEGER CPS_COMPLEX_NTHREADS()
```

In C:

```
int cps_complex_nthreads(void);
```

Active threads include threads spawned by cps_ppcall, cps_ppcalln, or cps_thread_create, as well as threads spawned automatically due to compiler-generated parallelism.

cps_complex_nodes

This function returns the total number of logical hypernodes available to the application from which it is called in the application's subcomplex. In Fortran it has the following form:

```
INTEGER CPS_COMPLEX_NODES()
```

In C:

```
int cps_complex_nodes(void);
```

cps_topology

This function fills an array with the application's view of the topology of the subcomplex on which it is running. In Fortran it has the following form:

```
INTEGER CPS_TOPOLOGY(NODES,N)INTEGER NODES(N)
```

In C:

```
int cps_topology(int nodes[], int n)
```

Each element of the `NODES` or `nodes` array corresponds to a logical hypernode, with the first element corresponding to hypernode 0, the second to hypernode 1, etc. Therefore, given default subscripting in Fortran, `NODES(1)` represents hypernode 0. Given default subscripting in C, `nodes[0]` corresponds to hypernode 0. On return from `cps_topology`, each element contains the number of threads running on the corresponding hypernode that belong to the calling application.

`N` or `n` represents the number of elements in the array. If there are more elements than actual hypernodes, the remaining elements are set to 0.

cps_wait_attr

As described in Chapter 3, "Compiler optimizations," idle threads can either be suspended or spin-waiting. This function allows you to get or change the current CPSlib thread wait attributes. In Fortran it has the following form:

```
INTEGER CPS_WAIT_ATTR(CMD, WAIT)
INTEGER CMD, WAIT(2)
```

In C:

```
typedef struct {
  int wait_attr; /* idle wait attribute */
  int wait_time; /*wait time in milliseconds*/
} cps_spin_state_t;
int cps_wait_attr(int *cmd,cps_wait_attr_t
                 *wait);
```

Where `CMD` or `cmd`, depending on its contents, indicates whether to get or change the current wait attributes. Accepted values are shown in Table 21.

Table 21 Accepted `CMD/cmd` values

C symbolic constant name	Value	Meaning
`CPS_GETWAIT`	0	Get the current wait attribute values and store them in the `WAIT` array or in the structure pointed to by `wait`
`CPS_SETWAIT`	1	Set the wait attributes to the values in the `WAIT` array or pointed to by `wait`; these values become effective after the next join or `cps_thread_exit` is executed
`CPS_SETWAITI`	2	Set the wait attributes as `CPS_SETWAIT` does, but force all active threads to join immediately

The `WAIT` array or the structure pointed to by `wait` contain the new or current wait attributes, depending on the value of `cmd`. `WAIT(2)` and `wait->wait_attr` take one of the values shown in Table 22.

CPSlib

Table 22 WAIT(2)/wait->wait_attr values

C symbolic constant name	Value	Meaning
CPS_SUSPEND	1	Suspend the thread after waiting the time specified in WAIT(2) or wait->wait_time
CPS_SPINWAIT	2	Spin-wait until the thread is reactivated

WAIT(2) and wait->wait_time take an integer representing the number of milliseconds the thread is to spin-wait before suspending itself (assuming the wait attribute is set to CPS_SUSPEND). These values default to 50 ms for non-oversubscribed threads and 0 ms for oversubscribed threads.

If unsuccessful, cps_wait_attr returns -1 and, in C, sets errno to EINVAL if:

- wait_time is greater than CPS_WAIT_MAX, and wait_attr is set to CPS_SUSPEND
- wait_attr is not set to CPS_SUSPEND or CPS_SPINWAIT or cmd is not CPS_GETWAIT, CPS_SETWAIT, or CPS_SETWAITI

High-level synchronization functions

These routines manipulate the barriers and mutexes that are used for synchronization. CPSlib barriers can be used to construct barriers such as those that can be constructed with compiler directives as described in Chapter 6, "Advanced shared-memory programming." Mutexes are areas of mutual exclusion, and are analogous to directive-controlled critical sections described in Chapter 6.

These constructs offer a lower degree of automation and a higher degree of control than those directive-specified constructs described in Chapter 6. However, they offer a higher degree of automation and a lower degree of control than the constructs that can be manually built using low-level synchronization functions described in the following section.

All of the functions described in this section return 0 on success and -1 on failure.

For information beyond that which follows, refer to the cps_barrier(3) and cps_mutex(3) man pages.

cps_barrier_alloc

This function allocates the barrier barr and sets its associated shared counter to zero. When each thread reaches the barrier, it increments the counter; when the counter equals the number of parallel threads, all threads may proceed.

In Fortran this function has the form:

```
INTEGER FUNCTION CPS_BARRIER_ALLOC(BARR)
INTEGER BARRIER
```

In C:

```
int cps_barrier_alloc(barrier_t *barr);
```

If this function fails when called from C, errno is set to ENOMEM if the memory required for barr cannot be allocated.

cps_barrier_free

This function releases the barrier barr. In Fortran it has the form:

```
INTEGER FUNCTION CPS_BARRIER_FREE(BARR)
INTEGER BARR
```

In C:

```
int cps_barrier_free(barrier_t *barr);
```

If this function fails when called from C, errno is set to EINVAL if barr was not allocated with a CPSlib barrier allocation function, or to EBUSY if the counter associated with barr is nonzero.

cps_barrier

This function increments the shared counter associated with the barrier barr. When the value of the shared counter is equal to the argument nthreads, the function returns, and the counter is set to zero. In Fortran, cps_barrier has the following form:

```
INTEGER FUNCTION CPS_BARRIER(BARR,NTHREADS)
INTEGER BARR, NTHREADS
```

In C:

```
int cps_barrier(barrier_t *barr, const int
                *nthreads);
```

If this function fails when called from C because barr was not allocated with a CPSlib barrier allocation function, errno is set to EINVAL.

cps_mutex_alloc

This function allocates the mutex mutx and unlocks it. In Fortran it has the form:

```
INTEGER FUNCTION CPS_MUTEX_ALLOC(MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_alloc(cps_mutex_t *mutx);
```

This function does not check whether mutx is already allocated; therefore, when successful, it always allocates a new mutex. When it fails, mutx is set to NULL (in Fortran, MUTX is set to 0).

If this function fails when called from C, it sets errno to ENOMEM if it cannot allocate the required memory.

cps_mutex_free

This function releases the mutex mutx. In Fortran it has the form:

```
INTEGER FUNCTION CPS_MUTEX_FREE(MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_free(cps_mutex_t *mutx);
```

If this function is successful when called from C, mutx is set to NULL. In Fortran, MUTX is undefined.

If unsuccessful when called from C, it sets errno to EINVAL if mutx was not allocated by a CPSlib allocation function, or to EBUSY if mutx has already been acquired.

cps_mutex_lock

If the mutex mutx is unlocked, this function acquires it and returns; if it is locked by another thread, cps_mutex_lock will wait until it is acquired before returning.

In Fortran, cps_mutex_lock has the following form:

```
INTEGER FUNCTION CPS_MUTEX_LOCK(MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_lock(cps_mutex_t *mutx);
```

If the calling thread has already acquired mutx this function returns -1 and, in C, sets errno to EDEADLK.

cps_mutex_unlock

This function releases the mutex mutx so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION CPS_MUTEX_UNLOCK(MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_unlock(cps_mutex_t *mutx);
```

cps_mutex_trylock

This function attempts to acquire mutx; if mutx is already locked by another thread, cps_mutex_trylock will return -1.

In Fortran, cps_mutex_trylock has the following form:

```
INTEGER FUNCTION CPS_MUTEX_TRYLOCK(MUTX)
INTEGER MUTX
```

In C:

```
int cps_mutex_trylock(cps_mutex_t *mutx);
```

If the calling thread has already acquired mutx this function returns -1 and, in C, sets errno to EDEADLK.

Low-level synchronization functions

These functions manipulate the counters and semaphores used for low-level synchronization. These functions require you to create and manually control your own synchronization semaphores.

Semaphores can be cache- or memory-based. Cache-based semaphores can be stored in the processor data cache, making them best for use in situations that generate minimal semaphore contention. Memory based semaphores are never brought into the processor data cache, making them preferable in situations that generate semaphore contention.

Because each semaphore has an associated counter, it can be used both as a lock (to implement, for example, a critical section) and a counter (to implement, for example, an ordered section).

For information beyond that which follows, refer to the cps_sema(3) man page.

c_init32

This function allocates the cache-based semaphore cs and initializes its associated counter to value. In Fortran it has the following form:

```
INTEGER FUNCTION C_INIT32(CS, VALUE)
INTEGER CS, VALUE
```

In C:

```
int c_init32(cache_sema_t *cs,const int *value);
```

If successful, c_init32 returns the counter value; otherwise it returns -1.

c_free32

This function frees the cache-based semaphore cs and sets it to NULL on success. In Fortran it has the following form:

```
INTEGER FUNCTION C_FREE32(CS)
INTEGER CS
```

In C:

```
int c_free32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_lock

This function acquires a cache-based semaphore cs. In Fortran it has the following form:

```
INTEGER FUNCTION C_LOCK(CS)
INTEGER CS
```

In C:

```
int c_lock(cache_sema_t *cs);
```

If the semaphore is already acquired, c_lock will wait until the semaphore is released before returning. It may or may not give up the processor in the interim.

c_unlock

This function releases the cache-based semaphore cs so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION C_UNLOCK(CS)
INTEGER CS
```

In C:

```
int c_unlock(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_cond_lock

If the cache-based semaphore cs is available, this function acquires it; otherwise, it returns -1 without waiting and allows execution to continue in the calling thread. In Fortran it has the following form:

```
INTEGER FUNCTION C_COND_LOCK(CS)
INTEGER CS
```

In C:

```
int c_cond_lock(cache_sema_t *cs);
```

c_fetch32

This function returns the value of the counter associated with the cache-based semaphore cs. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH32(CS)
INTEGER CS
```

In C:

```
int c_fetch32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_fetch_and_inc32

This function increments the value of the counter associated with the semaphore cs and returns the old value.

In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_INC32(CS)
INTEGER CS
```

In C:

```
int c_fetch_and_inc32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_fetch_and_dec32

This function decrements the value of the counter associated with the semaphore cs and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_DEC32(CS)
INTEGER CS
```

In C:

```
int c_fetch_and_dec32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_fetch_and_clear32

This function returns the current value of the counter associated with the semaphore cs and clears the counter. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_CLEAR32(CS)
INTEGER CS
```

In C:

```
int c_fetch_and_clear32(cache_sema_t *cs);
```

If unsuccessful, this function returns -1.

c_fetch_and_add32

This function adds value to the counter associated with the semaphore cs and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_ADD32(CS, VALUE)
INTEGER CS, VALUE
```

In C:

```
int c_fetch_and_add32(cache_sema_t *cs,
                      const int *value);
```

If unsuccessful, this function returns -1.

c_fetch_and_set32

This function returns the current value of the counter associated with the semaphore cs, and sets the semaphore to the new value contained in newval. In Fortran it has the following form:

```
INTEGER FUNCTION C_FETCH_AND_SET32(CS, NEWVAL)
INTEGER CS, NEWVAL
```

In C:

```
int c_fetch_and_set32(cache_sema_t *cs,
                      const int *newval);
```

If unsuccessful, this function returns -1.

m_init32

This function allocates the memory-based semaphore ms and initializes the counter associated with it to value. In Fortran it has the following form:

```
INTEGER FUNCTION M_INIT32(MS,VALUE)
INTEGER MS, VALUE
```

In C:

```
int m_init32(mem_sema_t *ms,const int *value);
```

If successful, this function returns the counter value; otherwise it returns -1.

m_free32

This function releases the memory-based semaphore ms and sets it to NULL on success. In Fortran it has the following form:

```
INTEGER FUNCTION M_FREE32(MS)
INTEGER MS
```

In C:

```
int m_free32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_lock

This function acquires the memory-based semaphore ms.If the semaphore is already acquired, m_lock will wait until the semaphore is released before returning. It may or may not give up the processor in the interim. In Fortran it has the following form:

```
INTEGER FUNCTION M_LOCK(MS)
INTEGER MS
```

In C:

```
int m_lock(mem_sema_t *ms);
```

m_unlock

This function releases the memory-based semaphore ms so that other threads may acquire it. In Fortran it has the following form:

```
INTEGER FUNCTION M_UNLOCK(MS)
INTEGER MS
```

In C:

```
int m_unlock(mem_sema_t *ms)
```

If unsuccessful, this function returns -1.

m_cond_lock

If the memory-based semaphore ms is available, this function acquires it; otherwise it returns -1 without waiting and allows execution to continue in the calling thread. In Fortran it has the following form:

```
INTEGER FUNCTION M_COND_LOCK(MS)
INTEGER MS
```

In C:

```
int m_cond_lock(mem_sema_t *ms);
```

m_fetch32

This function returns the value of the counter associated with the memory-based semaphore ms. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH32(MS)
INTEGER MS
```

In C:

```
int m_fetch32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_fetch_and_inc32

This function increments the value of the counter associated with the semaphore ms and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_INC32(MS)
INTEGER MS
```

In C:

```
int m_fetch_and_inc32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_fetch_and_dec32

This function decrements the value of the counter associated with the semaphore ms and returns the old value. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_DEC32(MS)
INTEGER MS
```

In C:

```
int m_fetch_and_dec32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

m_fetch_and_clear32

This function returns the current value of the counter associated with the semaphore ms and clears the counter. In Fortran it has the following form:

```
INTEGER FUNCTION M_FETCH_AND_CLEAR32(MS)
INTEGER MS
```

In C:

```
int m_fetch_and_clear32(mem_sema_t *ms);
```

If unsuccessful, this function returns -1.

sync_routine directive and pragma

Among the most basic optimizations performed by the Convex SPP Series compiler is code motion, which is described in Chapter 3, "Compiler optimizations." This optimization can move some code across routine calls. If the routine call is to a synchronization or parallelization function and the code moved must execute on a certain side of it, this movement can cause wrong answers. Anytime you use CPSlib functions rather than the directives or functions described in Chapter 4, "Basic shared-memory programming," and Chapter 6, "Advanced shared-memory programming," to synchronize or parallelize code, you must identify the functions with a sync_routine directive or pragma. sync_routine should be used to identify all CPSlib functions, as well as any user-written routines that accomplish synchronization or parallelization or hide calls to any synchronization or parallelization routines.

In Fortran, sync_routine has the following form:

C$DIR SPP SYNC_ROUTINE (*routinelist*)

In C, it has the following form:

#pragma _CNX SPP sync_routine (*routinelist*)

where *routinelist* is a comma-delimited list of synchronization procedures.

sync_routine is only effective for the listed routines that lexically follow it in the routine in which it appears.

CPSlib

Consider the following Fortran example:

```
      SUBROUTINE WORK(ARG1, ARG2, MUTX)
      INTEGER ARG1, ARG2, MUTX, CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK
C$DIR SYNC_ROUTINE(CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK)
      .
      .
      .
      DO I = 1, N
        .
        .
        .
        LCK = CPS_MUTEX_LOCK(MUTX)
        .
        .
        .
        LCK = CPS_MUTEX_UNLOCK(MUTX)
      ENDDO
      .
      .
      .
      END
```

Here, the subroutine WORK is called in parallel and contains a loop that contains a critical section protected by calls to CPSlib functions. Listing these CPSlib functions in a SYNC_PARALLEL directive at the beginning of the subroutine prevents the compiler from moving code out of the critical section.

An analogous C example follows:

```
#include <spp_prog_model.h>
work(int arg1, int arg2, int mutx) {
  int i, lck;
#pragma _CNX sync_routine(cps_mutex_lock, cps_mutex_unlock)
   .
   .
   .
#pragma _CNX loop_parallel(ivar=i)
  for(i=0; i<n; i++) {
     .
     .
     .
    lck = cps_mutex_lock(&mutx);
     .
     .
     .
    lck = cps_mutex_unlock(&mutx);
  }
}
```

Examples

The examples presented here demonstrate various constructs that can be programmed using the CPSlib functions described in the previous sections.

You can compile a program that uses CPSlib to achieve parallelism at any optimization level, however, you must pass appropriate flags to the linker to indicate that the program is parallel. This is done using the `-Wl,` option, which indicates that the comma-delimited options following it up to the next space are to be passed to the linker. You can pass either the `+min` and `+max` options to indicate the minimum and maximum number of CPUs on which to run the program, or the `+parallel` option to indicate that the program is parallel and that any available CPUs should be used.

The following example `fc` command line compiles the program `cpspar.f` at optimization level `-O2` and indicates to the linker that the program should run on a minimum of 2 and a maximum of 8 processors:

```
% fc -Wl,+min,2,+max,8 -O2 cpspar.f
```

The compiler does not recognize CPSlib parallelism, so while this command line would generate an optimization report, the report would only detail compiler optimizations; it would not report manually-coded parallelism. To see if your executable is parallel, use the `file` command.

You can also use the `mpa` utility to modify many attributes, including parallelism, of an executable file after compilation. Refer to the mpa(1) man page for more information.

CPSlib

Symmetric parallelism

There are two forms of symmetric parallelism: block parallelism, and cyclic parallelism. The CPSlib functions used in the examples that follow are described in detail in the section "CPS library functions" on page 369.

Block parallelism

Block parallelism is the most commonly used form of parallelism; it is the form generated by default by Convex SPP Series compilers. It involves splitting up the iterations of a loop into iteration blocks of similar size, and running each block on a separate processor.

A simple Fortran example that uses CPSlib to implement block parallelism follows. The CPSlib functions used here are described in detail in the "CPS library functions" section.

```
      PROGRAM CPSBLOCK
      REAL X(1000), Y(1000), Z(1000)
      INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS)
      EXTERNAL PARBLK ! REQUIRED BECAUSE PARBLK IS AN ARGUMENT
C INITIALIZE PARGS ARRAY:
      PARGS(1) = -2  ! ALLOCATE THREADS ON CALLING THREAD'S NODE
      PARGS(2) = 2   ! MINIMUM OF 2 THREADS
      PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM # OF THREADS
      PARGS(4) = 1       ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
C SPAWN THREADS:
      ITHREAD = CPS_PPCALLN(PARGS, PARBLK, 4, X, Y, Z, NTHR)
C IF SPAWN FAILS, REPORT:
      IF (ITHREAD .LT. 0) PRINT *,'PPCALLN FAILED'
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE PARBLK (X, Y, Z, NTHR)
      REAL X(1000), Y(1000), Z(1000)
      INTEGER CPS_NSTHREADS, CPS_STID, STID, NTHR
C$DIR SYNC_ROUTINE(CPS_NSTHREADS, CPS_STID)
      STID = CPS_STID()       ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND
        Z(J) = X(J) + Y(J)
      ENDDO
      RETURN
      END
```

This example calls CPS_NODE_CPUS to find the number of available threads, then calls CPS_PPCALLN to spawn parallel threads to run the subroutine PARBLK.

