Ada on a Hypercube

Russell M. Clapp and Trevor Mudge

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2110

Abstract

The widespread use of parallel machines, and hypercubes in particular, is being held back by the lack of high-order parallel programming languages. In this paper we discuss the issues involved in establishing an existing language that supports parallel processing, that is to say Ada, on a hypercube multiprocessor. An overview of the language is given, but the majority of the paper addresses the requirements and implementation of the run-time system, which is the key to establishing any parallel language. First, the requirements of the run-time system for Ada are described from a machine-independent point of view. Next, the approach taken toward implementing this system on a hypercube is discussed, with considerations given for language level program partitioning and interprocessor communication performance. Finally, the status of our current implementation is discussed and some concluding remarks are made about parallel languages in general, based on our experiences.

1 Introduction and Motivation

The widespread use of parallel machines is being held back by the slow rate at which high-order parallel programming languages and appropriate software development environments are being established for parallel machines. The availability of such languages will allow users of parallel systems to develop machine-independent concurrent software. This step to machine independence is critical if wasteful duplication of effort is to be avoided whenever application software is ported to a new parallel machine. Machine independent software will hasten the day when reusable software becomes a reality for parallel machines, as it presently is for conventional uniprocessors. It will also facilitate the development of truly parallel algorithms that can unlock the performance potential of parallel machines. This paper examines the problem of supporting an existing parallel language, Ada, on a large scale distributed-memory parallel computer, specifically a hypercube multiprocessor.

The programming of these distributed memory parallel machines is normally done by writing a separate program to run on each processor. These programs communicate using low-level message passing operations provided by the operating system and made available to the programmer through extensions to a sequential language (usually C or FORTRAN). Typically, these separate programs are copies of a single program that is written to allow different execution paths based on the program's location in the hypercube array of processors and on the data the program receives during its execution. This form of programming is referred to as the *Single Code Multiple Data* (SCMD) style [Buz88]. In the case where different programs are written for each processor, the programming style is referred to as *Multiple Code Multiple Data* (MCMD). Even in the MCMD case, though, the number of different programs written for a large scale multiprocessor is relatively small, since few applications require a large number of different interacting programs.

There are several problems with the above style of programming, and they are related to the separate program concept. Two major problems are the lack of type checking in communications between processors and the machine dependence of the code. These problems are can be solved by using a suitable parallel language. By parallel language we mean a programming language with units of concurrency that may be distributed across the processors of a multiprocessor and executed simultaneously. Such a language should provide strong type checking across processor boundaries, abstract interprocessor communications as interprocess communications, allow data sharing between processes to be specified at the language level, and provide for synchronous creation and termination of processes within a program. These features should all be implemented while still supporting SCMD style algorithms by providing a mechanism for processes to be replicated a large number of times. This support would allow a large class of algorithms to benefit from the other features mentioned earlier, and would provide a machine-independent language for programming parallel processors with single, coherent programs.

*This work was supported in part by Department of Defense grant number DOD-MDA904-87-C-4136

There are a number of languages that are being developed for parallel programming [BCG88,Dal88,FMO88, Ree88,Sha88]. The approach described here is based on Ada, an existing programming language that fits the above requirements. Ada is a procedural language, which supports parallelism. The key to establishing a parallel language on a multiprocessor is the run-time system. Before discussing this, a brief introduction to the Ada language is given. Then, the run-time system components necessary to support an implementation are discussed. Following that, the approach taken to distributing these components across the processors of a hypercube is described. The status of our current implementation on an NCUBE/ten is then presented as well as some concluding remarks regarding parallel languages and Ada in particular.

2 Overview of Ada

Ada is a general purpose procedural concurrent language that is also intended for use in real-time systems. It also includes features designed to support object oriented programming such as encapsulation, information hiding, generic units, and strong type checking. The real-time aspects of the language include support for timing and multitasking. Other language features include exception handling, dynamic memory allocation and object pointers.

Ada has a Pascal-like syntax and is block structured in a way similar to Algol. This block structuring allows for blocks to be declared and nested as well as procedures and functions (known collectively as subprograms) and tasks (which are the unit of concurrency). This structuring provides a shared memory model in that nested units can access variables declared in an enclosing scope. In the case of tasks, a task nested within a block, subprogram, or another task is said to be a *dependent* of that unit. A unit may not be exited until all tasks dependent upon it have terminated.

Encapsulation in Ada is provided by packages. Packages have a specification part and a body. The specification may contain type definitions and variables as well as subprogram and task specifications. The specification serves as the interface to the package. The body of the package may contain additional types and variables as well as subprogram and task bodies. Package specifications and bodies may be compiled separately and may also be used as library units. These packages may be imported by subprograms and other packages that serve as compilation units. This makes the imported package available for access through the interface defined in the specification.