PARBLK then determines the number of iterations necessary per processor; if the number of processors does not integrally divide the number of iterations, it automatically adjusts some blocks to handle the extra iterations. Finally, the loop in PARBLK performs its body in parallel, with each thread operating on the appropriate iteration range.

Note the error trap immediately after the call to CPS_PPCALLN; this is important, as it provides the only means of knowing if the spawn failed.

Cyclic parallelism

Cyclic parallelism distributes consecutive iterations of a loop to separate processors. It is similar to the parallelism achieved through use of the `loop_parallel(ordered)` directive and pragma, but it does not order the iterations automatically; you must handle any necessary ordering manually. `loop_parallel(ordered)` is discussed in Chapter 6, "Advanced shared-memory programming," on page 221.

A simple Fortran example that uses CPSlib to implement cyclic parallelism follows. The CPSlib functions used here are described in detail in the "CPS library functions" section.

```
      PROGRAM CPSCYCLE
      REAL X(1000), Y(1000), Z(1000), SUM
      INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS)
      EXTERNAL PARCYC ! PARCYC IS AN ARGUMENT
      READ *, NPROCSC INITIALIZE PARGS ARRAY:
      PARGS(1) = -2  ! ALLOCATE THREADS ON CALLING THREAD'S NODE
      PARGS(2) = 2   ! MINIMUM OF 2 THREADS
      PARGS(3) = CPS_NODE_CPUS() ! MAXIMUM # OF THREADS
      PARGS(4) = 1      ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
C SPAWN THREADS:
      ITHREAD = CPS_PPCALLN (PARGS, PARCYC, 4, X, Y, Z, NTHR)
C IF SPAWN FAILS, REPORT:
      IF (ITHREAD .LT. 0) PRINT *,'PPCALLN FAILED'
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE PARCYC (X, Y, Z, NTHR)
      REAL X(1000), Y(1000), Z(1000)
      INTEGER CPS_NSTHREADS, CPS_STID, STID, NTHR
C$DIR SYNC_ROUTINE(CPS_NSTHREADS, CPS_STID)
      STID = CPS_STID()       ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
C ACTUAL COMPUTATION:
      DO J = 1+STID, 1000, NTHR ! STEP BY NUMBER OF THREADS
         Z(J) = X(J) + Y(J)
      ENDDO
      RETURN
      END
```

This example works exactly like the block parallelism example, except the loop in PARCYC, its parallel subroutine, is cyclically parallel. Cyclic parallelism is accomplished here by offsetting the loop start value by spawn thread ID and stepping the loop by the

number of parallel threads. This ensures that each thread computes a unique array element on every step of the loop; NTHR elements are computed per step. Contiguous STIDs compute contiguous elements.

Asymmetric parallelism

A simple Fortran program that implements asymmetric parallelism follows. The CPSlib functions used here are described in detail in the "CPS library functions" section.

```
      PROGRAM ASYM
      REAL X1(1000), X2(1000), Y1(1000), Y2(1000), Z(1000)
      INTEGER CPS_THREAD_CREATE, CPS_THREAD_WAIT
C$DIR SYNC_ROUTINE(CPS_THREAD_CREATE, CPS_THREAD_WAIT)
      COMMON /POINTS/ X1, X2, Y1, Y2
      EXTERNAL DISTANCE ! DISTANCE IS AN ARGUMENT
      .
      . ! SERIAL CODE
      . ! EXAMPLE CONTINUED
C SPAWN ASYMMETRIC THREAD TO EXECUTE SUBROUTINE DISTANCE:
      ITHREAD = CPS_THREAD_CREATE(-2, DISTANCE, Z)
      IF (ITHREAD .LT. 0) PRINT*, "THREAD_CREATE FAILED IN MAIN"
      .
      . ! THIS CODE RUNS IN PARALLEL WITH DISTANCE
      .
      IWAIT = CPS_THREAD_WAIT(1) ! WAIT FOR ALL ASYMMETRIC
                                 ! THREADS TO TERMINATE
      IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
      .
      . ! THIS CODE RUNS SERIALLY AFTER PARALLEL THREADS
      . ! TERMINATE
      END

      SUBROUTINE DISTANCE(Z)
      REAL X1(1000), X2(1000), Y1(1000), Y2(1000), Z(1000)
      REAL X3(1000), Y3(1000)
      INTEGER CPS_THREAD_CREATE, CPS_THREAD_WAIT
C$DIR SYNC_ROUTINE(CPS_THREAD_CREATE, CPS_THREAD_WAIT)
      COMMON /POINTS/ X1, X2, Y1, Y2
      EXTERNAL FINDX ! FINDX IS AN ARGUMENT
C SPAWN ASYMMETRIC THREAD TO EXECUTE SUBROUTINE FINDX:
      JTHREAD = CPS_THREAD_CREATE(-2, FINDX, X3)
      IF (JTHREAD .LT. 0) PRINT*, "THREAD_CREATE FAILED IN DISTANCE"
      DO I = 1, 1000 ! COMPUTE Y3 IN PARALLEL WITH FINDX
        Y3(I) = (Y2(I) - Y1(I))**2
      ENDDO
```

CPSlib

```
10    IWAIT = CPS_THREAD_WAIT(0) ! FIND NUMBER OF ASYM THREADS
      IF(IWAIT .LT. 0) PRINT*, "CPS_THREAD_WAIT FAILED"
      IF(IWAIT .GT. 1) GOTO 10   ! SPIN UNTIL ONLY THIS THREAD
                                 ! IS ACTIVE
      DO I = 1, 1000 ! COMPUTE Z SERIALLY AFTER X3 AND Y3
        Z(I) = SQRT(X3(I) + Y3(I))
      ENDDO
      RETURN
      END

      SUBROUTINE FINDX(X3) ! RUNS IN PARALLEL WITH COMPUTATION
                           ! OF Y3
      REAL X1(1000), X2(1000), Y1(1000), Y2(1000), X3(1000)
      COMMON /POINTS/ X1, X2, Y1, Y2
      DO I = 1, 1000
        X3(I) = (X2(I) - X1(I))**2
      ENDDO
      RETURN
      END
```

In this example, the arrays X3 and Y3 must be computed before the array Z can be computed. The main program spawns an asymmetric parallel thread to run DISTANCE, which spawns an asymmetric thread to run FINDX. DISTANCE then computes Y3 while FINDX computes X3; all the while, the main program can be doing other work in parallel with both subroutines. When DISTANCE is done with Y3, it waits until FINDX is done with X3, then computes Z. The main program waits until DISTANCE is done, then proceeds with more work.

Note the way in which CPS_THREAD_WAIT is used when called from DISTANCE; this is explained further in the "CPS library functions" section.

Synchronization using high-level functions

This section demonstrates how to use barriers and mutexes to synchronize symmetrically parallel code.

Barriers

Remember that, when you use `cps_ppcall()` to spawn symmetric parallelism, before the function returns, a join operation takes place after all spawned threads terminate. This join is an implicit barrier, since thread 0 cannot proceed until all parallel threads terminate. In many cases, this is the only barrier synchronization you will require.

However, the `cps_barrier()` high-level synchronization functions allow you to explicitly create barriers if necessary.

The following Fortran example is similar to the symmetric parallelism example in the section "Block parallelism" on page 393 except that instead of relying on the implicit barrier contained in the call to `CPS_PPCALL()`, it contains an explicit `CPS_BARRIER()` in the subroutine `SUMMER`.

```
      PROGRAM BAR
      REAL A(1000)
      REAL SUM(:), TOTSUM
      INTEGER PARGS(4), SUMBAR, CPS_NODE_CPUS, CPS_PPCALLN
      INTEGER CPS_BARRIER_ALLOC, CPS_BARRIER_FREE
C$DIR SYNC_ROUTINE(CPS_BARRIER_ALLOC,CPS_BARRIER_FREE)
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS,CPS_PPCALLN)
      EXTERNAL SUMMER
      ALLOCATABLE(SUM)
      NCPUS = CPS_NODE_CPUS()
      ALLOCATE(SUM(0:NCPUS-1)) ! ONE ELEMENT FOR EACH CPU
      PARGS(1) = -2      ! ALLOCATE THREADS ON CALLING THREAD'S NODE
      PARGS(2) = 2       ! MINIMUM OF 2 THREADS PER NODE
      PARGS(3) = NCPUS   ! MAXIMUM # OF THREADS PER NODE
      PARGS(4) = 1       ! ALLOCATE MULTIPLE THREADS PER NODE
      DO I = 0, NCPUS-1 ! INITIALIZE SUM
        SUM(I) = 0.0
      ENDDO
      .
      .  ! SERIAL CODE
      .
      IERR = CPS_BARRIER_ALLOC(SUMBAR) ! ALLOCATE BARRIER
      IF (IERR .LT. 0) PRINT*, "BARRIER ALLOCATION FAILED"
```

CPSlib

```
C SPAWN PARALLEL THREADS:
      IERR = CPS_PPCALLN(PARGS,SUMMER,5,A,SUM,TOTSUM,SUMBAR,NCPUS)
      IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
      IERR = CPS_BARRIER_FREE(SUMBAR) ! FREE BARRIER
      IF (IERR .LT. 0) PRINT*, "BARRIER FREE FAILED"
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE SUMMER(A,SUM,TOTSUM,SUMBAR,NCPUS)
      INTEGER  STID, NTHR, SUMBAR
      INTEGER CPS_STID, CPS_NSTHREADS, CPS_BARRIER
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHREADS, CPS_BARRIER)
      REAL A(1000), SUM(0:NCPUS-1), TOTSUM
      STID = CPS_STID()        ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND ! EACH THREAD COMPUTES LOCAL SUM
        SUM(STID) = SUM(STID) + A(J)
      ENDDO
C WAIT UNTIL ALL THREADS ARE DONE COMPUTING THEIR PORTION OF SUM:
      IERR = CPS_BARRIER(SUMBAR, NTHR)
      IF (IERR .LT. 0) PRINT*, "BARRIER FAILED"
      IF(STID .EQ. 0) THEN ! THREAD 0 COMPUTES TOTAL SUM
        DO I = 0, NTHR-1
          TOTSUM = TOTSUM + SUM(I)
        ENDDO
      ENDIF
      RETURN
      END
```

Here, the subroutine SUMMER is called in parallel to compute the sum of the elements of array A. Each parallel thread computes its own sum in an element of the array SUM. The CPS_BARRIER function is used to prevent execution of any further code until all

threads have finished computing their portion of SUM. When CPS_BARRIER returns, thread 0 computes TOTSUM, and SUMMER returns.

Mutexes

CPSlib mutexes allow you to limit access to the sections of code they delimit to one thread at a time, allowing you to construct critical sections similar to those discussed in Chapter 4, "Basic shared-memory programming."

In the following Fortran example, the routine SUMMER performs the same task it did in preceding barrier example. However, here access to the TOTSUM computation takes place in fully parallel code; it is limited to one thread at a time by the mutex SUMMUTEX. This eliminates the need for each thread to compute independent SUM arrays as in the preceding barrier example.

```
      PROGRAM MUT
      REAL A(1000)
      REAL TOTSUM
      INTEGER PARGS(4), SUMMUTEX
      INTEGER CPS_NODE_CPUS,CPS_PPCALLN
      INTEGER CPS_MUTEX_ALLOC,CPS_MUTEX_FREE
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS,CPS_PPCALLN)
C$DIR SYNC_ROUTINE(CPS_MUTEX_ALLOC,CPS_MUTEX_FREE)
      EXTERNAL SUMMER
      NCPUS = CPS_NODE_CPUS()
      PARGS(1) = -2       ! ALLOCATE THREADS ON CALLING THREAD'S NODE
      PARGS(2) = 2        ! MINIMUM OF 2 THREADS PER NODE
      PARGS(3) = NCPUS    ! MAXIMUM # OF THREADS PER NODE
      PARGS(4) = 1        ! ALLOCATE MULTIPLE THREADS PER NODE
      TOTSUM = 0.0        ! INITIALIZE TOTSUM
      .
      .    ! SERIAL CODE
      .
      IERR = CPS_MUTEX_ALLOC(SUMMUTEX)   ! ALLOCATE MUTEX
      IF (IERR .LT. 0) PRINT*, "MUTEX ALLOCATION FAILED"
C SPAWN PARALLEL THREADS:
      IERR = CPS_PPCALLN(PARGS,SUMMER,3,A,TOTSUM,SUMMUTEX)
      IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
      IERR = CPS_MUTEX_FREE(SUMMUTEX)    ! FREE MUTEX
      IF (IERR .LT. 0) PRINT*, "MUTEX FREE FAILED"
      .
      .    ! SERIAL CODE
      .
      END
```

CPSlib

```
      SUBROUTINE SUMMER(A,TOTSUM,SUMMUTEX)
      INTEGER  STID, NTHR, SUMMUTEX
      INTEGER CPS_STID, CPS_NSTHREADS
      INTEGER CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHREADS)
C$DIR SYNC_ROUTINE(CPS_MUTEX_LOCK, CPS_MUTEX_UNLOCK)
      REAL A(1000),TOTSUM
      STID = CPS_STID()       ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR  ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR  ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND
C MUTEX LIMITS ACCESS TO TOTSUM TO ONE THREAD AT A TIME:
        IERR = CPS_MUTEX_LOCK(SUMMUTEX)
        IF (IERR .LT. 0) PRINT*, "MUTEX LOCK FAILED"
        TOTSUM = TOTSUM + A(J)
        IERR = CPS_MUTEX_UNLOCK(SUMMUTEX)
        IF(IERR .LT. 0) PRINT*, "MUTEX UNLOCK FAILED"
      ENDDO
      RETURN
      END
```

Here, as in the barrier example, SUMMER is called in parallel. Each parallel thread then waits until it can lock SUMMUTEX before updating TOTSUM.

Synchronization using low-level functions

This section demonstrates how to use semaphores to synchronize symmetrically parallel code.

Critical sections

Critical sections like the one in the preceding mutex example can be implemented in a similar fashion using cache-based or memory-based semaphores.

The following Fortran example is identical to the mutex example, but implements the critical section using a memory-based semaphore instead of a mutex:

```
      PROGRAM SEM
      REAL A(1000)
      REAL TOTSUM
      INTEGER PARGS(4), SUMSEM, SEMCNT
      INTEGER CPS_NODE_CPUS, CPS_PPCALLN, M_INIT32, M_FREE32
C$DIR SYNC_ROUTINE(CPS_NODE_CPUS, CPS_PPCALLN, M_INIT32, M_FREE32)
      EXTERNAL SUMMER
      NCPUS = CPS_NODE_CPUS()
      PARGS(1) = -2         ! ALLOCATE THREADS ON CALLING THREAD'S NODE
      PARGS(2) = 2          ! MINIMUM OF 2 THREADS PER NODE
      PARGS(3) = NCPUS      ! MAXIMUM # OF THREADS PER NODE
      PARGS(4) = 1          ! ALLOCATE MULTIPLE THREADS PER NODE
      TOTSUM = 0.0          ! INITIALIZE TOTSUM
      SEMCNT = 0 ! COUNTER FOR SEMAPHORE; VALUE IS IRRELEVANT
      .
      .  ! SERIAL CODE
      .
      IERR = M_INIT32(SUMSEM,SEMCNT) ! ALLOCATE SUMSEM
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE ALLOCATION FAILED"
      IERR = CPS_PPCALLN(PARGS,SUMMER,3,A,TOTSUM,SUMSEM)
      IF (IERR .LT. 0) PRINT*, "PPCALL FAILED"
      IERR = M_FREE32(SUMSEM)
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE FREE FAILED"
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE SUMMER(A,TOTSUM,SUMSEM)
      INTEGER  STID, NTHR, SUMSEM
      INTEGER CPS_STID, CPS_NSTHREADS, M_LOCK, M_UNLOCK
      REAL A(1000),TOTSUM
```

CPSlib

```
C$DIR SYNC_ROUTINE(CPS_STID, CPS_NSTHREADS, M_LOCK, M_UNLOCK)
      STID = CPS_STID()        ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
      ITPERPROC = 1000/NTHR   ! COMPUTE ITERATIONS PER THREAD
      IEXCESS = 1000-ITPERPROC*NTHR   ! COMPUTE EXCESS ITERATIONS
C COMPUTE LOOP START AND END FOR CASES OF NO EXCESS OR FOR THREADS
C THAT DO NOT HANDLE EXCESS:
      IF(STID .GE. IEXCESS) THEN
        MYSTART = IEXCESS*(ITPERPROC+1)+(STID-IEXCESS)*ITPERPROC+1
        MYEND = MYSTART + ITPERPROC - 1
C COMPUTE LOOP START AND END FOR THREADS THAT HANDLE EXCESS:
      ELSE
        MYSTART = STID * (ITPERPROC+1) + 1
        MYEND = MYSTART + (ITPERPROC+1) - 1
      ENDIF
C ACTUAL COMPUTATION:
      DO J = MYSTART, MYEND
C SEMAPHORE LIMITS ACCESS TO TOTSUM TO ONE THREAD AT A TIME:
        IERR = M_LOCK(SUMSEM)
        IF (IERR .LT. 0) PRINT*, "SEMAPHORE LOCK FAILED"
        TOTSUM = TOTSUM + A(J)
        IERR = M_UNLOCK(SUMSEM)
        IF(IERR .LT. 0) PRINT*, "SEMAPHORE UNLOCK FAILED"
      ENDDO
      RETURN
      END
```

Here as in the mutex example, `SUMMER` is called in parallel. Each parallel thread then waits until it can lock the memory-based semaphore `SUMSEM` before updating `TOTSUM`.

Ordered sections

Semaphores can also be used to construct ordered sections such as those constructed using the `loop_parallel(ordered)`, `begin_ordered_section` and `end_ordered_section` directives and pragmas, which are described in Chapter 6, "Advanced shared-memory programming."