Communication between tasks in Ada can be synchronous through the rendezvous mechanism or asynchronous through shared memory. Although memory sharing is provided for through block structuring and package specifications, it is the responsibility of the programmer to assure mutual exclusion in access to shared objects. A mechanism to protect shared data that closely resembles monitors can be easily implemented at the language level using tasks. One task can communicate directly with another by calling an entry of a task. The task receiving the call executes an **accept** statement on one of its entries to receive the call. The calling task names the entry it wants and the task possessing it. The receiving task accepts the next call on a FIFO queue of tasks all waiting on that entry (the receiver does not name a particular calling task in the accept statement). Parameters can be exchanged in both directions during a rendezvous, and the receiving task can execute an arbitrary number of statements while the two tasks are synchronized. Thus, from the caller's point of view, an entry call appears much like a procedure call.

There are variations of the entry call and accept statement that allow for logical conditions and time bounds to be included. An entry call may be conditional, so that the call is executed only if the called task is waiting on that entry. An entry call may also be timed, so that the call is canceled if it cannot be started within a specified time period. A time bound may also be placed on a group of one or more accept statements, so that a task can proceed with its execution if no call on any of its possible entries is received within the given duration. The entries and time bound are grouped as part of a **select** statement. The entries are said to form the alternatives of the select statement, because only one of them can be accepted each time the select statement is executed. The time bound for a select may also be replaced with the reserved word **else**, which indicates that a call is accepted on one of the entries only if it is already pending. A third case for the alternative to a group of accept statements is for the task to terminate. This action is taken if all sibling tasks are either terminated or waiting to terminate at their own **terminate** statements. This mechanism is used to synchronize task termination within the hierarchical structure. Orderly termination is further enforced by the rule requiring child tasks to be terminated in order for a parent task to terminate. The **delay**, **else**, and **terminate** alternatives of the **select** statement are mutually exclusive, i.e., only one of the three types may appear along with accept statements. Accept statements and select alternatives may also be guarded; boolean expressions must first be evaluated to determine whether an accept statement or select alternative should be considered.

Further discussion of Ada may be found in [Bar84], and the complete language definition appears in [LRM83].

2.1 Ada Example

To provide a concrete illustration of some of the above features, in particular using tasks to obtain parallelism, consider the sieve of Erastosthenes to select all of the prime numbers less than some integer M. Although many approaches to solving this problem are possible, a straightforward solution can be programmed using a "bucket brigade" of tasks to form a pipeline. The main program generates a stream of odd numbers which are input to the pipeline of tasks. The stream threads through the tasks and each task, T_i, picks off the i-th prime number, P_i, prints it, and then "filters" the remainder

of the stream so that all multiples of P_i are removed.[1] The resulting data stream is then passed to the next task in the pipeline. At each task, a number in the stream is received, processed, and then passed on after which the next input number is read. The entire pipeline of tasks are doing these steps simultaneously resulting in a high degree of concurrency.

The sieve program can be effectively run on a hypercube by embedding the pipeline structure into the hypercube configuration. A natural arrangement would place each task in the pipeline on adjacent processors in the form of a "snake". However, since the parallelism is specified at the language level without specifying the assignment of tasks to processors, other assignments of tasks to processors are possible.

A diagram of the logical flow of data is given in Fig. 1. Figure 2 shows the operation within each pipeline task using pseudo-code. The code of an actual Ada program is given in Fig. 3. Key words are shown in lowercase and user defined names and variables in uppercase.

In the program, the tasks $T_1, T_2, ...T_N$ are represented by the elements of the array `TASK_ARRAY`. They are identical and are instantiations of the task type `SIEVE`. The main program generates the stream of numbers by passing all odd numbers greater than or equal to 3 to the task `TASK_ARRAY(1)`. The task code accepts the entry `PLACE` in two locations in order to distinguish its first receipt of a call on this entry. This distinction is made because, it can be shown, the first call will always transmit the i-th prime number. It is copied into `P`. The value of `P` is printed, and it is also used to filter the remainder of the incoming stream. Filtering is performed within the select statement that is embedded in a loop. Until all of the tasks, `TASK_ARRAY(1)` ... `TASK_ARRAY(N)`, become idle, each task, `TASK_ARRAY(I)`, waits for a call on `PLACE` within the select. The value read in here is compared to the closest multiple of `P`. If this multiple is equal to the value read, no action is taken and the value is effectively filtered from the stream. Otherwise, the value is passed on to the next task, `TASK_ARRAY(I+1)`, in the pipeline. When the data stream has passed through all of the tasks, they are all waiting on the entry `PLACE` in the select statement with the terminate alternative. At this juncture, all the tasks terminate and the program ends.

The operation of the task `TASK_ARRAY(N)`, where $N = \sqrt{M}/2$, is different from the others because it prints all the numbers that it is passed. This is because the number of tasks needed to perform the sieve is bounded by N, and all of the values passed to the last task can be shown to be prime. In particular, no primes greater than the square root of the maximum value, M, need to be considered as factors because any number less than M that has as a factor a prime number greater than $\sqrt{M}$ will also have a prime number factor less than $\sqrt{M}$ and will thus be filtered out. Also, since we know trivially that even numbers are not prime (with the exception of 2), we can bound the number of tasks needed to sieve all prime numbers less than M by $N = \sqrt{M}/2$.

[1] We assume the 0-th prime, $P_0 = 2$.

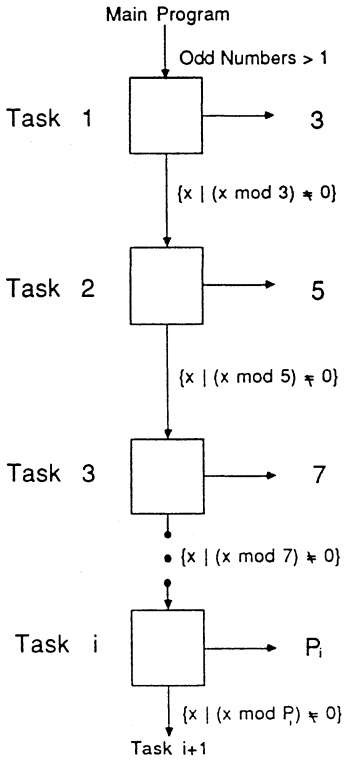

Figure 1: Task pipeline.

3 Run-Time System Requirements

Because of the large number of features available in the Ada language, there are many operations which must be included in the run-time system. These run-time system requirements can be specified independently of the target architecture. They are summarized in the points below.

Memory Management In addition to supporting the normal allocation and deallocation of storage of a Pascal-like language, a mechanism must be established to support the shared-memory model of Ada. This is necessary for the case where tasks are nested and may reference variables in an enclosing task. This requirement is more complicated than the case of nested blocks, because each task also needs its own independent stack space. The approach taken is to allocate a fixed amount of stack space for each stack and interconnect them in a *cactus stack*. In this data structure, a task's local stack space points back into the stack of its parent where visible variables, subprograms, and/or tasks may be located. This structure allows several tasks to share the trunk of the cactus stack while still maintaining their own individual stacks.

The run-time system must also support the dynamic allocation of memory objects of all types including tasks. Dynamically created tasks must also have their stacks linked into the cactus stack.

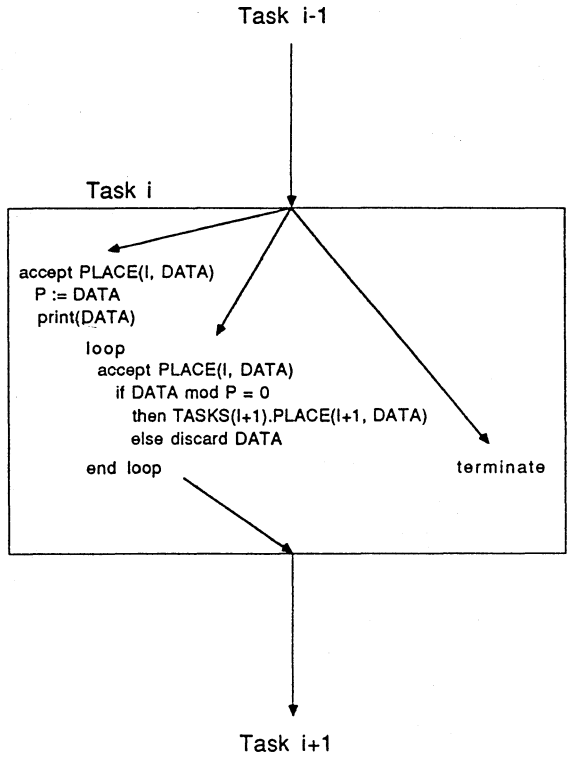

Figure 2: Task pseudo-code.

Task Activation and Termination The language rules for Ada specify that activation and termination of tasks be synchronized. A task may not activate until all of its children are active. Conversely, a task may not terminate until all of its children are terminated. In the case where a task depends directly on a block or subprogram, the execution of the parent code may not pass an **end** or **return** statement until the child task is terminated. Tasks may also be abnormally terminated by an **abort** statement, with the abnormal termination being propagated to all dependent tasks.

These rules provide for orderly management of the cactus stack, but require additional work by the run-time system. Several different task states are necessary so that the run-time system can coordinate the activation and termination. Tasks must be aware of their dependents and a mechanism for communicating state changes among these tasks is needed. Support is also needed for processing abort statements and the terminate alternative of the select statement.

Timing Support The notion of time is provided by Ada and is available for use in the form of a time-of-day clock, task delays, and time limits on task communication. Although not required by the language, the run-time system may also support the time-slicing of multiple tasks on a single processor. Since this is usually desirable, a single interrupting count-down timer (interval timer), normally available, should

```
with TEXT_IO; use TEXT_IO;
with MATH; use MATH;
package SIEVE_DECS is
  task type SIEVE is
     entry PLACE(I, DATA: in INTEGER);
  end SIEVE;

  M : INTEGER := 10_000; --Number ceiling
  N : INTEGER := INTEGER(SQRT(FLOAT(M)) / 2.0);
  TASK_ARRAY : array(1..N) of SIEVE;
end SIEVE_DECS;

package body SIEVE_DECS is
  task body SIEVE is --Code for task array
    P, NEW_NUM, CURRENT, LOC : INTEGER;
  begin
    accept PLACE(I, DATA : in INTEGER) do
      LOC := I;  --LOC is location in pipeline
      P := DATA; --P is prime for this task
    end PLACE;

    PUT_LINE(INTEGER'IMAGE(P)); --Print P
    CURRENT := P;

    loop  --Loop until termination
      select
        accept PLACE(I, DATA : in INTEGER) do
          NEW_NUM := DATA;
        end PLACE;
          --Check NEW_NUM against multiple of P
        if NEW_NUM > CURRENT then
          CURRENT := CURRENT + P;
        end if;

        if NEW_NUM < CURRENT then
          if LOC = N then --The last task
            PUT_LINE(INTEGER'IMAGE(NEW_NUM));
            else           --Pass data on
              TASK_ARRAY(LOC+1).PLACE(LOC+1,
                                      NEW_NUM);
            end if;
          end if;

      or
        terminate;
      end select;
    end loop;
  end SIEVE;
end SIEVE_DECS;

with TEXT_IO; use TEXT_IO;
with SIEVE_DECS; use SIEVE_DECS;
procedure MAIN is
  DATA : INTEGER := 3; --For data stream
begin --Beginning of main program
  PUT_LINE("2");
    --Pass stream of odd numbers to first task
  while DATA < M loop
    TASK_ARRAY(1).PLACE(1,DATA);
    DATA := DATA + 2;
  end loop;
end MAIN;
```