The parallel loop in the following Fortran example contains a backward LCD, which is isolated using low-level synchronization functions so that the threads must execute the LCD in iteration order.

```
      PROGRAM ORDERED ! DEMONSTRATES ORDERED SECTIONS USING CPS
                      ! LOW LEVEL SYNCHRONIZATION
      REAL X(1000), Y(1000)
      INTEGER PARGS(4), CPS_PPCALLN, NTHR, CPS_NODE_CPUS
      INTEGER M_INIT32, M_FREE32
C$DIR SYNC_ROUTINE(CPS_PPCALLN, CPS_NODE_CPUS, M_INIT32, M_FREE32)
      INTEGER ORDSEM, SEMCNT
      EXTERNAL ORDWORK
      PARGS(1) = -2   ! ALLOCATE THREADS CALLING THREAD'S NODE
      PARGS(2) = 2    ! MINIMUM OF 2 THREADS
      PARGS(3) = CPS_NODE_CPUS()   ! MAXIMUM OF NPROCS THREADS
      PARGS(4) = 1    ! ALLOCATE MULTIPLE THREADS PER HYPERNODE
      SEMCNT = 0
      .
      .  ! SERIAL CODE
      .
      IERR = M_INIT32(ORDSEM,SEMCNT) ! ALLOCATE ORDSEM
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE ALLOCATION FAILED"
C SPAWN THREADS:
      ITHREAD = CPS_PPCALLN(PARGS,ORDWORK,5,X,Y,NTHR,ORDSEM,SEMCNT)
      IF (ITHREAD .LT. 0) PRINT *,"PPCALLN FAILED"
      IERR = M_FREE32(ORDSEM)
      IF (IERR .LT. 0) PRINT*, "SEMAPHORE FREE FAILED"
      .
      .  ! SERIAL CODE
      .
      END

      SUBROUTINE ORDWORK (X, Y, NTHR,ORDSEM,SEMCNT)
      REAL X(1000), Y(1000)
      INTEGER CPS_NSTHREADS, CPS_STID, M_FETCH32
      INTEGER ORDSEM,SEMCNT,STID, NTHR, CNTVAL
      INTEGER M_FETCH_AND_INC32, M_FETCH_AND_CLEAR32
C$DIR SYNC_ROUTINE(CPS_NSTHREADS, CPS_STID, M_FETCH32)
C$DIR SYNC_ROUTINE(M_FETCH_AND_INC32, M_FETCH_AND_CLEAR32)
      STID = CPS_STID()       ! GET MY STID
      NTHR = CPS_NSTHREADS() ! GET NUMBER OF THREADS SPAWNED
```

CPSlib

```
C ACTUAL COMPUTATION:
      DO J = 2+STID, 1000, NTHR ! CYCLIC DECOMPOSITION
        .
        .  ! DEPENDENCE-FREE PARALLEL CODE
        .
10      CNTVAL = M_FETCH32(ORDSEM)  ! GET SEMAPHORE COUNTER VALUE
        IF(CNTVAL .EQ. STID) THEN   ! IF IT'S MY STID'S TURN
C PERFORM LCD COMPUTATION:
          X(J) = X(J-1) + Y(J)
          IF(CNTVAL .GE. NTHR-1) THEN            ! HIGHEST STID
            IERR = M_FETCH_AND_CLEAR32(ORDSEM)  ! RESETS COUNTER
            IF(IERR .LT. 0) PRINT*, "FETCH-CLEAR FAILED"
          ELSE ! ALL OTHER STIDS INCREMENT COUNTER:
            IERR = M_FETCH_AND_INC32(ORDSEM)
            IF(IERR .LT. 0) PRINT*, "FETCH-INC FAILED"
          ENDIF
        ELSE
          GOTO 10 ! LOOP AND TRY AGAIN IF CNTVAL .NE. STID
        ENDIF
      ENDDO
      RETURN
      END
```

This example uses a cyclic decomposition in the parallel `J` loop because, by definition, ordered sections must be executed in iteration order, and this is impossible using a block decomposition.

As in the example in the "Cyclic parallelism" section, here the starting index is offset according to spawn thread ID and the loop steps by the number of parallel threads. This ensures that each thread computes a unique array element on every step of the loop; `NTHR` elements are computed per step. Contiguous `STID`s compute contiguous elements.

The loop is ordered by the first `IF` statement in the loop, which only allows the body of the loop (including the LCD) to execute if the counter associated with the semaphore `ORDSEM` is equal to the current `STID`. This counter is incremented (or reset when the highest `STID` is reached) in the body of the loop, forcing the threads to execute in iteration order. The counter associated with `ORDSEM` controls access to the LCD; no explicit semaphore lock is needed.

Substantial nonordered work must be present in this loop to make the overhead of the ordered section worthwhile. Assuming this condition is met, once all the threads pass through the ordered section once, their execution of the nonordered code will be staggered such that they will stay busy while they are outside the ordered code.

The ordered parallelism described here is similar to that achieved through use of compiler directives in the "Ordered sections" section of Chapter 6, "Advanced shared-memory programming."

Glossary

A

ABI
Application Binary Interface. A software implementation that allows the same executable programs to run on computer systems with different hardware architectures. SPP Series systems are implemented with an ABI that allows compatibility with other computer systems that run Hewlett-Packard's HP-UX operating system.

absolute address
An address that does not undergo virtual-to-physical address translation when used to reference memory or the I/O register area.

accumulator
A variable used to accumulate value. Accumulators are typically assigned a function of themselves, and this can create dependences when done in loops.

actual argument
In Fortran, a value that is passed by a call to a procedure (function or subroutine). The actual argument appears in the source of the calling procedure; the argument that appears in the source of the called procedure is a *dummy argument*. C conventions refer to actual arguments as *actual parameters*.

actual parameter
In C, a value that is passed by a call to a procedure (function). The actual parameter appears in the source of the calling procedure; the parameter that appears in the source of the called procedure is a *formal parameter*. Fortran conventions refer to actual parameters as *actual arguments*.

address
A number used by the operating system to identify a storage location.

address space
Memory space, either physical or virtual, available to a process.

agent
The gate array on SPP Series systems that provides a high-speed interface between the pairs of PA-RISC processors in a functional block and the *crossbar*. Also called the *CPU Agent* and the *CPA*.

alias
An alternative name for some object, especially an alternative variable name that refers to a memory location. Aliases can cause data dependences, which prevent the compiler from parallelizing parts of a program.

alignment
A condition in which the address, in memory, of a given data item is integrally divisible by a particular integer value, often the size of the data item itself. Alignment simplifies the addressing of such data items.

ALL
Acronym for Assembler, Loader, and Libraries.

allocatable array
In Fortran 90, a named array whose rank is specified at compile time, but whose bounds are determined at run time.

allocate
An action performed by a program at runtime in which memory is reserved to hold data of a given type. In Fortran 90, this is done through the creation of *allocatable arrays*. In C, it is done through the dynamic creation of memory blocks using `malloc`.

ALU
Arithmetic logic unit. A basic element of the central processing unit (CPU) where arithmetic and logical operations are performed.

Amdahl's law

A statement that the ultimate performance of a computer system is limited by the slowest component. In the context of SPP systems this is interpreted to mean that the serial component of the application code will restrict the maximum speed-up that is achievable.

American National Standards Institute (ANSI)

A repository and coordinating agency for standards implemented in the U.S. Its activities include the production of Federal Information Processing (FIPS) standards for the Department of Defense (DoD).

ANSI

See *American National Standards Institute.*

apparent recurrence

A condition or construct that fails to provide the compiler with sufficient information to determine whether or not a recurrence exists. Also called a *potential recurrence.*

Application Binary Interface (ABI)

A software implementation that allows the same executable programs to run on computer systems with different hardware architectures. SPP Series systems are implemented with an ABI that allows compatibility with other computer systems that run Hewlett-Packard's HP-UX operating system.

argument

In Fortran, either a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called (*dummy argument*) or the variable or constant that is passed by a call to a procedure (*actual argument*). C conventions refer to arguments as *parameters.*

arithmetic logic unit (ALU)

A basic element of the central processing unit (CPU) where arithmetic and logical operations are performed.

array

An ordered structure of operands of the same data type. The structure of an array is defined by its rank, shape, and data type.

array section

A Fortran 90 construct that defines a subset of an array by providing starting and ending elements and strides for each dimension. For an array `A(4,4)`, `A(2:4:2,2:4:2)` is an array section containing only the evenly indexed elements `A(2,2)`, `A(4,2)`, `A(2,4)`, and `A(4,4)`.

array-valued argument

In Fortran 90, an *array section* that is an actual argument to a subprogram.

ASCII

American Standard Code for Information Interchange. This encodes printable and non-printable characters into a range of integers.

assembler

A program that converts assembly language programs into executable machine code.

assembly language

A programming language whose executable statements can each be translated directly into a corresponding machine instruction of a particular computer system.

automatic array

In Fortran, an array of explicit rank that is not a dummy argument and is declared in a subprogram.

B

balancing

Expressions are represented internally as trees whose height corresponds to the depth of the expression. Tree-height reduction or balancing is an optimization that attempts to simplify expressions by shortening their tree height.

bandwidth

A measure of the rate at which data can be moved through a device or circuit. Bandwidth is usually measured in millions of bytes per second (Mbytes/sec) or millions of bits per second (Mbits/sec).

bank conflict

An attempt to access a particular memory bank before a previous access to the bank is complete.

barrier

A structure used by the compiler in barrier synchronization. Also sometimes used to refer to the construct used to implement barrier synchronization. See also *barrier synchronization*.

barrier synchronization

A control mechanism used in parallel programming that ensures all threads have completed an operation before continuing with the next operation. On SPP Series machines, barrier synchronization can be automated by certain CPSlib routines and compiler directives. See also *barrier.*

basic block

A linear sequence of machine instructions with a single entry and a single exit.

bit

A binary digit.

block-shared memory

Memory that is addressed by the same virtual address from any hypernode in the subcomplex on which the memory was allocated. This memory class is used to store arrays that are dynamically allocated at runtime, when the number of hypernodes on which the process is running is known. The virtual pages of the arrays are then divided into a number of chunks equal to the number of available hypernodes, and these chunks (which likely contain multiple contiguous pages each) are distributed to the subcomplex-global physical pages of the available hypernodes, 1 chunk per hypernode. If the number of pages of a `block_shared` array is not integrally divisible by the number of hypernodes, the array size is increased to allow integral division. Compare with *node-private memory, thread-private memory, near-shared memory,* and *far-shared memory.*

blocking factor

Integer representing the stride of the outer strip of a pair of loops created by blocking.

branch

A class of instructions which change the value of the program counter to a value other than that of the next sequential instruction.

byte

A group of contiguous bits starting on an addressable boundary. Generally, a byte is 8 bits in length.

C

cache

A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the main memory that are, or are believed to be, currently in use. Cache memory is physically separate from main memory, and can be accessed with substantially less latency. The SPP Series of computers employs separate data and instruction cache memories.

cache, direct mapped

A form of cache memory that addresses encached data by a function of the data's virtual address. On SPP1000 Series and SPP1600 Series computers, the cache address is identical to the least-significant 20 bits of the data's virtual address. This means cache thrashing can occur when the virtual addresses of two data items are an exact multiple of 1 Mbyte (20 bits) apart. On SPP1200 Series computers, the main cache address is identical to the least-significant 18 bits of the data's virtual address. Cache thrashing can therefore occur on SPP1200 machines when the virtual addresses of two data items are an exact multiple of 256 kbytes (18 bits) apart.

cache, fully associative

A form of cache memory that attempts to prevent encached data from being overwritten by storing an incoming element in a cache location that is determined using a hashing algorithm. The incoming element's virtual address is irrelevant. Unlike a direct mapped cache, a fully associative cache will never displace a cache line unless the cache is full. SPP1200 Series and SPP1600 Series machines use a 2 kbyte on-chip fully associative assist cache to mitigate cache thrashing.

cache hit

A *cache hit* occurs if data to be loaded is residing in the cache.

cache line

A chunk of contiguous data that is copied into a cache in one operation. On SPP Series computers, processor cache lines consist of 32 bytes of data; when a processor cache miss occurs and data must be fetched from outside the processor cache, the requested data is brought in as part of a 32-byte cache line. CTIcache lines consist of 64 bytes of data (two contiguous processor cache lines); when a CTIcache miss occurs, the requested data is brought into the CTIcache as part of a 64-byte cache line.

cache memory
A small, high-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of the main memory that are, or are believed to be, currently in use. Cache memory is physically separate from main memory, and can be accessed with substantially less latency. SPP Series computers employ separate data and instruction cache memories.

cache miss
A *cache miss* occurs if data to be loaded is not residing in the cache.

cache purge
The act of invalidating or removing entries in a cache memory.

cache thrashing
Cache thrashing occurs when two or more data items that are frequently needed by the program map to the same cache address. In this case, each time one of the items is encached it overwrites another needed item, causing constant cache misses and impairing data reuse.

CCMC
The Convex Coherent Memory Controller gate array, which, among other tasks, provides an interface to the CTI interface and to memory and maintains cache coherency information. Each pair of processors has a CCMC.

central processing unit (CPU)
The central processing unit (CPU) is that portion of a computer that recognizes and executes the instruction set.

clock cycle
The duration of the square wave pulse sent throughout a computer system to synchronize operations. The clock cycle time for SPP1000 Series systems is 10 nanoseconds; for SPP1200 Series and SPP1600 Series systems, it is 8.33 nanoseconds.

clone
A compiler-generated copy of a loop or procedure. When the SPP Series procedural compilers generate code for a parallelizable loop, they generate two versions: a serial clone and a parallel clone. See also *dynamic selection.*

code
A computer program, either in source form or in the form of an executable image on a machine.

coherency

A term frequently applied to caches. If a data item is referenced by a particular processor on a multiprocessor system, the data is copied into that processor's cache and is updated there if the processor modifies the data. If another processor references the data while a copy is still in the first processor's cache, a mechanism is needed to ensure that the second processor does not use an outdated copy of the data from memory. The state that is achieved when both processors' caches always have the latest value for the data is called cache coherency. On SPP Series computers, an item of data may reside concurrently in several processors' caches.

column-major order

Memory representation of an array such that the columns are stored contiguously. For example, given a two-dimensional array `A(3,4)`, the array element `A(3,1)` immediately precedes element `A(1,2)` in memory. This is the default storage method for arrays in Fortran.

compiler

A computer program that translates computer code written in a high-level programming language, such as Fortran, into equivalent machine language.

Compiler Parallel Support library (CPSlib)

A library of low-level parallelization and synchronization routines. Refer to Appendix D, "Compiler Parallel Support Library," for more information.

complex

The complete set of processor and memory resources available on an SPP Series system.

concurrent

In parallel processing, threads that can execute at the same time are called concurrent threads.

conditional induction variable

A loop induction variable that is not necessarily incremented on every iteration.

constant folding

Replacement of an operation on constant operands with the result of the operation.

constant propagation
The automatic compile-time replacement of variable references with a constant value previously assigned to that variable. Constant propagation is performed within a single procedure by conventional compilers.

control parallel programming
A type of parallel programming in which different functional sections of code are assigned to different processors for simultaneous execution. See also *data parallel programming*.

conventional compiler
A compiler that cannot perform interprocedural optimization. This term encompasses all Convex compilers and virtually all compilers available from other vendors.

counter
A variable that is used to count the number of times an operation occurs. CPSlib semaphores are associated with counters so that they can facilitate barrier synchronization or the creation of ordered sections.

CPA
CPU Agent. The gate array on SPP Series systems that provides a high-speed interface between the pairs of PA-RISC processors in a functional block and the *crossbar*. Also called the *CPU Agent* and the *agent*.

CPSlib (Compiler Parallel Support library)
A library of low-level parallelization and synchronization routines. Refer to Appendix D, "Compiler Parallel Support Library," for more information.

CPU
Central processing unit. The central processing unit (CPU) is that portion of a computer that recognizes and executes the instruction set.

CPU Agent
The gate array on SPP Series systems that provides a high-speed interface between the pairs of PA-RISC processors in a functional block and the *crossbar*. Also called the *agent* and the *CPA*.

CPU-private memory

Data that is accessible by a single thread only (not shared among the threads constituting a process). A thread-private data object has a unique virtual address which maps to a unique physical address within each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses. Compare with *node-private memory*, *near-shared memory*, *far-shared memory*, and *block-shared memory*.

CPU time

The amount of time the CPU requires to execute a program. Because programs share access to a CPU, the wall clock time of a program may not be the same as its CPU time. If a program can use multiple processors, the CPU time may be greater than the wall clock time. (See *wall clock time*.)

critical section

A portion of a parallel program that can be executed by only one thread at a time.

crossbar

A switching device that connects the CPUs, banks of memory, and I/O controller on a single hypernode of an SPP Series system. Because the crossbar is nonblocking, all ports can run at full bandwidth simultaneously, provided there is not contention for a particular port.

CSR

Control/Status Register. A CSR is a software-addressable hardware register used to hold control information or state.

CTIcache

A partition of physical memory that exists on each hypernode and is used to store copies of global data fetched from other hypernodes.

CTI interface

The hardware interface between the CTI rings and the CCMC.

CTI (Coherent Toroidal Interface) ring

The ring interconnect that connects all the hypernodes of a multihypernode SPP Series system together in a ring topology. While the CTI ring is derived from the IEEE SCI standard, complete compatibility is sacrificed to provide lower latencies.

D

data cache (Dcache)

A small cache memory with a one clock cycle access time. This cache holds prefetched and current data. On SPP1000 Series systems, it is 1 Mbyte in size. SPP1200 Series systems have a 2 kbyte on-chip cache and a 256 kbyte off-chip cache. SPP1600 Series systems have a 2 kbyte on-chip cache and a 1 Mbyte off-chip cache. See also *cache, fully associative* and *cache, direct mapped.*

data dependence

A relationship between two statements in a program, such that one statement must precede the other to produce the intended result. (See also *loop-carried dependence (LCD)* and *loop-independent dependence (LID).*)

data localization

Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

data parallel programming

A type of parallel programming in which the data upon which an operation is to be performed is divided among several threads, which execute concurrently in separate processors.

data type

A property of a data item which determines how its bits are grouped and interpreted. For processor instructions, the data type identifies the size of the operand and the significance of the bits in the operand. Some example data types include `INTEGER`, `int`, `REAL`, and `float`.

Dcache

Data cache. A small cache memory with a one clock cycle access time. This cache holds prefetched and current data. On SPP1000 and SPP1600 systems, this cache is 1 Mbyte in size. On SPP1200 systems, it is a 256 kbyte cache.

deadlock

A condition in which a thread waits indefinitely for some condition or action that cannot, or will not, occur.

direct memory access (DMA)

A method for gaining direct access to main memory and achieving data transfers without involving the CPU.

distributed memory
A memory architecture used in multi-CPU systems, in which the system's memory is physically divided among the processors. In most distributed-memory architectures, distributed memory is accessible from only a single processor; sharing of data requires explicit message passing. Convex's GSM is an exception. See also *global memory, globally shared memory (GSM).*

distributed part
A loop generated by the compiler in the process of loop distribution.

DMA
Direct memory access. A method for gaining direct access to main memory and achieving data transfers without involving the CPU.

double
A double-precision floating-point number that is stored in 64 bits in C.

doubleword
A primitive data operand which is 8 bytes (64 bits) in length. Also called a *longword*. See also *word*.

dummy argument
In Fortran, a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called. The dummy argument appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an *actual argument*. C conventions refer to dummy arguments as *formal parameters*.

dynamic selection
The process by which the compiler chooses the appropriate runtime clone of a loop. See also *clone*.

E

encache
To copy data or instructions into a cache.

exception
A hardware-detected event that interrupts the running of a program, process, or system. See also *fault*.

execution stream
A series of instructions executed by a CPU.

F

far-shared memory

Memory that is addressed by the same virtual address from any hypernode in the subcomplex on which the memory was allocated. Far-shared memory is physically distributed by pages, in a manner that is approximately round-robin, to all the hypernodes in the subcomplex, so the virtual address maps to a single physical address located on one of the hypernodes. Access latencies therefore vary as a function of hypernode and data element. Compare with *node-private memory*, *thread-private memory*, *near-shared memory*, and *block-shared memory*.

fault

A type of *interruption* caused by an instruction which requests a legitimate action which cannot be carried out immediately due to a system problem.

floating point

A numerical representation of a real number. On SPP Series computers, a floating point operand has a sign (positive or negative) part, an exponent part, and a fraction part. The fraction is a fractional representation. The exponent is the value used to produce a power of two scale factor (or portion) that is subsequently used to multiply the fraction to produce an unsigned value.

FLOPS

Floating-point operations per second. A standard measure of computer processing power in the scientific community.

formal parameter

In C, a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called. The formal parameter appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an *actual parameter*. Fortran conventions refer to formal parameters as *dummy arguments*.

Fortran

A high-level software language used mainly for scientific applications.

Fortran 90
The international standard for Fortran adopted in 1991.

function
A procedure whose call can be imbedded within another statement, such as an assignment or test. Any procedure in C or a procedure defined as a `FUNCTION` in Fortran.

functional unit (FU)
A part of a *CPU* that performs a set of operations on quantities stored in *registers*.

G

gate
A construct that restricts execution of a block of code to a single thread. A thread locks a gate on entering the gated block of code and unlocks the gate on exiting the block. When the gate is locked, no other threads can enter. Compiler directives can be used to automate gate constructs; gates can also be implemented using *semaphores*.

Gbyte
See *gigabyte*.

gigabyte
1073741824 (2^{30}) bytes.

global optimization
A restructuring of program statements that is not confined to a single basic block. Global optimization, unlike interprocedural optimization, is confined to a single procedure. Global optimization is done by all Convex compilers at optimization level `-O1` and above.

global memory
Memory that is accessible from several processors. See also *distributed memory, globally shared memory (GSM).*

global register allocation (GRA)
A method by which the compiler attempts to store commonly-referenced scalar variables in registers throughout the code in which they are most frequently accessed.

global variable
A variable whose scope is greater than a single procedure. In C programs, a global variable is a variable that is defined outside of any one procedure. Fortran has no global variables per se, but `COMMON` blocks can be used to make certain memory locations globally accessible.

global virtual memory
Globally shared memory. A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory. On SPP Series computers, globally shared memory is distributed among the hypernodes, but any hypernode's memory is accessible from any processor on any hypernode. This type of memory is sometimes referred to as *shared virtual memory* or *globally shared memory.*

globally shared memory (GSM)
A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory. On SPP Series computers, globally shared memory is distributed among the hypernodes, but any hypernode's memory is accessible from any processor on any hypernode. This type of memory is sometimes referred to as *shared virtual memory* or *global virtual memory.*

granularity
In the context of parallelism, a measure of the relative size of the computation done by a thread or parallel construct. Performance is generally an increasing function of the granularity. In higher-level language programs, possible sizes are routine, loop, block, statement, and expression. Fine granularity is exhibited by parallel loops, tasks and expressions; coarse granularity is exhibited by parallel processes.

H

hand-rolled loop
A loop, more common in Fortran than C, that is constructed using `IF` tests and `GOTO` statements rather than a language-provided loop structure such as `DO`.

hidden alias
An alias that, because of the structure of a program or the standards of the language, goes undetected by the compiler. Hidden aliases can result in undetected *data dependences,* which may result in wrong answers.

High Performance Fortran (HPF)
An ad-hoc language extension of Fortran 90 that provides user-directed data distribution and alignment. HPF is not a standard, but rather a set of features desirable for parallel programming.

hoist
An optimization process that moves a memory load operation from within a loop to the basic block preceding the loop.

host

In the context of PVM, a computer system into which communications hardware and software have been installed, and which is accessed as a separate entity by other hosts.

HP

Hewlett-Packard, the manufacturer of the PA-RISC chips used as CPUs in SPP Series systems.

HP-UX

Hewlett-Packard's UNIX-based operating system for its PA-RISC workstations and servers.

hypercube

A topology used in some massively parallel processing systems. Each processor is connected to its binary neighbors. The number of processors in the system is always a power of two; that power is referred to as the dimension of the hypercube. For example, a 10-dimensional hypercube has 2^{10}, or 1,024 processors.

hypernode

In SPP Series systems, a set of up to eight processors and physical memory organized as a symmetric multiprocessor (SMP) running a single image of the operating system microkernel. An SPP system consists of one or more hypernodes, with a high speed *CTI ring* connecting the hypernodes. When discussing multidimensional parallelism or memory classes, hypernodes are generally called nodes.

I

Icache

Instruction cache. On SPP1000 and SPP1600 systems, a 1-Mbyte cache memory with a one clock cycle access time. This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On SPP1200 systems it is 256 kbytes in size.

IEEE

Institute for Electrical and Electronic Engineers. An international professional organization and a member of ANSI and ISO.

induction variable

A variable that changes linearly within the loop, that is, whose value is incremented by a constant amount on every iteration. For example, in the following Fortran loop, `I`, `J` and `K` are induction variables, but `L` is not.

```
DO I = 1, N
   J = J + 2
   K = K + N
   L = L + I
ENDDO
```

inlining

The replacement of a procedure (function or subroutine) call, within the source of a calling procedure, by a copy of the called procedure's code.

Institute for Electrical and Electronic Engineers (IEEE)

An international professional organization and a member of ANSI and ISO.

instruction

One of the basic operations performed by a CPU.

instruction cache (Icache)

On SPP1000 and SPP1600 systems, a 1-Mbyte cache memory with a one clock cycle access time. This cache holds prefetched instructions and permits the simultaneous decoding of one instruction with the execution of a previous instruction. On SPP1200 systems it is 256 kbytes in size.

instruction mnemonic

A symbolic name for a machine instruction.

integral division

Division that results in a whole number solution with no remainder. For example, 10 is integrally divisible by 2, but not by 3.

interface

A logical path between any two modules or systems.

interleaved memory

Memory that is divided into multiple banks to permit concurrent memory accesses. The number of separate memory banks is referred to as the memory stride.

interprocedural optimization
Automatic analysis of relationships and interfaces between all subroutines and data structures within a program. Traditional compilers analyze only the relationships within the procedure being compiled.

interprocessor communication
The process of moving or sharing data, and synchronizing operations between processors on a multiprocessor system.

intrinsic
A function or subroutine that is an inherent part of a computer language. For example, `SIN` is a Fortran intrinsic.

J

job scheduler
That portion of the operating system that schedules and manages the execution of all processes.

join
The synchronized termination of parallel execution by spawned tasks or threads.

jump
Departure from normal one-step incrementing of the program counter.

K

kbyte
See *kilobyte*.

kernel
The core of the operating system where basic system facilities, such as file access and memory management functions, are performed.

kernel thread identifier (ktid)
A unique integer identifier (not necessarily sequential) assigned when a thread is created.

kilobyte
1024 (2^{10}) bytes.

L

large application
An application requiring a virtual address space larger than 4 Gbytes.

latency
The time delay between the issuing of an instruction and the completion of the operation. A common benchmark used for comparing SPP systems is the latency of coherent memory access instructions. This particular latency measurement is believed to be a good indication of the *scalability* of an SPP system; low latency equates to low system overhead as system size increases.

libpvm
The C programming library (libpvm3.a), Fortran programming library (libfpvm3.a), or group library (libgpvm3.a), supporting the PVM message-passing programming style.

linker
A software tool that combines separate object code modules into a single object code module or executable program.

load
An instruction used to move the contents of a memory location into a register.

locality of reference
An attribute of a memory reference pattern that refers to the likelihood of an address of a memory reference being physically close to the CPU making the reference.

local optimization
Restructuring of program statements within the scope of a basic block. Local optimization is done by all Convex compilers at optimization level -O0 and above.

localization
Data localization. Optimizations designed to keep frequently used data in the processor data cache, thus eliminating the need for more costly CTIcache or main memory accesses.

logical address
Logical address space is that address as seen by the application program.

logical memory
Virtual memory. The memory space as seen by the program, which may be larger than the available physical memory. The virtual memory of an SPP Series computer can be up to 4 Gbytes (however, through use of node-private memory, this 4 Gbytes can be mapped to a *larger* set of physical memory). SPP-UX can map this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called *virtual memory*.

longword (l)
Doubleword. A primitive data operand which is 8 bytes (64 bits) in length. See also *word*.

loop blocking
A loop transformation that strip mines and interchanges a loop to provide optimal reuse of the encachable loop data.

loop-carried dependence (LCD)
A dependence between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. A loop carries a dependence from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores into an address that is referred to on a different iteration of the loop. To parallelize a loop containing an LCD, you generally must manually synchronize the LCD assignment and manually parallelize the loop.

loop constant
A constant or expression whose value does not change within a loop.

loop distribution
The restructuring of a loop nest to create simple loop nests. Loop distribution creates two or more loops, called distributed parts, which can serve to make parallelization more efficient by increasing the opportunities for loop interchange and isolating code that must run serially from parallelizable code. It can also improve data localization and other optimizations.

loop-independent dependence (LID)
A dependence between two operations executed on the same iteration of all enclosing loops such that one operation must precede the other to produce correct results.

loop induction variable

A variable that changes linearly within the loop, that is, whose value is incremented by a constant amount on every iteration. For example, in the following Fortran loop, I, J and K are induction variables, but L is not.

```
DO I = 1, N
   J = J + 2
   K = K + N
   L = L + I
ENDDO
```

loop interchange

The reordering of nested loops. Loop interchange is generally done to increase the granularity of the parallelizable loop(s) present or to allow more efficient access to loop data.

loop invariant

Loop constant. A constant or expression whose value does not change within a loop.

loop invariant computation

An operation that yields the same result on every iteration of a loop.

loop replication

The process of transforming one loop into more than one loop to facilitate an optimization. The optimizations that replicate loops are IF-DO and if-for optimizations, dynamic selection, loop unrolling, unroll and jam, and loop blocking.

M

machine exception

A fatal error in the system that cannot be handled by the operating system. See also *exception*.

main memory

On SPP Series systems, physical memory other than the processor caches that is not allocated as part of the CTIcache.

main procedure

A procedure invoked by the operating system when an application program starts up. The main procedure is the main program in Fortran; in C, it is the function main().

main program
In a Fortran program, the program section invoked by the operating system when the program starts up.

Massively Parallel Processor (MPP)
A computer architecture in which several to thousands of processors are combined to work simultaneously on solving complex problems.

Mbyte
See *megabyte (Mbyte).*

megabyte (Mbyte)
1048576 (2^{20}) bytes.

megaflops (MFLOPS)
One million floating-point operations per second.

memory bank conflict
An attempt to access a particular memory bank before a previous access to the bank is complete.

memory management
The hardware and software that control memory page mapping and memory protection.

message
Data copied from one process to another (or the same) process. The copy is initiated by the sending process, which specifies the receiving process. The sending and receiving processes need not share a common address space. (Note: depending on the context, a process may be a *thread*.)

Message-Passing Interface (MPI)
A message-passing and process control library. For information on the Convex implementation of MPI, refer to the *Convex MPICH User's Guide for Exemplar Systems* (DSW-493).

message passing
A type of programming in which program modules (often running on different processors or different hosts) communicate with each other by means of system library calls that package, transmit, and receive data. All message-passing library calls must be explicitly coded by the programmer.

MIMD (multiple instruction stream multiple data stream)
A computer architecture that uses multiple processors, each processing its own set of instructions simultaneously and independently of others. MIMD also describes when processes are performing different operations on different data. Compare with *SIMD*.

MPP
Massively Parallel Processor. A computer architecture in which several to thousands of processors are combined to work simultaneously on solving complex problems.

multiprocessing
The creation and scheduling of processes on any subset of CPUs in a system configuration.

mutex
A variable used to construct an area (region of code) of *mutual exclusion*. When a mutex is locked, entry to the area is prohibited; when the mutex is free, entry is allowed.

mutual exclusion
A protocol that prevents access to a given resource by more than one thread at a time.

N

near-shared memory
Memory that is addressable by the same virtual address from any hypernode in the subcomplex on which the memory was allocated. Near-shared memory resides physically on the hypernode from which it was allocated, and is accessed with lowest latency from that hypernode. Access latencies are higher from other hypernodes. Compare with *thread-private memory*, *node-private memory*, *block-shared memory*, and *far-shared memory*.

negate
An instruction that changes the sign of a number.

network
A system of interconnected computers that enables machines and their users to exchange information and share resources. SPP Series systems provide support for FDDI networks.

node
On SPP Series computers, a node is equivalent to a *hypernode*. The term "node" is generally used in place of hypernode when discussing parallelism or memory classes.

node-private memory

Memory residing on a hypernode that is accessible only by CPUs on the same hypernode in an SPP Series system. A node-private data object has a unique virtual address by which all threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode. Compare with *thread-private memory*, *near-shared memory*, *block-shared memory*, and *far-shared memory*.

non-uniform memory access (NUMA)

This term describes memory access times in systems such as the SPP Series, in which accessing different types of memory (for example, memory local to the current hypernode or memory remote to the current hypernode) results in non-uniform access times.

nonblocking crossbar

A switching device that connects the CPUs, banks of memory, and I/O controller on a single hypernode of an SPP Series system. Because the crossbar is nonblocking, all ports can run at full bandwidth simultaneously provided there is not contention for a particular port.

NUMA

Non-uniform memory access. This term describes memory access times in systems such as the SPP Series, in which accessing different types of memory (for example, memory local to the current hypernode or memory remote to the current hypernode) results in non-uniform access times.

offset

In the context of a process address space, an integer value that is added to a base address to calculate a memory address. Offsets in SPP Series systems are 32-bit values, and must keep address values within a single 4-Gbyte memory space.

opcode

A predefined sequence of bits in an instruction that specifies the operation to be performed.

operating system

The program that manages the resources of a computer system. The SPP Series system uses the SPP-UX operating system. SPP-UX is compatible with Hewlett-Packard's HP-UX operating system.

optimization
The refining of application software programs to minimize processing time. Optimization takes maximum advantage of a computer's hardware features and minimizes input/output traffic and idle processor time.

optimization level
The degree to which source code is optimized by the compiler. The Convex Fortran and C compilers have five levels of optimization: level `-no`, `-O0`, `-O1`, `-O2`, and `-O3`.

oversubscript
An array reference that falls outside declared bounds.

oversubscription
In the context of parallel threads, a process attribute that permits the creation of more threads within a process than the number of processors available to the process.

P

PA-RISC
The Hewlett-Packard Precision Architecture reduced instruction set processor chip. This is the processor chip used in SPP Series systems.

packet
A group of related items. A packet may refer to the arguments of a subroutine or to a group of bytes that is transmitted over a network.

page
A page is the unit of virtual or physical memory controlled by the memory management hardware and software. On SPP Series systems, a page is 4 K (4,096) contiguous bytes. See also *virtual memory*.

page fault
A page fault occurs when a process requests data that is not currently in main memory. This requires the operating system to retrieve the page containing the requested data from disk.

page frame
A page frame is the unit of physical (main) memory in which pages are placed. Referenced and modified bits associated with each page frame aid in memory management.

parallel optimization
The transformation of source code into parallel code (parallelization) and restructuring of code to enhance parallel performance.

Parallel Virtual Machine (PVM)
A message-passing and process control library available on SPP systems. For more information, refer to *PVM/GSM User's Guide for Exemplar Systems* (DSW-501).

parallelization
The process of transforming serial code to a form of code that can run simultaneously on multiple CPUs while preserving semantics. At optimization level `-O3`, the Convex compilers automatically parallelize loops in your program and recognize compiler directives with which you can manually specify parallelization of both loops and tasks.

parallelization, loop
The process of splitting a loop into several smaller loops, each of which operates on a subset of the data of the original loop, and generating code to run these loops on separate processors in parallel.

parallelization, ordered
The process of splitting a loop into several smaller loops, each of which iterates over a subset of the original data with a stride equal to the number of loops created, and generating code to run these loops on separate processors. Each iteration in an ordered parallel loop begins execution in the original iteration order, allowing dependences within the loop to be synchronized to yield correct results via gate constructs.

parallelization, stride-based
The process of splitting up a loop into several smaller loops, each of which iterates over several discontiguous chunks of data, and generating code to run these loops on separate processors in parallel. Stride-based parallelism can only be achieved manually by using compiler directives or CPSlib functions.

parallelization, strip-based
The process of splitting up a loop into several smaller loops, each of which iterates over a single contiguous subset of the data of the original loop, and generating code to run these loops on separate processors in parallel. Strip-based parallelism is the default for automatic parallelism and for directive-initiated loop parallelism in absence of the `chunk_size =` *n* or `ordered` attributes.

parallelization, task
The process of splitting up source code into independent sections which can safely be run in parallel on available processors. Convex programming languages provide compiler directives and pragmas which allow you to identify parallel tasks in source code.

parameter
In C, either a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called (*formal parameter*) or the variable or constant that is passed by a call to a procedure (*actual parameter*). In Fortran, a symbolic name for a constant.

path
An environment variable that you set in your shell configuration file that allows you to access commands in various directories without having to specify a complete path name.

physical address
A unique identifier that selects a particular location in the computer's memory. Because SPP-UX supports virtual memory, programs running on SPP Series computers address data by its virtual address; SPP-UX then maps this address to the appropriate physical address. See also *virtual address.*

physical address space
The set of possible addresses for a particular physical memory.

physical memory
Computer hardware that stores data. SPP Series systems can contain up to 2 Gbytes of physical memory per hypernode, for a total of 16 Gbytes of physical memory on a full 8-hypernode system.

pipeline
An overlapping operating cycle function that is used to increase the speed of computers. Pipelining provides a means by which multiple operations occur concurrently by beginning one instruction sequence before another has completed. Maximum efficiency is achieved when the pipeline is "full", i.e., when all stages are operating on separate instructions.

pipelining
Issuing instructions in an order that best utilizes the pipeline.

procedure
A unit of program code. In Fortran, a function, subroutine or main program; in C, a function.

process
A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads.

process memory
The portion of system memory that is used by an executing process.

programming model
A description of the features available to efficiently program a certain computer architecture.

program unit
A procedure or main section of a program.