Figure 3: Ada code listing.

be dedicated to this purpose.

The strategy for managing several time delays and limits is based on the time-of-day. The run-time system manages a queue of events, a *timed events queue*, that is ordered in increasing time-of-day values. Whenever a time-slice interrupt or system call is made, a check is performed to see if the current time-of-day implies that any delays or limits have expired. If so, the associated events are processed and removed from the queue. Obviously, with this scheme it is possible for a delay or time limit to last longer than specified. However, this is consistent with the language definition, which states that a time delay must last at least as long as the delay value specified [LRM83].

This scheme may also be used in conjunction with an additional interval timer that is not used for time-slicing. At each interrupt, the next delay interval in the list is loaded into this timer. In the case where a second timer is not available, a time-of-day clock is relied upon. If such a clock is not available in hardware, careful use of the time-slicing interval timer is required. A scheme for using a single timer to implement all timing support (i.e. time-slicing, time-of-day, and delays) is described in [CMV87]. This scheme has yet to be extended to cover the multiple processor case, however. This problem is discussed in Sec. 4.2 below.

Task Communication A major requirement of the run-time system is support for the rendezvous mechanism. This must cover the simple, conditional, and timed entry calls as well as the various alternatives of the select statement for accepting entry calls. Entry queues must be provided for each entry to store pending calls that cannot be immediately accepted. Workspace for rendezvous parameters must also be provided.

The run-time system implements the necessary synchronization through the use of several additional task states. Calls that cannot be immediately accepted are placed in entry queues, and also added to the timed events queue in the case of a timed entry call. In case of a call time-out, the entry call is removed from the entry queue. Tasks accepting entry calls take them from the appropriate queues, or wait for one to arrive depending on the accepting code used. If a time bound is placed in a select statement, an entry is made to the timed events queue. When the delay expires, the task moves on to the next statement beyond the select if no call was accepted. Guards for accept statements are evaluated by code generated by the compiler and their values are passed to the run-time system.

Run-time system calls are made by tasks attempting to call or accept an entry. The run-time system must provide mechanisms for exchanging state information and rendezvous parameters between tasks in addition to the requirements stated above.

Exception Handling and Propagation Exception handling is provided in Ada as a means for recovering from error conditions. When an exception occurs, it must be trapped by the operating system or run-time system (in the case of division by zero) or detected by the run-time system or generated code (in the case of a tasking error or an array bounds violation). When an exception occurs, a branch to the handler for that exception occurs. This address is contained in the current call frame. If no handler exists at this level, the exception is propagated up the dynamic calling chain until either a handler is found or the program is aborted.

It is also possible in some cases to propagate exceptions between tasks. When a task is elaborated, if an exception occurs, it is propagated to its parent task. Also, an exception may be propagated during a rendezvous if the called task has no handler or if the calling or called task is aborted. Outside of these situations, exceptions are not propagated outside of tasks.

The run-time system then, in addition to detecting and trapping exceptions, must ensure that proper handlers are found by propagating exceptions either by rolling back the call stack or by notifying a task as necessary.

Task Scheduling The run-time system must provide a task scheduler, which may involve time-slicing as discussed above. The scheduler must switch contexts between tasks at scheduling points by saving state, selecting another task to run, and dispatching that task. The state of a task is saved by storing the program status word and program counter and saving all necessary registers in the task's local task space. *Scheduling points* occur whenever operating system or run-time system calls are made in addition to time-slice interrupts. Scheduling should be fair, and account for task priorities if present.

As an aid to the scheduler and run-time system in general, each task has a data structure referred to as a task control block (tcb). This structure contains the task identifier, program status word, program counter, state identifier, local stack space, pointers to parent's and dependent's tcbs, pointers to rendezvous parameter space and entry queues, and possibly other information. These tcbs may be linked in a circular list or linked into queues depending on their state. A ready queue of tcbs is kept in order of priority.

Additional Requirements Other run-time system requirements include support for generic units, compiler attributes, I/O, predefined packages, and interrupts. Generic units are mainly supported by the code generator, but may require some run-time type checking. Some compiler attributes are queries to the run-time system requesting basic information, e.g., E'COUNT returns the number of tasks currently waiting on entry E. These are supported as calls to the run-time system. I/O, predefined packages, and interrupts are all highly system dependent but each has a language level specified interface. The role of the run-time environment in this case is to implement that interface. Each of the above listed features is discussed in more detail in [CIM88].

4 Distributed Run-Time System

In order to support the distributed execution of a single Ada program on a hypercube multiprocessor, the run-time system must be extended to account for multiple processors that lack a common shared memory. In an effort to keep the requirements of the run-time system at a reasonable level, certain restrictions are placed on the units of an Ada program that may be distributed across multiple processors. We examine these restrictions in the next section, and then present the associated run-time system requirements in the following section. Consideration is then given to relaxing these restrictions at the expense of additional run-time system overhead. This may be desirable in the case of a target machine with very fast interprocessor communication.

4.1 Language Units of Distribution

The Ada language definition does not specify how a program is to be partitioned for execution on multiple processors. This decision is left to the implementor. We have chosen to restrict the allowable units of distribution so that a reasonable amount of granularity may be obtained without unnecessarily complicating the run-time system. This was also one of the goals of the study done in [VMB88], where library packages and library subprograms were proposed as units of distribution. They were chosen so as to reduce the number of potential remote references and to eliminate the need for cross-processor dynamic scope management. For example, the above units of distribution ensure that nested blocks and subprograms cannot be remotely located from their enclosing scope.

As discussed in [ClM88], more flexibility is desired for distributing Ada programs on a large scale distributed memory multiprocessor such as a hypercube. To allow for this, we propose the same units as in [VMB88], but also allow tasks that are declared or have their types declared in a library package specification to be distributed. We also allow these tasks to be distributed when they are array components. This scheme allows tasks that are dependents of library packages to be distributed, but requires that tasks nested within other tasks reside on the same processor as their parent. This is true because nested tasks must be declared in the parent task's body, which must be defined in a package body, and such tasks are not allowed to be distributed according to the rules given above.

These units of distribution allow a large number of tasks, possibly identical, to be distributed to separate processors while still retaining a flexible naming scheme [ClM88]. This allows SCMD style algorithms to be easily supported within the parallel language. Also, these distributable units save the run-time system some effort, by not allowing nested tasks to be distributed. This restriction simplifies the implementation of task termination via the terminate alternative of the select statement, and prevents the need for cactus stack pointers to cross processor boundaries, thus reducing the number and type of potential remote references.

In order to further simplify the run-time system, our initial approach disallows the dynamic migration of tasks once they have been assigned a processor. Instead, statically declared tasks, as well as library packages and library subprograms are assigned a processor at compile-time. Dynamically created tasks are loaded onto the processor executing the allocating statement and are thereafter prohibited from moving. While this scheme does not allow for dynamic load balancing of tasks, it does simplify the run-time system's job of locating tasks and does not violate the restrictions stated in the above paragraph. The prospect of migrating tasks is discussed further in Sec. 4.3 below.

4.2 Run-Time System Components

The chosen units of distribution together with the target architecture dictate the specific requirements of the run-time system. A summary of these requirements is outlined in the subsections below. Additional details regarding these run-time system components including implementation strategies can be found in [ClM88].

Run-Time System Kernel The first step in implementing the run-time system for a hypercube target is to replicate the task scheduling kernel on each processor. Along with this, the various data structures associated with a particular task should be located on the same processor as that task. These data structures include tcbs, entry queues, and rendezvous work space.

In order to support language features across processor boundaries, the run-time system needs an interprocessor communication facility. In most cases, this is provided by the operating system. In our implementation on the NCUBE, we made use of the existing store and forward communication facility, Vertex. Because of the need for communication between processors that do not share a physical memory, a message passing based run-time system was designed. This system implements all communication between tasks in the run-time system as messages. A queue of pending messages is read and processed at each scheduling point. The processing of messages results in changing task states, system queues, or sending more messages. The inspiration for the message based approach came from the work described in [Wea84]. An implementation of a partial run-time system has been built and is described in [CMV87]. This implementation provides the basic foundation for building a complete Ada run-time system.

Memory Management As mentioned in Sec. 4.1 above, the units of distribution and restriction on task movement eliminates the possibility of cactus stack pointers crossing processor boundaries. This allows each processor to manage a cactus stack in the same manner as the uniprocessor case. Relative addressing is then used in the place of remote memory references involving message passing.

In the case of dynamic non-stack allocation and deallocation of objects, the run-time system manages a heap of storage

and is responsible for error checking on requests and raising an exception when storage is exhausted. In most cases, the run-time system allocates blocks of memory from the operating system, which is the situation with Vertex. The run-time system should allocate large blocks to form a heap, and then manage the heap accordingly. This reduces the number of calls to the operating system. An additional benefit of this scheme is that the run-time system provides an operating system independent interface for the code generator.

The class of objects that may be allocated dynamically includes tasks. In this case, storage must be allocated for the code as well as the data structures associated with the task. The run-time system must also initialize this storage. If the task being created has its type defined in the package residing on the processor executing the allocation, a local memory copy may be needed to initialize the code section. Otherwise, the code must be loaded from a remote processor, using the message passing primitives provided.