PVM
Parallel virtual machine. A message-passing and process control library available on SPP Series systems.

Q

queue
A data structure in which entries are made at one end and deletions at the other. Often referred to as first-in, first-out (FIFO).

R

rank
The number of dimensions of an array.

read
A memory operation in which the contents of a memory location are copied and passed to another part of the system.

recurrence
A cycle of dependences among the operations within a loop in which an operation in one iteration depends on the result of a following operation that executes in a previous iteration.

recursion
An operation that is defined, at least in part, by a repeated application of itself.

recursive call
A condition in which the sequence of instructions in a procedure causes the procedure itself to be invoked again. Such a procedure must be compiled for *reentrancy*.

reduced instruction set computer (RISC)
An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code. The PA-RISC processor used in SPP Series computers employs a RISC architecture.

reduction
An arithmetic operation that performs a transformation on an array to produce a scalar result.

reentrancy
The ability of a program unit to have multiple versions in existence that may execute in parallel. Each version maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables. Procedures must be compiled for reentrancy in order to be invoked in parallel or to be used for recursive calls. Reentrant compilation is the default on SPP Series computers.

reference
Any operation that requires a cache line to be encached; this includes load as well as store operations, since writing to any element in a cache line requires the entire cache line to be encached.

register
A hardware entity that contains an address, operand, or instruction status information.

reuse, data
In the context of a loop, the ability to use data fetched for one loop operation in another operation. In the context of a cache, reusing data that was encached for a previous operation; since data is fetched as part of a cache line, if any of the other items in the cache line are used before the line is flushed to memory, reuse has occurred.

reuse, spatial
Reusing data that resides in the cache as a result of the fetching of another piece of data from memory. Typically, this involves using array elements that are contiguous to (and therefore part of the cache line of) an element that has already been used, and therefore is already encached.

reuse, temporal
Reusing a data item that has been used previously.

RISC

Reduced instruction set computer. An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code. The PA-RISC processor used in SPP Series computers employs a RISC architecture.

rounding

A method of obtaining a representation of a number that has less precision than the original in which the closest number representable under the lower precision system is used.

row-major order

Memory representation of an array such that the rows of an array are stored contiguously. For example, given a two-dimensional array `A[3][4]`, array element `A[0][3]` immediately precedes `A[1][0]` in memory. This is the default storage method for arrays in C. It can be selected in Convex Fortran by using the `ROWWISE` compiler directive.

S

Scalable Parallel Processor (SPP)

A computer architecture that permits a sufficiently parallel application to run with a relatively small number of processors or in a processor array containing hundreds to thousands of processors. A key design goal for SPP systems is to enable performance to increase linearly with respect to its number of processors.

SCI

Scalable Coherent Interface. This is defined by IEEE standard 1596-1992. The interface is physically defined as a pair of 18-bit, differential ECL, unidirectional links. Each link provides 16 bits of data with two control signals. Data is sampled on both the rising and falling edges of the clock. This interface provides the basis for the CTI rings used in SPP Series systems; however, total compatibility with the standard has been sacrificed to provide increased performance.

scope

The domain in which a variable is visible in source code. The rules that determine scope are different for Fortran and C.

semaphore

An integer variable assigned one of two values: one value to indicate that it is "locked," and another to indicate that it is "free." Semaphores can be used to synchronize parallel threads. CPSlib provides a set of manipulation functions to facilitate this.

shape
The number of elements in each dimension of an array.

shared virtual memory
Globally shared memory. A memory architecture in which memory can be accessed by all processors in the system. This architecture can also support virtual memory. On SPP Series computers, globally shared memory is distributed among the hypernodes, but any hypernode's memory is accessible from any processor on any hypernode. This type of memory is sometimes referred to as globally shared memory or global virtual memory.

shell
An interactive command interpreter that is the interface between the user and the operating system.

SIMD (single instruction stream multiple data stream)
A computer architecture that performs one operation on multiple sets of data. A processor (separate from the SMP array) is used for the control logic, and the processors in the SMP array perform the instruction on the data. Compare with *MIMD (multiple instruction stream multiple data stream).*

single
A single-precision floating-point number stored in 32 bits. See also *double.*

SMP
Symmetric multiprocessor. A multiprocessor computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processors; the operating system runs on any or all of the processors.

socket
An endpoint used for interprocess communication.

socket pair
Bidirectional pipes that enable application programs to set up two-way communication between processes that share a common ancestor.

source code
The uncompiled version of a program, written in a high-level language such as C or Fortran.

source file
A file that contains program source code.

space
A contiguous range of virtual addresses within the system-wide virtual address space. Spaces are 4 Gbytes in size in SPP Series systems.

spatial reference
An attribute of a memory reference pattern that pertains to the likelihood of a subsequent memory reference address being numerically close to a previously referenced address.

spawn
To activate existing threads.

spawn context
A parallel loop, region, or task list that initiates the spawning of threads and defines the structure within which the threads' spawn thread IDs are valid.

spawn thread identifier (stid)
A sequential integer identifier associated with a particular thread that has been spawned. stids are only assigned to spawned threads, and they are assigned within a spawn context; therefore, duplicate stids may be present amongst the threads of a program, but stids are always unique within the scope of their spawn context. stids are assigned sequentially and run from 0 to one less than the number of threads spawned in a particular spawn context.

SPMD
Single program multiple data. A single program executing simultaneously on several processors. This is usually taken to mean that there is redundant execution of sequential scalar code on all processors.

SPP
Scalable parallel processor. A computer architecture that permits a sufficiently parallel application to run with a relatively small number of processors or in a processor array containing hundreds to thousands of processors. A key design goal for SPP systems is to enable performance to increase linearly with respect to its number of processors.

stack
A data structure in which the last item entered is the first to be removed. Also referred to as last-in, first-out (LIFO). SPP-UX provides every thread with a stack which is used to pass arguments to functions and subroutines and for local variable storage.

store
An instruction used to move the contents of a register to memory.

strip length, parallel
In strip-based parallelism, the amount by which the induction variable of a parallel inner loop is advanced on each iteration of the (conceptual) controlling outer loop.

strip mining
The transformation of a single loop into two nested loops. Conceptually, this is how parallel loops are created by default. A conceptual outer loop advances the initial value of the inner loop's induction variable by the parallel strip length. The parallel strip length is based on the trip count of the loop and the amount of code in the loop body. Strip mining is also used by the data localization optimization.

subcomplex
In SPP Series systems, a logical entity that provides control over the allocation of processors and physical memory to different applications and users.

subroutine
A software module that can be invoked from anywhere in a program.

superscalar
A class of *RISC* processors that allow multiple instructions to be issued on each clock period.

Symmetric Multiprocessor (SMP)
A multiprocessor computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processors; the operating system runs on any or all of the processors.

synchronization
A method of coordinating the actions of multiple threads so that operations occur in the right sequence. When manually optimizing code, you can synchronize programs using compiler directives, calls to library routines, or assembly-language instructions. You do so, however, at the cost of additional overhead; synchronization may cause at least one CPU to wait for another.

system administrator (sysadmin)
The system manager.

system manager
The person responsible for the management and operation of a computer system. Also called the system administrator and the sysadmin.

system subcomplex
In an SPP Series system, a subcomplex that is automatically created at boot time by the operating system to run system processes, including `init` and processes spawned by `init`. The Subcomplex Manager will not allow users to destroy this subcomplex, nor remove the last processor from this subcomplex.

T

term
A constant or symbolic name that is part of an *expression*.

thread
An independent execution stream that is executed by a CPU. One or more threads, each of which can execute on a different CPU, make up each process. Memory, files, signals, and other process attributes are generally shared among threads in a given process, enabling the threads to cooperate in solving the common problem. Threads are created and terminated by instructions that can be automatically generated by Convex compilers, inserted by adding compiler directives to source code, or coded explicitly using library calls or assembly-language.

thread create
To activate existing threads.

thread identifier
An integer identifier associated with a particular thread. See *thread identifier, kernel (ktid)* and *thread identifier, spawn (stid)*.

thread identifier, kernel (ktid)
A unique integer identifier (not necessarily sequential) assigned when a thread is created.

thread identifier, spawn (stid)
A sequential integer identifier associated with a particular thread that has been spawned. stids are only assigned to spawned threads, and they are assigned within a spawn context; therefore, duplicate stids may be present amongst the threads of a program, but stids are always unique within the scope of their spawn context. stids are assigned sequentially and run from 0 to one less than the number of threads spawned in a particular spawn context.

thread-private memory

Data that is accessible by a single thread only (not shared among the threads constituting a process). A thread-private data object has a unique virtual address which maps to a unique physical address within each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses. Compare with *node-private memory, near-shared memory, far-shared memory,* and *block-shared memory.*

tree-height reduction

Expressions are represented internally as trees whose height corresponds to the depth of the expression. Tree-height reduction or balancing is an optimization that attempts to simplify expressions by shortening their tree height.

trip count

The number of iterations a loop executes.

U

unsigned

A value that is always positive.

user interface

The portion of a computer program that processes input entered by a human and provides output for human users.

utility

A software tool designed to perform a frequently used support function.

V

vector

An ordered list of items in a computer's memory, contained within an array. A simple vector is defined as having a starting address, a length, and a stride. An indirect address vector is defined as having a relative base address and a vector of values to be applied as offsets to the base.

vector processor

A processor whose instruction set includes instructions that perform operations on a *vector* of data (such as a row or column of an array) in an optimized fashion. Convex C Series systems employ vector processors; SPP Series systems do not.

virtual address

The address by which programs access their data. SPP-UX maps this address to the appropriate physical memory address. See also *space.*

virtual aliases
Two different virtual addresses that map to the same physical memory address.

virtual machine
A collection of computing resources configured so that a user or process can access any of the resources, regardless of their physical location or operating system, from a single interface.

virtual memory
The memory space as seen by the program, which is typically larger than the available physical memory. The virtual memory of an SPP Series computer can be up to 4 Gbytes (however, through use of node-private memory, this 4 Gbytes can be mapped to a *larger* set of physical memory). SPP-UX maps this virtual memory to a smaller set of physical memory, using disk space to make up the difference if necessary. Also called *logical memory.*

W

wall clock time
The chronological time an application requires to complete its processing. If an application starts running at 1:00 p.m. and finishes at 5:00 a.m. the following morning, its wall clock time is sixteen hours. Compare with *CPU time.*

word
A contiguous group of bytes that make up a primitive data operand and start on an addressable boundary. In SPP1000, SPP1200, and SPP1600 Series computer systems, a word is four bytes (32 bits) in length. See also *doubleword.*

workstation
A stand-alone computer that has its own processor, memory, and possibly a disk drive and can typically sit on a user's desk.

write
A memory operation in which a memory location is updated with new data.

Z

zero
In floating point number representations, zero is represented by the sign bit with a value of zero and the exponent with a value of zero.

Index

Symbols

* (asterisk) entry
  in optimization report 349

A

ABI
  defined 409
aborts
  program 280, 301
absolute address
  defined 409
accumulator
  defined 409
accumulator variables
  and floating-point imprecision 278
actual argument
  defined 409
actual parameter
  defined 409
address
  defined 409
address space
  defined 410
affinity
  threads to CPUs 110
agent
  defined 410
algebraic simplification 47
alias 99
  defined 410
-alias addr option 343
-alias array_args option 341
-alias cautious option 259, 342
-alias global option 342
-alias no_addr option 343
-alias no_global option 342
-alias ptr_args option 343
-alias restrict_args option 344
-alias standard 259, 342
-alias worst option 259, 342
aliases
  hidden 256
  potential 341
aliasing 256
  and ANSI C 258
  and C compatibility modes 257
  and C pointers 257
  ANSI C 257
  ANSI C algorithm 258
  ANSI C, sometimes unsafe 258
  command-line options 259
  Fortran example 256
  global variables 257, 263
  in C 257
  local variables 257
  stop variable 262
  worst-case 257
-align cache option 267, 337, 338
-align cache_check option 337, 339
-align cseries option 337
-align cti option 22, 337
-align options
  combining 340
-align spp option 337
ALIGN_CTI directive and pragma 22, 314
aligning data 21, 267
  for cseries 337
  on CTIcache boundaries 21, 337
alignment
  and -align cti option 267
  and ALIGN_CTI directive and pragma 267, 314
  data 21, 42, 267
  defined 410
  of arrays 267
  of COMMON blocks 337
  on natural boundaries 42, 337
ALL
  defined 410
alloc_barrier function 215
alloc_barrier_8 function 215
alloc_gate function 215
alloc_gate_8 function 215
allocatable array
  defined 410
allocate
  defined 410
allocation
  of barriers 215
  of gates 215
  of registers 39
ALU
  defined 410
Amdahl's law
  defined 411

American National Standards Institute (ANSI)
defined 411
analysis column
in analysis table 352
in test table 353
analysis table
in optimization report 352
ANSI
defined 411
ANSI C aliasing 258
ANSI C aliasing algorithm 258
ANSI Fortran
non-compliant argument passing 60
apparent dependence 282, 298
apparent LCDs 120, 298, 299
apparent recurrence
defined 411
application binary interface
defined 411
architecture
memory 14
organization 11
overview 11
argument
actual 409
defined 411
arithmetic logic unit (ALU)
defined 411
array
defined 411
array section
defined 411
array table
dependences 354
line number column 354
optimization column 354
variable name column 354
arrays
aligning 267, 338
dimensions and thrashing 26
scalar replacement of elements 93
storage order 327
array-valued argument
defined 412
ASCII
defined 412
assembler
defined 412
assembly language
defined 412
ASSIGN statement 50
assigned GOTO statements 94
assignment substitution 44
assignments
elimination of redundant 43, 49
assist cache 17
asterisk (*) entry
in optimization report 349
asymmetric parallelism 367
CPSlib example 397
asymmetric threads
compiler parallel support library functions 373
attributes
for directives and pragmas 126, 315
automatic array
defined 412
automatic parallelization 105
disabling 148, 332

B

backward LCDs 119
balancing
defined 412
trees 39
bandwidth
defined 412
bank conflict
defined 412
barrier
defined 412
barrier synchronization
defined 413
barrier_t data type 214
barrier8_t data type 214
barriers 213, 314
allocating 215
allocating with cps_barrier_alloc function 383
and the compiler parallel support library 382
CPSlib example 399
deallocation 216
freeing with cps_barrier_free function 383
in C 213
in Fortran 214
incrementing with cps_barrier function 383
wait_barrier function 217
basic block 45
defined 413
optimizations 48
BEGIN_TASKS directive and pragma 137, 315
bit
defined 413
block parallelism 127, 393
CPSlib example 393
BLOCK_LOOP directive and pragma 76, 316
BLOCK_SHARED directive 316
block_shared memory 172
defined 413
dynamic allocation 199
static assignments 182
using 247
block_shared memory class 316

blocking
BLOCK_LOOP directive and pragma 76, 316
explained 70
NO_BLOCK_LOOP directive and pragma 76, 321
blocking factor
defined 413
specifying 316
-blockloop compiler option 77, 334
branch
defined 413
byte
defined 413

C

c_cond_lock function 387
c_fetch_and_add32 function 388
c_fetch_and_clear32 function 388
c_fetch_and_dec32 function 387
c_fetch_and_inc32 function 387
c_fetch_and_set32 function 388
c_fetch32 function 387
c_free32 function 386
c_init32 function 386
c_lock function 386
c_unlock function 386
cache
assist 17
based semaphores 386
data 13
defined 414
instruction 13
interconnect 14
preventing thrashing 24
thrashing 23
thrashing and COMMON blocks 26
thrashing example 23, 264
cache addresses 17
cache coherency 3
cache hit
defined 414
cache line
defined 414
cache lines 16
CTI 16
false sharing 264
processor 16
cache memory
defined 415
cache miss
defined 415
-cache n option 332
cache purge
defined 415
cache size
specifying 332
cache thrashing 23
defined 415
illustrated 24
caches 13
caution
on NO_SIDE_EFFECTS 52
CCMC 14
defined 415
central processing unit (CPU)
defined 415
chunk_size
attribute to loop_parallel 127
chunk-based parallelism 127, 129
example 130
clock cycle
defined 415
clone
defined 415
clones
and reentrant compilation 148
loop 115
cloning
loop 115
clustered workstations
compilers 4
configurability 6
interprocess communication 5
memory 4
peripherals 6
vs. SPP 4
code
defined 415
code motion 55, 309
and wrong answers 256
coherency
defined 416
in caches 3
coherent memory controller 14
Coherent Toroidal Interconnect 2
Col. Num. column
in test table 353
column-major order
defined 416
example 64
combining -align options 340
COMMON blocks
and cache thrashing 26
padding for C Series 337
common subexpressions
elimination of 44, 53
compile time
and -mrl option 333
compiler
defined 416

compiler directives 313
- ALIGN_CTI 22, 314
- BARRIER 314
- BEGIN_TASKS 137, 315
- BLOCK_LOOP 316
- BLOCK_SHARED 316
- CRITICAL_SECTION 316
- DYNSEL 116, 317
- END_CRITICAL_SECTION 317
- END_ORDERED_SECTION 317
- END_PARALLEL 317
- END_TASKS 137, 318
- FAR_SHARED 318
- FAR_SHARED_POINTER 318
- GATE 318
- LOOP_PARALLEL 319
- LOOP_PRIVATE 320
- misused 255, 281
- NEAR_SHARED 320
- NEAR_SHARED_POINTER 321
- NEXT_TASK 137, 321
- NO_BLOCK_LOOP 321
- NO_DISTRIBUTE 321
- NO_DYNSEL 117, 322
- NO_FUSE 80, 322
- NO_LOOP_DEPENDENCE 282, 298, 322
- NO_PARALLEL 322
- NO_PEEL 90, 322
- NO_PROMOTE_TEST 92, 323
- NO_SIDE_EFFECTS 51, 323
- NO_UNROLL 323
- NO_UNROLL_AND_JAM 323
- NODE_PRIVATE 323
- NODE_PRIVATE_POINTER 324
- OPT_LEVEL 324
- ORDERED_SECTION 324
- PARALLEL 325
- PARALLEL_PRIVATE 325
- PEEL 90, 325
- PEEL_ALL 325
- PREFER_FUSE 80, 326
- PREFER_PARALLEL 326
- PROMOTE_TEST 92, 327
- PROMOTE_TEST_ALL 92, 327
- ROW_WISE 327
- SAVE_LAST 327
- SCALAR 327
- SYNC_ROUTINE 218, 328
- TASK_PRIVATE 328
- tasking 137
- THREAD_PRIVATE 328
- THREAD_PRIVATE_POINTER 328
- UNROLL 329
- UNROLL_AND_JAM 329

compiler optimizations 37
- options 37
- overview 8

compiler options
- -alias addr 343
- -alias array_args 341
- -alias cautious 259, 342
- -alias global 342
- -alias no_addr 343
- -alias no_global 342
- -alias ptr_args 343
- -alias restrict_args 344
- -alias standard 259, 342
- -alias worst 259, 342
- -align cache 267, 337, 338
- -align cache_check 337, 339
- -align cseries 337
- -align cti 22, 337
- -align spp 337
- -blockloop 77, 334
- C aliasing 259
- -cache n 332
- cross compilation 332
- -ds 334
- -fl 80
- -il 344
- -is 344
- loop replication 333
- +max 393
- +min 393
- misc. optimization 344
- misused 255
- -mo 345
- -mrl 63, 333
- -nds 116, 334
- -nfl 80
- -nga 62, 336
- -ngs 62, 336
- -nmo 344
- -no 37, 38, 43, 331
- -noautopar 38, 122, 148, 332
- -noblock 77, 334
- -nonodepar 38, 114, 148, 332
- -nopeel 90
- -noptst 92
- -nore 149, 345
- -nosc 41
- -nsr 94, 345
- -nuj 85, 333
- -nur 82, 333
- -O0 37, 43, 331
- -O1 37, 48, 331
- -O2 37, 63, 88, 331
- -O3 37, 105, 331
- -or all 332
- -or array 332
- -or loop 332
- -or none 332
- -or private 332
- -or table 331, 347