Task Activation and Termination The synchronization of task elaboration, activation, and termination is easily implemented in the parallel case. This is due to the language units of distribution and the restriction of task migration. These rules create a situation where the only task dependency to cross a processor boundary is that of a task depending upon a library package. A library package is not an active body of code, and the language definition states that the termination of a task that depends on a library package is not defined [LRM83]. This gives the implementor the freedom to let tasks that are dependent on packages hang indefinitely on a terminate alternative, but places the burden of detecting program completion on the programmer, who may terminate tasks explicitly with the **abort** statement. It is possible, however, for the run-time system to support collective termination of distributed tasks. Strategies to implement this support have been proposed and are discussed in Sec. 4.3 below.

The minimum requirement, then, for support of synchronized activation and termination is the same as in the uniprocessor case. Since all dependencies are local to a given processor, task states can be readily examined and altered by the run-time system kernel. This is also true of tasks terminating via the terminate alternative of a select statement. Quiescence of tasks all waiting to terminate on a single processor can be detected in a straightforward manner [BaR85].

Timing Support The approach for implementing basic timing support is to replicate the linked list structure of timed events on each processor. This assumes that each processor has a mechanism for keeping track of at least the relative passage of time. This capability is usually supplied in the form of an interval timer on each processor, as is the case with the NCUBE. The linked list on each processor contains records pertaining to events related to delays and rendezvous time-outs for tasks residing on that processor.

As stated in Sec. 3, it is possible to use a single interval timer to manage both task time-slicing and to keep track of the relative time since the processor was initialized. What is needed, though, is a common sense of the time-of-day by all processors to support the CLOCK function of the CALENDAR package. This function returns the absolute time-of-day when called. In principle, the CLOCK function may be supported by synchronizing all interval timers before beginning any program, then keeping track of time passage as before. However, this scheme requires a significant amount of start up overhead, and is subject to drift due to ticks lost during timer manipulation.

Another possible solution is to implement a centralized time-of-day server. However, the time delay in accessing such a server may be too great to make the returned value reliable. The value may be adjusted to compensate for this overhead, but the exact amount may be indeterminable, due to variable delays in message passing and conflicts arising from multiple simultaneous requests. This approach is discussed further in Sec. 4.3 below.

The best solution to this problem is support in hardware for two timers per processor. In addition to the interval timer, a time-of-day clock is also provided. This allows time-slicing and the event queue to be managed as described above, but the time-of-day is provided by a separate clock. A similar solution that is proposed in [VoM87b] utilizes a time-of-day clock along with a readable/writable compare register. The compare register contains an absolute time value that indicates the next timer interrupt. In the case of either solution, though, the time-of-day clocks for all processors must be synchronized. This can be achieved by driving all ticks from the same line, as is currently done with the interval timers on the NCUBE.

Task Communication In common with the support for other language features across processors, a message passing scheme is needed to implement the rendezvous mechanism. These messages must indicate the task state changes that are needed in executing a rendezvous as well as transmit parameters. A straightforward approach is used, with a minimum number of messages.

The message passing begins with the calling task requesting the rendezvous. In the case of the simple call, the request is sent with parameters and the calling task waits for a reply with results. If the call is conditional, a call message with parameters is also sent, but the run-time system on the receiving node sends a negative reply if the called task can not immediately accept the rendezvous. If the called task is waiting to accept the call, a reply message is not sent until the critical section is executed and results are returned. This is possible because the calling task is suspended awaiting a reply in both cases.

The message passing protocol for the timed entry call is more complex. A total of four messages are required to execute the rendezvous. As before, a call message is sent initially, but an entry to the timed events queue is also made. This entry in the timer queue is based on the time-out interval supplied in the call. When the accepting task is ready to exe-

cute the critical section, a reply message is sent indicating this fact. If the time-out has yet to occur on the calling node, a "go ahead" message is sent, and the rendezvous is executed. Upon completion, a reply message is sent with results. If the time-out does occur before the go-ahead is sent, an abort message is sent to cancel the request. This message identifies the specific call to be canceled, and allows the called task to recover whether or not it had sent a ready message. The protocol described here for the timed entry call was modeled after that in [Wea84].

As for the accepting task, message replys and state changes are processed according to the type of accept. In the simple case, the incoming request is placed in the appropriate entry queue unless it can be immediately serviced. When it can be serviced, the first message in the queue is removed, and the reply is sent after execution of the critical section. If the task attempts a simple accept when no calls are pending, it waits indefinitely until one arrives.

In the case of a select statement surrounding several accepts, the task executes any one of the entries with a true guard from those that are queued. If no such calls are pending, the task waits for a call to arrive. If a delay alternative with a true guard is present, an entry to the timed events queue is made to bound the amount of time that the task will wait for a call. In the case of an else alternative, the code following the else is executed if no calls are pending on the selected entries with true guards. If a terminate alternative is present instead of a delay or else alternative, the task will wait on the selected entries if none are pending until a call arrives or the task is terminated by the run-time system.

Further details of this implementation can be found in [ClM88] and [CMV87].

Remote Subprogram Call and Object Reference Support for remote subprogram calls and remote object references must be included in the run-time system since these entities are bound to processors according to the package they reside in, and packages may export interfaces via specifications to code units on other processors. It is also possible in some cases for objects that are declared in a package body to be visible to tasks taken from the same package but distributed to a remote processor. References to these objects and subprograms must be implemented via a run-time system call. Among the parameters to such a call is the identifier of the processor holding the object. This is known to the caller as long as the binding specification is made when the package containing the object is compiled.

Remote references can also occur through the use of access variables (commonly known as pointers). To address this problem, access values are implemented as processor–address pairs. Each pointer reference then results in a check of the processor part to see if the reference is remote. If it is, a call to the run-time system is made. An alternative approach to this problem could be to disallow the passing of pointer values across processors, but this would violate the language definition.

In all cases of remote reference, the referencing task is blocked while the run-time system makes the request by sending a message to the kernel on the processor holding the object or subprogram. Because the remote processor may be more than one hop away on the hypercube and the run-time kernel at the remote site must process the request before responding, the requesting processor performs a context switch to allow another task to run while awaiting a reply. The reply message may contain an acknowledgement of a write to an object, the value of an object being read, a function return value, or parameters returned from a procedure. The value returned is passed back to the requesting task. In the case of a remote subprogram call, the call parameters are placed in the message and the run-time system at the receiving end places these values on the stack space of a server task; this task is scheduled like any other. This approach is similar to the scheme described in [BiN84], except there is no need to bind the caller and callee dynamically, because the location of the subprogram is known when it is compiled.

Exception Handling and Propagation The additional support for exceptions needed in the multiprocessor case involves the propagation of exceptions across processors. This can occur when exceptions are raised in remote subprograms where there is no handler or in the case of an exception in a rendezvous between tasks on different processors. In these situations, the return messages normally sent must include an indication of the exceptional condition that is being propagated. In the case of a calling task in a rendezvous being aborted, a message must be sent to cancel the rendezvous if it has not already started. If it has begun, the called task executes the rendezvous as it normally would and no exception occurs.

4.3 Extensions for Fast Communication

In the prescence of very fast internode communication time, it may be desirable to remove some of the restrictions on program distribution and change the implementation of the run-time system. Possible changes would affect task migration, task distribution, a network sense of time, and remote object access. The possibilities are discussed in the paragraphs below.

Migration of Distributable Tasks One approach in allowing the migration of tasks is to allow only tasks that are currently distributable to be migrated. This may involve the implicit migration of nested tasks, in order to preserve the guarantee that nested tasks all reside on the same processor as their parent. In all cases of task movement, all data structures associated with the tasks as well as all code must be moved via internode communication. In the case where a large amount of memory is available on each processor, it may be possible for each node to retain a copy of all migratable code so that code movement is not necessary. The advantage of this approach to migration is that it does not

violate the assumptions that simplify the cactus stack implementation and synchronized task termination.

Distribution and Migration of all Tasks This unrestricted migration is implemented as in the case above, but the run-time system must also support cactus stack pointers across processors and the synchronized activation and termination of tasks across processors including the termination of tasks via the terminate alternative of the select statement. Cactus stack relative references would have to be modified to incorporate the use of pointers that indicate processor as well as address. Synchronized activation and termination can be implemented in a straight forward manner using message passing when states need to be altered. A solution for the simple case appears in [Wea84], and a solution for the case of the terminate alternative of the select statement can be obtained by adding messages to the solution given in [FSS87]. However, unrestricted migration of tasks also causes a problem when a task needs to be located. A possible approach to solving this problem is to adopt the method presented in [Ros87].

Time-of-Day Server In the case where only a single interval timer is available on each processor, a centralized time-of-day server is a possible approach to solving the common system wide sense of time problem. The usefulness of such a timer is dependent on the its access time with respect to its resolution. In addition to the time needed to read the timer, the access time is made up of the communication delay and the time to resolve access conflicts if more than one processor requests the time-of-day simultaneously. If this access time can be limited to a small value in comparison with its resolution, then a centralized time server can be used to support a common sense of the time-of-day throughout the system.

In the case of the hypercube, the time server can be implemented by a host processor or a dedicated node processor. The communication delay in accessing the timer is then bounded by the maximum number of hops between the caller and the time server. In the case of a dedicated node processor as the server, this number of hops is the dimension of the hypercube.

Remote Object Reference In the presence of fast communication times, it may be desirable to implement a remote object reference as a *processor synchronous* operation, as suggested in [LeB82]. In this approach, the referencing processor remains idle while it awaits a response from the called processor's run-time system. This approach is preferred if the remote reference can be performed in an amount of time less than it takes to perform a context switch to another task. In the case of the hypercube, it may be necessary to employ this strategy in only some of the remote references. The decision of whether or not to schedule another task is based on the number of hops away the referenced variable is, since this determines the lower bound on communication time.

5 Status and Conclusions

The current implementation of the Ada run-time system on the NCUBE hypercube contains support for task scheduling, rendezvous, delays, and synchronized activation and termination. Initially, a general approach was taken and the message passing schemes discussed above were used in support of features executed within one processor as well as across processor boundaries. This approach will simplify the transition to incorporate the capabilities described in Sec. 4.3 above, but may cause unnecessary overhead given the current restrictions. In some cases, the run-time system code will be modified to take advantage of these restrictions. The two approaches may then be compared through performance measurement, using the techniques and algorithms given in [CDV86]. More details regarding the actual implementation and an example of its use may be found in [CMV87].