+parallel 393
-peel 90, 335
-peelall 335
-noptst 335
-ptst 92, 335
-ptstall 92, 335
-re 345
-sr 94, 345
-tm 332
-uj 85, 333
-ujn 85, 333
-uo 57, 309, 345
-ur 82, 333
-urn 82, 333
-Wl, 393
compiler parallel support library 365
accessing 369
and asymmetric parallelism 367
and symmetric parallelism 365
and sync_routine 391
asymmetric parallelism example 397
asymmetric thread functions 373
barriers 382, 399
block parallelism example 393
c_cond_lock function 387
c_fetch_and_add32 function 388
c_fetch_and_clear32 function 388
c_fetch_and_dec32 function 387
c_fetch_and_inc32 function 387
c_fetch_and_set32 function 388
c_fetch32 function 387
c_free32 function 386
c_init32 function 386
c_lock function 386
c_unlock function 386
cps_barrier function 383
cps_barrier_alloc function 383
cps_barrier_free function 383
cps_complex_cpus function 379
cps_complex_nodes function 380
cps_complex_nthreads function 379
cps_is_parallel function 379
cps_ktid 376
cps_mutex_alloc function 384
cps_mutex_free function 384
cps_mutex_lock function 384
cps_mutex_trylock function 385
cps_mutex_unlock function 385
cps_node_cpus function 378
cps_node_id function 378
cps_node_nthreads function 379
cps_nsthreads function 376
cps_plevel function 377
cps_ppcall function 370
cps_ppcalln function 370
cps_stid function 376
cps_thread_create function 373
cps_thread_exit function 374
cps_thread_wait function 375
cps_topology function 380
cps_wait_attr function 381
critical sections using low-level functions 403
cyclic parallelism example 396
defined 416
finding hypernode ID 378
finding kernel thread ID 376
finding number of cpus 378
finding number of threads 376
finding spawn thread ID 376
high-level synchronization functions 382, 399
low-level ordered section example 404
low-level semaphores and critical sections 403
low-level semaphores and ordered sections 404
low-level synchronization functions 385, 403
m_cond_lock function 390
m_fetch_and_clear32 function 391
m_fetch_and_dec32 function 390
m_fetch_and_inc32 function 390
m_fetch32 function 390
m_free32 function 389
m_init32 function 389
m_lock function 389
m_unlock function 389
mutexes 382, 401
PARAMS values 370
params->max values 371
params->min values 371
params->node values 370
params->threadscope values 371
setting stack size for spawned threads 372
spawning symmetric threads 370
specifying parallelism at compile time 393
symmetric parallelism examples 393
thread information functions 376
thread-management functions 370
compiler pragmas 313
complex
defined 416
computed statements 94
concurrent
defined 416
concurrent execution 38
cond_lock_gate function 216
cond_lock_gate_8 function 216
conditional induction variable
defined 416
conditionals
short-circuit evaluation 41
constant folding 45
defined 416
constant propagation 45, 48
defined 417
constants
strength reduction of 57

control parallel programming
defined 417
conventional compiler
defined 417
copy propagation 52
counted loop 261
counter
defined 417
CPA
defined 417
cps.h 369
CPS_ANY_NODE constant 371
cps_barrier function 383
cps_barrier_alloc function 383
cps_barrier_free function 383
cps_complex_cpus function 379
cps_complex_nodes function 380
cps_complex_nthreads function 379
CPS_DIFFERENT_NODE constant 371
CPS_GETWAIT constant 381
cps_is_parallel function 379
cps_ktid function 376
cps_mutex_alloc function 384
cps_mutex_free function 384
cps_mutex_lock function 384
cps_mutex_trylock function 385
cps_mutex_unlock function 385
cps_node_cpus function 378
cps_node_id function 378
cps_node_nthreads function 379
CPS_NODE_PARALLEL constant 371
cps_nsthreads function 376
CPS_PL_ASYMMETRIC constant 377
CPS_PL_NODE constant 377
CPS_PL_NONE constant 377
CPS_PL_NTHREAD constant 377
CPS_PL_PARALLEL constant 377
CPS_PL_THREAD constant 377
cps_plevel function 377
cps_ppcall function 370
setting stack size for spawned threads 372
cps_ppcalln function 370
CPS_SAME_NODE constant 371
CPS_SETWAIT constant 381
CPS_SETWAITI constant 381
CPS_SPINWAIT constant 382
CPS_STACK_SIZE environment variable
and CPSlib parallelism 372
and loop_parallel 149
and loop_private data 150
and task_private data 150
cps_stid function 376
CPS_SUSPEND constant 382
cps_thread_create function 373
cps_thread_exit function 374
CPS_THREAD_PARALLEL constant 371
cps_thread_wait function 375
cps_topology function 380
cps_wait_attr function 381
CPSlib
defined 417
See compiler parallel support library
CPU
defined 417
CPU agent
defined 417
CPU time
defined 418
CPU-private memory
defined 418
CPUs
and thread affinity 110
minimum/maximum number 1
critical section
defined 418
critical sections 146, 316
and gates 223
and the compiler parallel support library 382
low-level CPSlib example 403
manually implemented 238
multiple 225
using CPSlib mutexes 401
CRITICAL_SECTION directive and pragma 146, 316
crossbar 11
defined 418
illustrated 12
CSR
defined 418
CTI 2
CTI interface
defined 418
CTI ring
defined 418
CTI rings 2
illustrated 12
CTIcache 167
defined 418
illustrated 13
CTIcache lines 16
interleaving 29
customer support xxvi
cyclic parallelism
and CPSlib ordered sections 406
example 396

D

data alignment
  on CTIcache lines 21, 267, 337
  on natural boundaries 42
  on processor cache lines 267, 337, 338
data cache 13
  defined 419
data dependence
  defined 419
data localization 63
  and loop replication 63
  benefits 64
  data reuse 70
  defined 419
  inhibitors 94
  preventing 104, 327
  spatial reuse 70
  strip mining 66
data ordering 42
data parallel programming
  defined 419
data privatization
  and prefer_parallel 134
  in parallel loops 320
  in parallel regions 160
  in parallel tasks 158
  in tasks 328
  loop 150
data reuse 70
  defined 437
  example 71
  spatial 70
  temporal 70
data type
  defined 419
dead code
  eliminating 52
deadlock
  defined 419
dependence 281
dependences
  apparent 282
  C example 282
  Fortran example 281
  hidden 288
  ignoring 322
  isolating with ordered sections 229
  loop-carried 94, 281
  ordering 324
  reductions 121, 298
  unordered 146, 316
Dependences column
  in array table 354
determined order of execution 303
direct mapped cache
  defined 414
direct memory access
  defined 419
directives
  ALIGN_CTI 22, 314
  attributes 126, 315
  BARRIER 214, 314
  BEGIN_TASKS 315
  BLOCK_LOOP 316
  BLOCK_SHARED 316
  CRITICAL_SECTION 146, 223, 316
  DYNSEL 116
  END_CRITICAL_SECTION 146, 223, 317
  END_ORDERED_SECTION 224, 317
  END_PARALLEL 143, 317
  END_TASKS 318
  FAR_SHARED 318
  FAR_SHARED_POINTER 318
  form 313
  Fortran compiler 313
  GATE 214, 318
  loop blocking 76
  LOOP_PARALLEL 126, 319
  LOOP_PARALLEL(ORDERED) example 221
  LOOP_PRIVATE 151, 320
  memory class 173
  misused 281
  NEAR_SHARED 320
  NEAR_SHARED_POINTER 321
  NEXT_TASK 321
  NO_BLOCK_LOOP 76, 321
  NO_DISTRIBUTE 68, 321
  NO_DYNSEL 117, 322
  NO_FUSE 80
  NO_LOOP_DEPENDENCE 97, 120, 322
  NO_PARALLEL 122, 322
  NO_PEEL 322
  NO_PROMOTE_TEST 92, 323
  NO_SIDE_EFFECTS 323
  NO_UNROLL 323
  NO_UNROLL_AND_JAM 86, 323
  NODE_PRIVATE 323
  NODE_PRIVATE_POINTER 324
  ORDERED_SECTION 224, 324
  PARALLEL 143, 293, 325
  PARALLEL_PRIVATE 161, 325
  PEEL 90, 325
  PEEL_ALL 90, 325
  PREFER_FUSE 80, 326
  PREFER_PARALLEL 126, 326
  PROMOTE_TEST 92, 327
  PROMOTE_TEST_ALL 92, 327
  ROW_WISE 327
  SAVE_LAST 163, 327
  SCALAR 104, 327
  SYNC_ROUTINE 218, 328

TASK_PRIVATE 158, 328
THREAD_PRIVATE 328
THREAD_PRIVATE_POINTER 328
UNROLL 82, 329
UNROLL_AND_JAM 86
dist
attribute to LOOP_PARALLEL 132
Dist entry
in optimization report 349
distributed memory
defined 420
distributed part
defined 420
distribution
loop 67
DoD 411
double
defined 420
doubleword
defined 420
-ds option 334
dummy argument
defined 420
dynamic memory
and memory class pointers 183
and memory_class_malloc 184
assigning block_shared class 199
assigning classes 187
assigning far_shared class 197
assigning near_shared class 191
assigning node_private class 188
assigning thread_private class 187
class assignments 182
default classes 186
dynamic memory class assignments 182
and wrong answers 283
incorrect pointer use 286
dynamic selection 115
and dynsel directive and pragma 317
and reentrancy 149
defined 420
in optimization report 357
workload-based 115
DYNSEL directive and pragma 116, 317
DynSel entry
in optimization report 349

E

elimination
of common subexpressions 44, 53
of dead code 52
of redundant uses 45
of type conversions 310
encache
defined 420
END_CRITICAL_SECTION directive and pragma 146, 317
END_ORDERED_SECTION directive and pragma 317
END_PARALLEL directive and pragma 143, 317
END_TASKS directive and pragma 137, 318
entries
multiple routine 94, 118
environment variables
CPS_STACK_SIZE 149, 372
error message
overflow 46
evaluation order 300
exception
defined 420
executables
modifying attributes via mpa 379
execution stream
defined 420
Exemplar system overview 1
exits
multiple routine 94, 118
expressions
equivalent 309
-ext option
and aliasing 257

F

false cache line sharing 294
FAR_SHARED directive 318
far_shared memory 172
defined 421
static assignments 181
far_shared memory class 318
FAR_SHARED_POINTER directive 186, 247, 285, 318
fault
defined 421
Federal Information Processing (FIPS) 411
.fil file 344
creating 344
file utility 393
-fl option 80
floating point
defined 421
floating-point imprecision 57, 255, 279
example 278
FLOPS
defined 421
FMPYADD instruction 85
folding
constant 48
of constants 45
Footnoted Iter. Var. column
in variable name footnote table 354

footnotes
in optimization report 354
in optimization report, example 357
for loops
specifying induction variables for parallelization 128
formal parameter
defined 421
Fortran
defined 421
Fortran 90
defined 422
Fortran 90 array expressions
and parallelization 114
forward LCDs 118
free_barrier function 216
free_barrier_8 function 216
free_gate function 216
free_gate_8 function 216
fully associative cache
defined 414
function
defined 422
functional block 13
illustrated 13
functional unit
defined 422
functional units 38
functions
parallelization of intrinsic 121
fusion
of loops 78

G

gate
defined 422
gate_t data type 214
gate8_t data type 214
gates 213, 318
allocating 215
deallocation 216
in C 213
in Fortran 214
locking 216
unlocking 217
Gbyte
defined 422
gigabyte
defined 422
global
variable aliasing 263
global memory
defined 422
global optimization
defined 422
global register allocation 59
and compile time 63
and scalar replacement 93
global variable
defined 422
global variable aliasing 257
global virtual memory
defined 423
globally shared memory 3
defined 423
GRA 59
granularity
defined 423
GSM (globally shared memory) 3
defined 423

H

hand-rolled loop
defined 423
hand-rolled loops
manually parallelizing 153
hidden alias
defined 423
hidden aliases 256
hidden dependences 288
Fortran example 288
hidden ordered sections 304
Fortran example 305
High Performance Fortran (HPF)
defined 423
high-level synchronization functions 382
hoist
defined 423
host
defined 424
HP
defined 424
HP-UX
defined 424
hypercube
defined 424
hypernode 1
defined 424
finding logical ID using compiler parallel support library 378
finding logical ID using my_node() function 209
illustrated 12
hypernode-local memory 14
hypernode-parallelism
and LOOP_PARALLEL 111
and parallel regions 111
and parallel tasks 111
and PREFER_PARALLEL 111
disabling 38, 114, 148
vs. thread-parallelism 111

hypernodes
  finding available using compiler parallel support library 380
  finding number of active threads using compiler parallel support library 379
  finding thread topology 380
  minimum/maximum number 1

I

Id Num.
  in analysis table 352
  in loop table 348
Id Num. column
  in privatization table 353
idle threads
  setting wait attributes 381
  specifying states 381
  spin-waiting 110
  states of 110
  suspended 110
IF tests
  short circuiting 41
IF-DO optimizations 87
if-for optimizations 87
  options 335
-il option 344
incorrect answers
  and array pointers 286
  and evaluation order 300
  and floating point imprecision 277
  and hidden dependences 288
  and incrementing by zero 301
  and large trip counts 304
  and misused directives 281
  and misused memory classes 283
  and no_loop_dependence 299
  and ordered sections 304
  and parallel execution 303
  and parallelism 305
incrementing by zero 301
  examples 302
induction variables 57, 261
  and parallelization directives 128
  conversion 311
  in parallel hand-rolled loops 153
  indicating to compiler 153
  primary 153
  privatizing secondary 154
  secondary 154
inhibitors of localization 94
  aliasing 99
  GOTO statements 103
  I/O statements 103
  loop-carried dependences 94
  multiple entries/exits 102
  procedure calls 103
  RETURN/STOP statements 103
inhibitors of parallelization 118
  loop-carried dependences 118
inline substitution 344
inlining
  defined 425
  using -il 344
Institute for Electrical and Electronic Engineers(IEEE)
  defined 425
instruction
  defined 425
instruction cache 13
  defined 425
instruction mnemonic
  defined 425
instructions
  scheduling 38, 43
  span-dependent 38
integral division
  defined 425
interchange
  loop 69
Interchange entry
  in optimization report 349
interconnect cache 14
interface
  defined 425
interleaved memory
  defined 425
interleaving 29
  example 30
interprocedural optimization
  defined 426
interprocessor communication
  defined 426
intrinsic
  defined 426
intrinsic functions
  parallelization of 121
invalid subscripts 280
invariant expressions 310
-is option 344
Iter. Var.
  in analysis table 352
  in loop table 348
Iter. Var. column
  in privatization table 353
iteration variables 301

J

job scheduler
  defined 426
join
  defined 426
joins
  and cps_ppcall 371
  and CPSlib asymmetric threads 367
  and CPSlib symmetric parallelism 367
  as implicit barrier 399
jump
  defined 426

K

kbyte
  defined 426
kernel
  defined 426
kernel thread ID 212
  finding using compiler parallel support library 376
kernel thread identifier
  defined 442
kilobyte
  defined 426
ktid
  defined 426

L

large application
  defined 427
large trip counts 304
latency
  defined 427
level of parallelism
  finding 210
level_of_parallelism() function 210
libpvm
  defined 427
limits of optimization 255
Line Num. column
  in analysis table 352
  in array table 354
  in loop table 348
  in privatization table 353
  in test table 353
linker
  defined 427
load
  defined 427
load balancing
  and logical hypernode ID 209
local optimization
  defined 427
local variable aliasing 257
locality of reference
  defined 427
localization
  of data 63
  preventing 327
lock_gate function 216
lock_gate_8 function 216
logical address
  defined 427
logical hypernode ID
  finding using CPSlib 378
  finding using my_node() function 209
loop
  counter pointer 261
loop blocking 70
  BLOCK_LOOP directive and pragma 316
  compiler options 334
  data reuse 70
  defined 428
  example 72
  illustration 73
  matrix multiply example 75
  NO_BLOCK_LOOP directive and pragma 321
  related directives and pragmas 76
  spatial reuse 70
  temporal reuse 70
loop-carried dependence
  illustrated 96
  defined 428
loop-carried dependences 94, 281
  and parallelization 118
  apparent 120
  backward 119
  forward 118
  NO_LOOP_DEPENDENCE
    directive and pragma 322
  ordering 324
  output 120
  unordered 316
loop cloning 115
loop constant
  defined 428
loop distribution 67
  defined 428
  in optimization report 357
  NO_DISTRIBUTE directive and pragma 321
  parts in optimization report 357
loop fusion 78, 83
  and Fortran 90 array assignments 79
  and NO_FUSE directive and pragma 322
  and other transformations 79