Several conclusions can be made about parallel languages on distributed memory multiprocessors. These range from the programmability of such machines to the structure and implementation of the run-time environment. Based on the above discussion, we can make the following points:

- Parallel languages improve the multiprocessor programming environment. Such languages allow multiprocessors to be coded with a single program that provides abstraction through high level structures. The benefits of this approach include strong type checking, multitasking, and the opportunity to create coherent parallel programs. An additional benefit of parallel languages is the reusability of machine independent concurrent software that is coded in these languages.

- Specification of the allowable program units of distribution greatly impacts the requirements of the run-time system. Placing some straightforward restrictions on the units of distribution can simplify the duties of the run-time system without unreasonably hampering the programmer or violating the language definition.

- Even when supporting a large number of language features, efficiency can be achieved if support for costly operations does not hamper the implementation of other operations. This is a goal that we wanted to achieve, and it influenced the implementation and choice of units of distribution. Implementing intraprocessor rendezvous through direct queue manipulation instead of message passing and restricting task migration to allow only local cactus stack pointers are instances of this approach.

- Developing run-time support for Ada on a parallel target provides valuable experience for the study of similar parallel languages on multiprocessors. The units of the run-time system were coded in a high level language in our implementation and were organized into well defined modules. They can be easily modified to provide specific support for other languages that have similar models of concurrency, e.g., Concurrent C.

References

[BaR85] Baker, T.P. and G.A. Riccardi, "Ada Tasking: From Semantics to Efficient Implementation," *IEEE Software*, pp. 34-46, March 1985.

[Bar84] Barnes, J.G.P., *Programming in Ada*, Addison-Wesley, Reading, Mass., 1984.

[BiN84] Birrell, A.D. and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, February 1984.

[BCG88] Bjornson, R., N. Carriero and D. Gelernter, "Linda on Distributed-Memory Machines," *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[Buz88] Buzzard, G.D. "High Performance Communications on Hypercube Multiprocessors," Ph.D. Thesis, The University of Michigan, (work in progress).

[CDV86] Clapp, R.M., L. Duchesneau, R.A. Volz, T.N. Mudge and T. Schultze, "Toward Real-Time Performance Benchmarks for Ada," *Communications of the ACM*, vol. 29, no. 8, pp. 760-778, August 1986.

[CMV87] Clapp, R.M., T.N. Mudge and R.A. Volz, "Distributed Run-Time Support for Ada on the NCUBE Hypercube Multiprocessor," Technical Report RSD-TR-10-87, Robotics Research Laboratory, The University of Michigan, August 1987.

[ClM88] Clapp, R.M. and T.N. Mudge, "Distributed Ada on a Loosely Coupled Multiprocessor," Technical Report RSD-TR-3-88, Robotics Research Laboratory, The University of Michigan, January 1988.

[Dal88] Dally, W.J., "Object-Oriented Concurrent Programming in CST," *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[FMO88] Felten, E.W., R. Morison and S.W. Otto, "Coherent Parallel C," *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[FiW86] Fisher, D.A. and R.M. Weatherly, "Issues in the Design of a Distributed Operating System for Ada," *IEEE Computer*, pp. 38-47, May 1986.

[FSS87] Flynn, S., E. Schonberg and E. Schonberg, "The Efficient Termination of Ada Tasks in a Multiprocessor Environment," *ACM Ada Letters*, vol. 7, no. 7, pp. 55-75, November/December 1987.

[LeB82] LeBlanc, T.J., "The Design and Performance of High-Level Language Primitives for Distributed Programming," Ph.D. Thesis, TR-492, Computer Science Department, University of Wisconsin, December 1982.

[LRM83] *Ada Programming Language (ANSI-MIL-STD-1815A)*. Washington, D.C. 20301: Ada Joint Program Office, Department of Defense, OUSD (R&D), January 1983.

[Ree88] Reeves, A.P., "Programming Environments for Highly Parallel Multiprocessors," *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[Ros87] Rosenblum, D.S., "An Efficient Communication Kernel for Distributed Ada Run-Time Tasking Supervisors," *ACM Ada Letters*, vol. 7, no. 2, pp. 102-117, March/April 1987.

[Sha88] Shapiro, E. "In Search of a Base Language for Parallel Computers," *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[VMB88] Volz, R.A., T.N. Mudge, G.D. Buzzard and P. Krishnan, "Translation and Execution of Distributed Ada Programs: Is It Still Ada," *IEEE Transactions on Software Engineering*, (to appear).

[VoM87a] Volz, R.A. and T.N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 449-459, April 1987.

[VoM87b] Volz, R.A. and T.N. Mudge, "Instruction Level Mechanisms for Accurate Real-Time Task Scheduling," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 988-992, August 1987.

[Wea84] Weatherly, R.M., "A Message-Based Kernel to Support Ada Tasking," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pp. 136-144, October 1984.