Index

loop ID number
in optimization report 350
loop induction variable
defined 429
loop interchange 69
defined 429
in optimization report 358
loop invariant computation
defined 429
loop jamming 83
loop parallelization
defined 434
loop peeling 89
and loop replication 90
compiler options 335
disabling 335
enabling 335
in optimization report 361
PEEL directive and pragma 325
PEEL_ALL directive and pragma 325
preventing 322
loop private data
SAVE_LAST directive and pragma 163, 327
loop replication 335
defined 429
loop replication compiler options 333
loop unroll and jam 83
compiler options 85
matrix multiply example 83
options 333
loop unrolling 80
and jamming 83
compiler options 333
depth 81
factor 81
partial 81
total 81
UNROLL directive and pragma 329
using -ur 333
LOOP_PARALLEL directive and pragma 126, 135, 319
LOOP_PARALLEL(chunk_size) directive and pragma 127
LOOP_PARALLEL(dist) directive and pragma 128, 132
LOOP_PARALLEL(ivar) directive and pragma 128
LOOP_PARALLEL(max_threads) directive and pragma 127
LOOP_PARALLEL(nodes) directive and pragma 127
LOOP_PARALLEL(ordered) directive and pragma 127, 221
LOOP_PARALLEL(threads) directive and pragma 127
LOOP_PRIVATE directive and pragma 150, 320
and CPS_STACK_SIZE 150
Fortran example 151
loop-independent dependence (LID)
defined 428
loops with calls
parallelizing 136
low-level synchronization functions 385

M

m_cond_lock function 390
m_fetch_and_clear32 function 391
m_fetch_and_dec32 function 390
m_fetch_and_inc32 function 390
m_fetch32 function 390
m_free32 function 389
m_init32 function 389
m_lock function 389
m_unlock function 389
machine exception
defined 429
main memory
defined 429
main procedure
defined 429
main program
defined 430
manual synchronization 236
massively parallel processing (MPP) 1
massively parallel processor
defined 430
matrix multiplication 69
matrix multiply
blocking example 75
+max option 393
max_threads
attribute to loop_parallel 127
Mbyte
defined 430
megabyte
defined 430
megaflops
defined 430
memory
physical 14
private vs. shared 168
subcomplex 36
virtual 15
memory class pointers 183
C 184
Fortran 183
private pointers with shared data 185

memory classes 167
acceptable pointer/data class combinations 185
and spp_prog_model.h 174, 206
and suitable pointer classes 186
assigning in C 173
assigning in Fortran 173
assignments 173
block_shared 172, 316
default for dynamic allocation 186
dynamic assignments 182
dynamically assigning block_shared 199
dynamically assigning far_shared 197
dynamically assigning near_shared 191
dynamically assigning node_private 188
dynamically assigning thread_private 187
far_shared 172
FAR_SHARED directive 318
FAR_SHARED_POINTER directive 318
incorrect use examples 283
near_shared 171
near_shared_pointer 320, 321
node_private 171, 323
node_private_pointer 324
physical addressing illustrated 169
pointers 183
static assignments 174
static far_shared assignments 181
static near_shared assignments 180
static node_private assignments 177
static thread_private assignments 175
thread_private 170, 328
thread_private_pointer 328
virtual addressing illustrated 168
virtual to physical mapping 168
memory configurations 2
memory management
defined 430
memory type of stack
finding 211
memory_class_malloc function 184
memory_type_of_stack() function 211
message
defined 430
overflow error 46
message passing 7, 251
defined 430
parallelism of programs 251
message passing/shared memory hybrids 8
MIMD
defined 431
+min option 393
miscellaneous optimization options 344
misused directives 255
-mo option 345
moving code 55, 309
mpa utility 379, 393
MPI message passing 252
MPP (massively parallel processing) 1
-mrl option 63, 333
and dynamic selection 117
and global register allocation 63
and loop blocking 77
and loop peeling 90
and test promotion 92
and unroll and jam 86
and unrolling 83
multi-op instructions 344
multiple routine entries 94, 118
multiprocessing
defined 431
mutex
defined 431
mutexes
allocating with cps_mutex_alloc function 384
conditionally acquiring with cps_mutex_trylock function 385
CPSlib example 401
defined 382
freeing with cps_mutex_free function 384
locking with cps_mutex_lock function 384
unlocking with cps_mutex_unlock function 385
mutual exclusion
defined 431
mutual exclusion areas
CPSlib example 401
my_node() function 209
my_thread() function 208

N

natural data type boundaries
alignment on 42, 337
-nds option 116, 334
NEAR_SHARED directive 320
near_shared memory 171
defined 431
dynamic allocation 191
static assignments 180
NEAR_SHARED_POINTER directive 321
negate
defined 431
nested parallelism 111
network
defined 431
new loops
in loop table 350
NEXT_TASK directive and pragma 137, 321
-nfl option 80
-nga compiler option 62, 336
-ngs compiler option 62, 336
-nmo option 345
-no option 37, 38, 43, 331
NO_BLOCK_LOOP directive and pragma 76, 321

Index

NO_DISTRIBUTE directive and pragma 68, 321
NO_DYNSEL directive and pragma 117, 322
NO_FUSE directive and pragma 80, 322
NO_LOOP_DEPENDENCE directive and pragma 97, 282, 298, 322
  and apparent dependences 282
  improper use 299
NO_PARALLEL directive and pragma 122, 322
NO_PEEL directive and pragma 90, 322
NO_PROMOTE_TEST directive and pragma 92, 323
NO_SIDE_EFFECTS directive and pragma 51, 323
  caution 52
NO_UNROLL directive and pragma 323
NO_UNROLL_AND_JAM directive and pragma 86, 323
-noautopar compiler option 38, 122, 148, 332
-noblock compiler option 77, 334
node
  defined 431
node-parallelism
  and LOOP_PARALLEL 111
  and parallel regions 111
  and parallel tasks 111
  and prefer_parallel 111
  disabling 148
  specifying for loops 127
  specifying for regions 143
  specifying for tasks 138
  vs. thread-parallelism 111
NODE_PRIVATE directive 323
node_private memory 171, 432
  dynamic allocation 188
  incorrect pointer use example 287
  incorrect use example 285
  static assignments 177
NODE_PRIVATE_POINTER directive 324
nodes
  attribute to BEGIN_TASKS 138
  attribute to LOOP_PARALLEL 127
  attribute to PARALLEL 143
  attribute to PREFER_PARALLEL 127
nonblocking crossbar
  defined 432
nondeterminism
  parallel 303
-nonodepar compiler option 38, 114, 148, 332
non-uniform memory access (NUMA)
  defined 432
-nopeel option 90, 335
-noptst option 92, 335
-nore option 149, 345
-nosc option 41
notational conventions xxiv
-nsr option 94, 345
-nuj option 85, 333
num_node_threads() function 208
num_nodes() function 207
num_procs() function 206
num_threads() function 207
number of hypernodes
  finding 207
number of processors
  finding 206
number of threads
  finding 207
number of threads on hypernode
  finding 208
-nur option 82, 333

O

-O0 option 37, 43, 331
-O1 option 37, 48, 331
  and invariant code 256
-O2 option 37, 63, 88, 331
-O3 option 37, 105, 331
offset
  defined 432
opcode
  defined 432
operating system
  defined 432
opt_level pragma 324
optimization 263
  basic-block 48
  defined 433
  global 48
  IF-DO 87
  limits of 255
  machine-dependent 48
  potentially unsafe 345
  report 260, 347
  unsafe 57
Optimization column
  in array table 354
optimization level
  and opt_level pragma 324
  defined 433
optimization options
  -cache n 332
  -noblock 334
  -or table 331

optimization report 332, 347
* entry 349
analysis column 352
analysis table 352
Blocked entry 351
contents 347
Dist entry 349
DynSel entry 349
ID number column 348, 352
Inter entry 349
Interchange entry 351
iteration variable column 348, 352, 353
line number column 348
loop peeling 362
nested loop example 355
new loops 362
new loops column 350
optimizing/special transformations column 350
PARALLEL entry 349
PAR-NODE entry 349
Pattern entry 351
Peel entry 349
Promote entry 349
Reduction entry 351
Removed entry 351
reordering transformation column 349
Scalar entry 349
single loop example 359
StripMine entry 351
test table 353
Unroll entry 351
variable name footnotes 354
optimizing/special transformations
in loop table 350
-or table option 332, 347
order of evaluation 300
ordered
attribute to loop_parallel 127
ordered parallelism 221
example 222
ordered parallelization
defined 434
ordered sections 228, 324
and dependences 229
and gates 224
hidden 304
low-level CPSlib example 404
ordered task parallelization 138
ORDERED_SECTION directive and pragma 324
ordering
of data for proper alignment 42
output LCDs 120
overflow 309
error message 46
overhead
parallelization 115
oversubscript
defined 433
oversubscripting 280
oversubscription 280
defined 433

P

packet
defined 433
page
defined 433
page fault
defined 433
page frame
defined 433
PARALLEL directive and pragma 143, 293, 325
PARALLEL entry
in optimization report 349
parallel optimization
defined 434
+parallel option 393
parallel regions 142
and END_PARALLEL directive and pragma 317
and PARALLEL directive and pragma 142, 325
privatizing data in 325
parallel strip length
defined 441
parallel tasks
BEGIN_TASKS directive and pragma 315
END_TASKS directive and pragma 318
NEXT_TASK directive and pragma 321
Parallel Virtual Machine (PVM)
defined 434
parallel virtual machine (PVM) library 253
PARALLEL_PRIVATE directive and pragma 160, 161, 325
example 161
parallelism
and CPSlib barriers 399
and CPSlib programs 393
and loop induction variables 128
asymmetric example 397
block 393
chunk-based 129
cyclic 396
default 106, 109
disabling automatic 332
finding level of using compiler parallel support library 377
nested 111
node vs. thread 111
ordered example 221
simple example 106
stride-based 130
strip-based 107

symmetric 393
thread activity for two-dimensional 113
using compiler parallel support library
to determine presence 379
parallelization
and Fortran 90 constructs 114
asymmetric, using CPSlib 367
automatic implementation 107
basic operation 105
BEGIN_TASKS directive and pragma 315
by chunks 127
defined 434
in optimization report 357
inhibitors of 118
limitations 136
LOOP_PARALLEL directive and pragma 319
LOOP_PRIVATE directive and pragma 320
maximum threads in a loop 127
NEXT_TASK directive and pragma 321
nondeterministic execution 303
of for loops 128
of Fortran loops 128
of loops with calls 136
optimization level -O3 105
optimization overview 10
optimizations 115
ordered 127, 221
ordering dependences 324
PREFER_PARALLEL directive and pragma 326
preventing 122, 322, 327
region node-way 143
region thread-way 143
simple manual loop 126
simple manual task 137
specifying node 127
specifying threads 127
stride-based 130
symmetric, using CPSlib 365
task node-way 138
task thread-way 138
TASK_PRIVATE directive and pragma 328
parallelization directives
list 123
parallelization overhead 115
parameter
actual 409
defined 435
parentheses
use of 300
PA-RISC
defined 433
PA-RISC 7100 3
PAR-NODE entry
in optimization report 349
partial loop unrolling 81
path
defined 435
-pcc option
and aliasing 257
PEEL directive and pragma 90, 325
Peel entry
in optimization report 349
-peel option 90, 335
PEEL_ALL directive and pragma 90, 325
-peelall option 335
peeling
loop 89
physical address
defined 435
physical address space
defined 435
physical hypernode ID 209, 378
physical memory 14
access times 14
and subcomplexes 36
classes 167
configurations 2, 3
defined 435
GSM 3
hypernode local 14
interconnect cache 14
interleaving 29
maximum 2
minimum 2
partitioning 14
subcomplex-global 14
pipeline
defined 435
pipelining 39
defined 435
pointer
loop counter 261
pointers
default memory classes 186
memory class 183
memory class in Fortran 183
memory_class_malloc form 184
to shared data 185
potentially unsafe optimizations 345
pragmas
align_cti 22, 314
begin_tasks 315
block_loop 76, 316
C compiler 313
critical_section 146, 223, 316
dynsel 116, 317
end_critical_section 146, 223, 317
end_ordered_section 224, 317
end_parallel 143, 317
end_tasks 318
form 313
gate 318
loop blocking 76
loop_parallel 126, 319

loop_private 151, 320
misused 281
next_task 321
no_block_loop 76, 321
no_distribute 68, 321
no_dynsel 117, 322
no_fuse 80, 322
no_loop_dependence 97, 282, 322
no_parallel 122, 322
no_peel 322
no_promote_test 323
no_side_effects 323
no_unroll 323
no_unroll_and_jam 86, 323
opt_level 324
ordered_section 224, 324
parallel 143, 325
parallel_private 325
peel 325
peel_all 325
prefer_fuse 80, 326
prefer_parallel 126, 326
promote_test 327
promote_test_all 327
save_last 163, 327
scalar 104, 327
sync_routine 218, 328
task_private 158, 328
unroll 82, 329
unroll_and_jam 86
PREFER_FUSE directive and pragma 80, 326
PREFER_PARALLEL directive and pragma 126, 134, 326
Priv. Var. column
in privatization table 353
private data objects 15
privatization
of secondary induction variables 154
privatization information
in optimization report 354
privatization table 353
procedure
defined 435
procedure calls
parallelizing 136
process
defined 436
process memory
defined 436
processor cache line 16
processor functional units 38
program unit
defined 436
programming model 7
defined 436
message passing 7
message passing/shared memory hybrids 8
shared memory 7
Promote entry
in optimization report 349
PROMOTE_TEST directive and pragma 92, 327
PROMOTE_TEST_ALL directive and pragma 92, 327
promotion
test 91
propagating constants 45, 48
propagating copies 52
-ptst option 92, 335
-ptstall option 92, 335
PVM
defined 434, 436
PVM message passing 253
PVM/GSM library 253

Q

queue
defined 436

R

rank
defined 436
-re option 345
read
defined 436
REAL variables
effect of 57
recurrence
defined 436
recursion
defined 436
recursive call
defined 436
reduced instruction set computer (RISC)
defined 437
reduction
as dependence 298
C example 299
defined 437
Fortran example 298
parallelizable 121
strength 57
tree height 39, 40
redundant loads
elimination of 44
redundant-assignment elimination 43, 49

redundant-test elimination 88
redundant-use elimination 45
reentrancy
  defined 437
reentrant compilation 148, 345
  and loop replication 148
reentrant procedure
  defined 51
reference 437
region parallelization 142
  and other optimizations 143
  and PARALLEL directive and pragma 142, 325
  example 144
  specifying maximum threads 143
register
  allocation of 39
  defined 437
registers
  global allocation of 59
reordering transformation
  and incorrect answers 300
  in loop table 349
replication
  of loops 333, 335
report
  optimization 332
RISC
  defined 437
-rl option 335
rounding
  defined 438
roundoff error 255, 279, 309
ROW_WISE directive 327
row-major array storage 327
row-major order
  defined 438

S

SAVE_LAST directive and pragma 163, 327
Scalable Coherent Interface 2
  defined 438
scalable parallel processing (SPP) 1
scalable parallel processor
  defined 438
SCALAR directive and pragma 104, 327
Scalar entry
  in optimization report 349
scalar replacement 93
  and global register allocation 93
scheduling
  instructions 38, 43
SCI 2
  defined 438
scope
  defined 438
Secondary 154
semaphore
  defined 438
semaphores
  acquiring cache-based 386
  allocating cache-based for synchronization 386
  allocating memory-based 389
  and compiler parallel support library functions 385
  conditionally acquiring cache-based 387
  conditionally locking memory-based 390
  CPSlib low-level and critical sections 403
  CPSlib low-level and ordered sections 404
  freeing cache-based 386
  freeing memory-based 389
  locking memory-based 389
  unlocking cache-based 386
  unlocking memory-based 389
serial execution
  specifying for a loop 327
shape
  defined 439
shared data objects 15
shared memory
  advanced programming example 246
  basic programming 125
shared memory address space 14
shared memory programming 7
  advanced 205
shell
  defined 439
short-circuit evaluation of conditionals 41
side effects
  ignoring 323
SIMD
  defined 439
simplification
  algebraic 47
  trigonometric 47
single
  defined 439
SMP
  defined 441
socket
  defined 439
socket pair
  defined 439
source code
  defined 439
source file
  defined 439
space
  defined 440
span-dependent instructions 38
spatial reference
  defined 440
spatial reuse 70
  defined 437

spawn
  defined 440
spawn context 376
  defined 440
spawn thread ID 212
  finding using compiler parallel support library 376
spawn thread identifier
  defined 442
spawn_sym_t structure
  declared 370
spawning
  and compiler parallelism 108
  and two-dimensional parallelism 113
  using CPSlib 370
spin-waiting 110
SPMD
  defined 440
SPP (Scalable Parallel Processing) 1
  defined 438
spp_prog_model.h 174, 206
-sr option 94, 345
stack
  defined 440
  finding memory class of 211
  for spawned threads 148
  setting size for spawn threads 149, 372
statements
  ASSIGN 50
static memory class assignments 174
-std option
  and aliasing 257
stid
  defined 440
stop value 262, 263
  global variable 263
  variable aliasing 262
store
  defined 441
strength reduction 309
  arithmetic 57
  at -O1 57
  of induction variables 57
stride-based parallelism 130
stride-based parallelization
  defined 434
stride-parallelized loop
  example 131
strip mining 66
  conceptual example 108
  defined 441
  in optimization report 357
strip-based parallelism 107, 109, 130
strip-based parallelization
  defined 434
subcomplex
  defined 441
subcomplexes 33
  finding number of active threads
    using compiler parallel support library 379
  illustrated 34
  memory 36
  physical configuration 33
subcomplex-global memory 14
subexpressions
  elimination of 44, 53
subroutine
  defined 441
subscripts
  invalid 280
substitution
  of assignments 44
superscalar
  defined 441
support
  technical xxvi
suspended threads 110
symmetric multiprocessor
  defined 441
symmetric parallelism 365
  block 393
  cyclic 396
  examples 393
symmetric threads
  compiler parallel support library
    spawn functions 370
SYNC_ROUTINE directive and pragma 218, 328
  and CPSlib 391
synchronization
  and critical sections 224
  barriers 213
  CPSlib high-level functions 399
  defined 441
  denoting CPSlib routines 391
  denoting routines 218
  functions 215
  gates 214
  high-level functions 382
  in ordered parallel loops 229
  low-level CPSlib functions 403
  low-level functions 385
  manual 236
  tools for 213
synchronizing code 213
sysadmin
  defined 441
system administrator
  defined 441
system manager
  defined 442
system organization 11
system subcomplex
  defined 442

T

target machine
  specifying 332
target machine option 332
task list
  defined 137
task parallelization
  defined 435
  examples 140
  ordered 138
  specifying maximum threads 138
task private data 158, 328
TASK_PRIVATE directive and pragma 158, 328
  and CPS_STACK_SIZE 150
  C example 159
tasking directives 137, 139
temporal reuse 70
  defined 437
term
  defined 442
test elimination
  redundant 88
test promotion 91
  and loop replication 92
  disabling 335
  enabling 335
  in optimization report 353, 362
  preventing 323
  PROMOTE_TEST directive and pragma 327
  unlimited 327
test table
  analysis column 353
  column number column 353
  in optimization report 353
  line number column 353
  test transformation column 353
test transformation column
  in test table 353
thread
  defined 105, 442
  idle states 110
thread affinity 110
thread create
  defined 442
thread identifier
  defined 442
thread IDs 212
  and my_thread() 213
  and num_nodes() 213
  and num_threads() 212
  assignments 212
  finding 208
  kernel 212
  num_node_threads() 213
  spawn 212
thread-management functions
  in compiler parallel support library 370
thread parallelization
  specifying for regions 143
  specifying for tasks 138
THREAD_PRIVATE directive 328
thread_private memory 170
  dynamic allocation 187
  incorrect use 284
  static assignments 175
THREAD_PRIVATE_POINTER directive 328
thread-parallelism
  specifying for loops 127
  within node-parallelism 113
thread-private
  defined 443
threads
  attribute to BEGIN_TASKS 138
  attribute to LOOP_PARALLEL 127
  attribute to PARALLEL 143
  attribute to PREFER_PARALLEL 127
  finding topology on hypernodes 380
  setting wait attributes 381
  suspended 110
-tm option 332
topology
  of threads on hypernodes 380
traditional parallel computers 2
tree
  balancing 39
tree-height reduction 39, 40
  defined 443
triangular loops 290
  auto-parallelization 291
  parallelizing the inner loop 293
  parallelizing the outer loop 292
trip count
  defined 443
  overflow 304
type conversions
  eliminating 309, 310
  of constants 46
typographic conventions xxiv

U

-uj option 85, 333
-ujn option 85
unbalanced tree 39
unlock_gate function 217
unlock_gate_8 function 217
unroll and jam
  and loop replication 86
  compiler options 85, 333
  of loops 83
UNROLL directive and pragma 82, 329, 333
UNROLL_AND_JAM directive and pragma 86
unrolling
  and loop replication 83
  depth 81
  factor 81
  loop 80, 333
unsafe optimizations 57
  potential 57
unsigned
  defined 443
-uo option 57, 309, 345
  example 310
-ur option 82, 333
-urn option 82, 333
user interface
  defined 443
User Variable Name column
  in variable name footnote table 354
utility
  defined 443

V

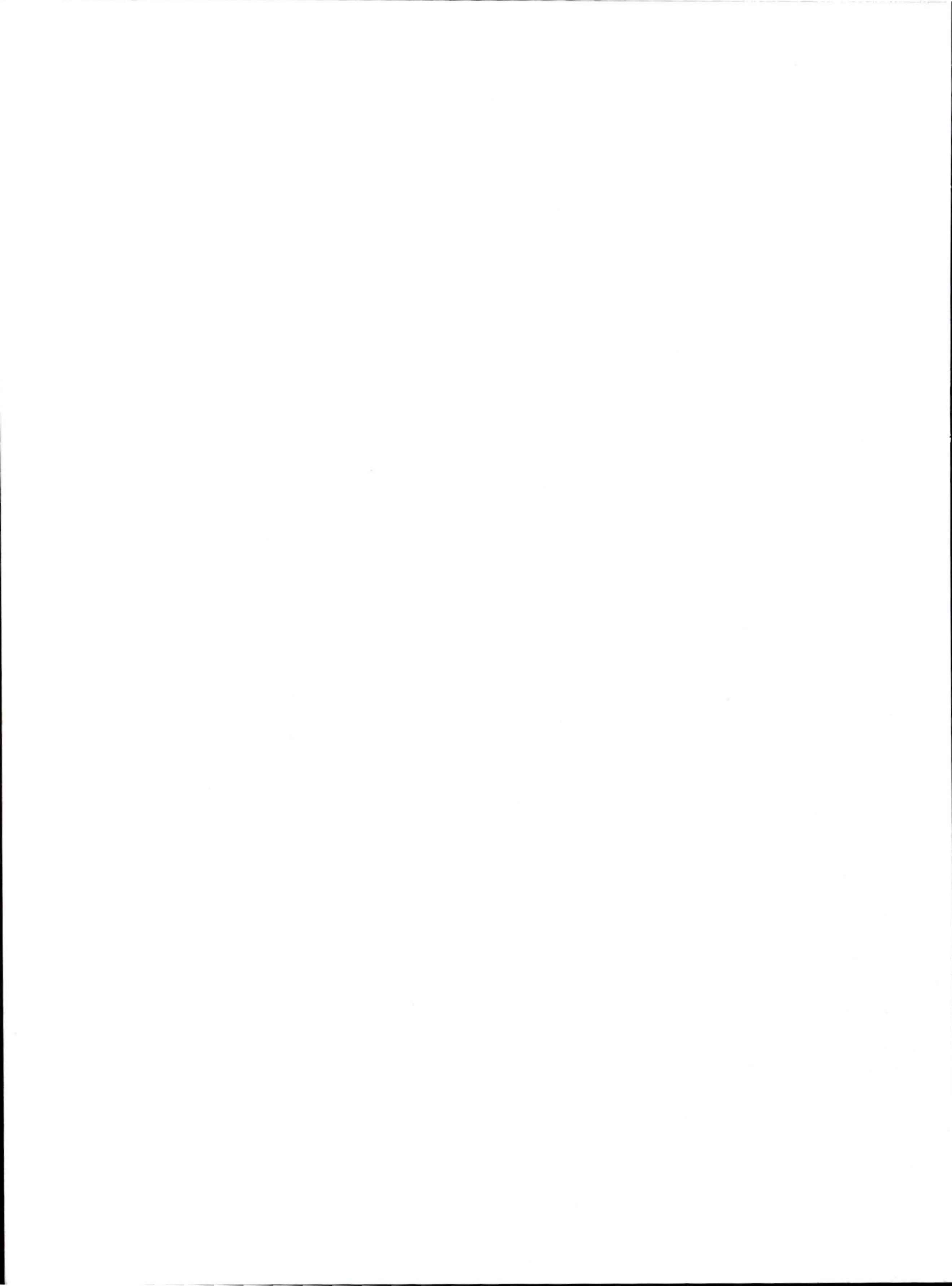
value
  iteration 301
Var. Name column
  in array table 354
variable privatization
  in optimization report 353, 354
  loop example 151
  LOOP_PRIVATE directive and pragma 150
  of secondary induction variables 154
  PARALLEL_PRIVATE directive and pragma 160
  region example 161
  saving last values 163
  task example 159
  TASK_PRIVATE directive and pragma 158
variables
  abbreviated in optimization report 354
  footnoted in optimization report 357
  induction 57, 311
vector
  defined 443
vector processor
  defined 443
virtual address
  defined 443
virtual address space 14
virtual aliases
  defined 444
virtual machine
  defined 444
virtual memory 15
  and subcomplexes 36
  block_shared 16
  classes 15
  defined 444
  far_shared 15
  near_shared 15
  node_private 15
  thread_private 15

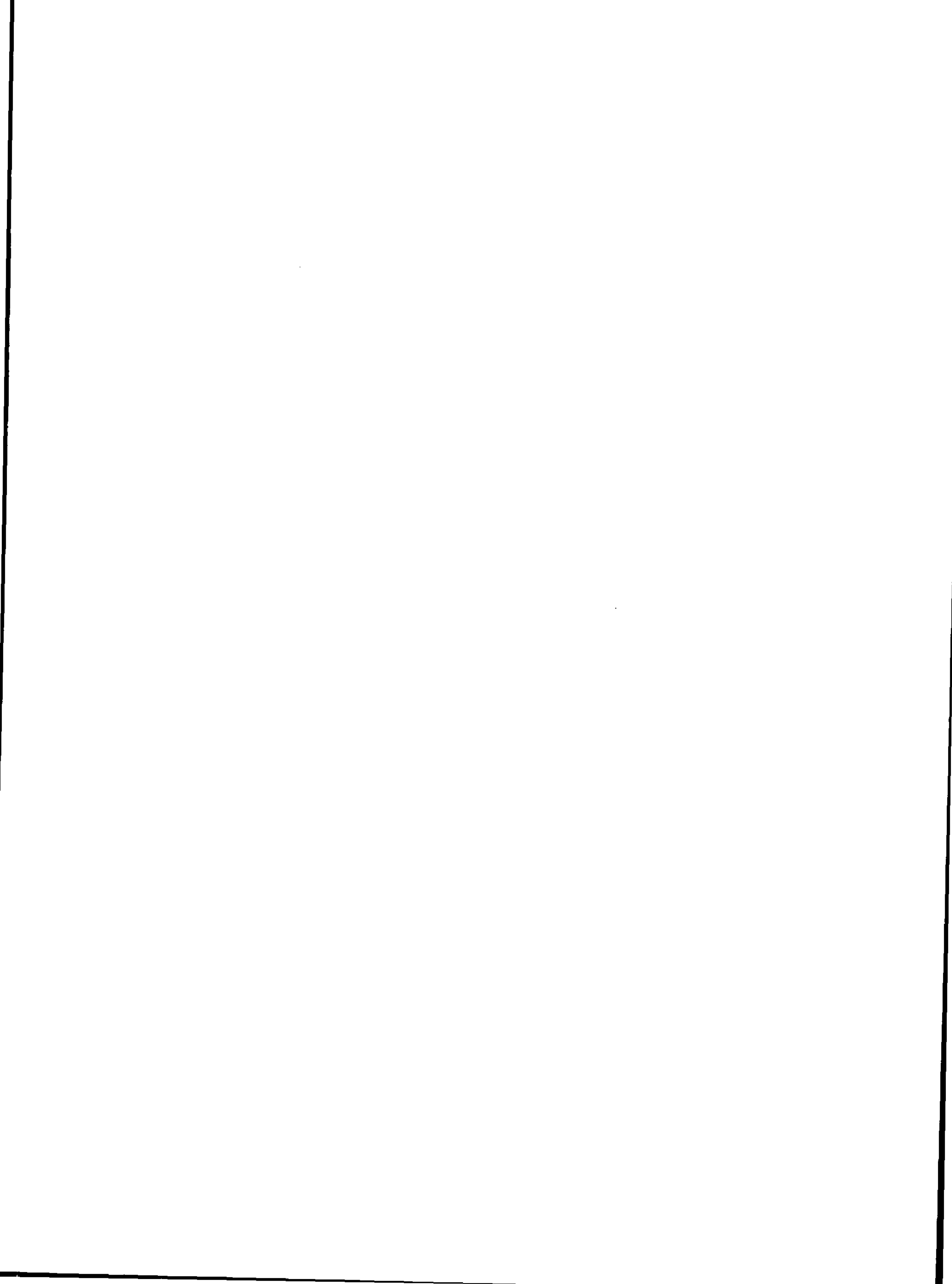
W

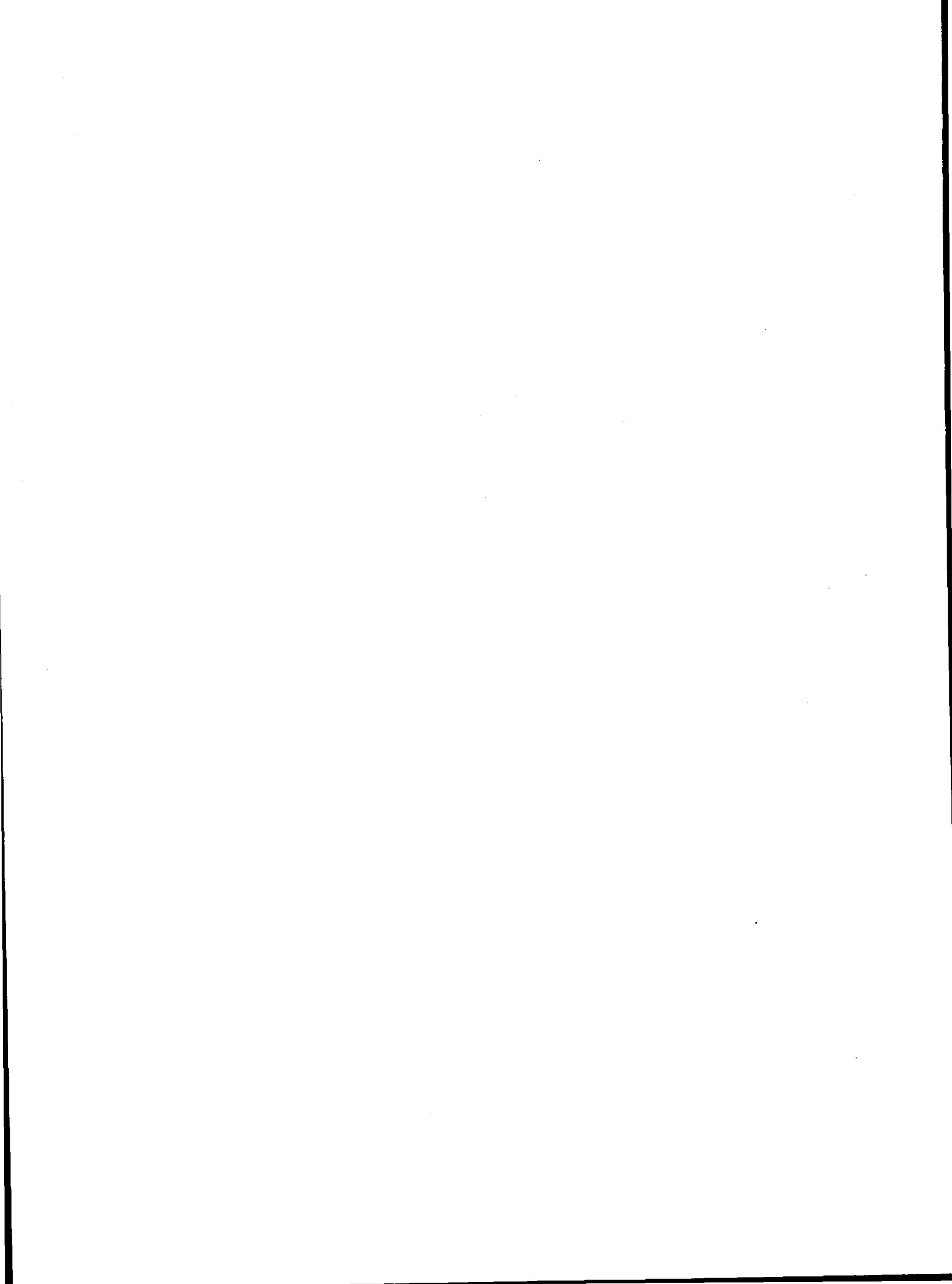
wait attributes
  for idle threads 381
wait_barrier function 217
wait_barrier_8 function 217
wall clock time
  defined 444
-Wl, option 393
word
  defined 444
workload-based dynamic selection 115
workstation
  defined 444
write
  defined 444
wrong answers
  and aliases 256
  and code motion 256
  and floating point imprecision 277
  and large trip counts 304
  and memory classes 283
  and no_loop_dependence 299

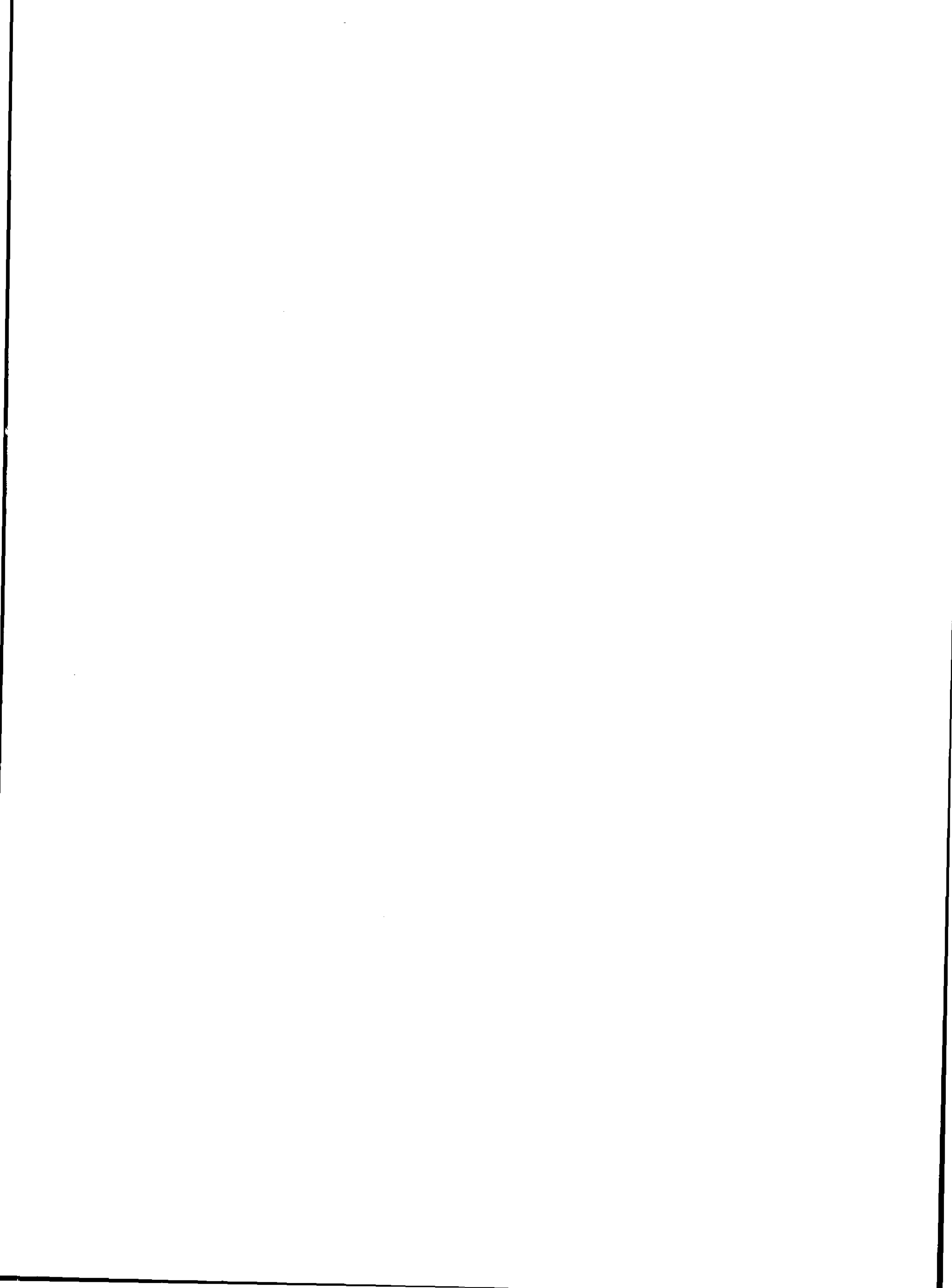
Z

zero
  defined 444
zero stride 301









CONVEX PRESS

ORDER NUMBER
DSW-067

DOCUMENT NUMBER
710-030230-004