The Charm (4.5) Programming Language Manual

April 18, 1996

The Charm software was developed as a group effort. The earliest prototype, Chare Kernel(1.0), was developed by Wennie Shu and Kevin Nomura working with Laxmikant Kale. The second prototype, Chare Kernel(2.0), a complete re-write with major design changes, was developed by a team consisting of Wayne Fenton, Balkrishna Ramkumar, Vikram Saletore, Amitabh B. Sinha and Laxmikant Kale. The translator for Chare Kernel(2.0) was written by Manish Gupta.

Charm(3.0), with significant design changes, was developed by a team consisting of Attila Gursoy, Balkrishna Ramkumar, Amitabh B. Sinha and Laxmikant Kale, with a new translator written by Nimish Shah. Version 4.0 was developed by Attila Gursoy, Sanjeev Krishnan, Amitabh B. Sinha and Laxmikant Kale. Version 4.3 with the addition various libararies developed by Attila Gursoy, Sanjeev Krishnan, Robert Neely, Amitabh B. Sinha and Laxmikant Kale. Version 4.5, which is based on Converse runtime system, was developed by a team of Milind A. Bhandarkar, Robert Brunner, Attila Gursoy, Sanjeev Krishnan, Joshua M. Yelon, and Laxmikant Kale.

Contents

1 Introduction

Charm is a parallel programming system that supports an explicitly parallel C-based language. It was designed with two major objectives:

- To allow machine independent parallel programming over the class of MIMD machines, without loss of efficiency.
- To help control the complexity of parallel programming.

The language is suitable for programming both shared and nonshared memory machines. Charm creates a division of labor between the programmer and the system, wherein the programmer is responsible for the creation of parallel sub-computations, while the system decides when and where to execute them. In addition, the language provides machine-independent abstractions and features (specifically those for information sharing), which are implemented differently, but efficiently on different types of machines.

Care has been taken to ensure that the abstractions are not overly specialized or narrow. In this sense, Charm represents the lowest level at which it is possible to abstract over resource management and machine-dependent expression. In addition to serving as a direct application programming language, Charm can serve as a "back-end" for other explicitly or implicitly parallel higher-level languages, parallelizing compilers, and domain-specific application packages.

Charm runs on a variety of parallel machines. *Installation manual* lists all the machines supported with this release. However, keep in mind that we port charm to new architectures regularly, indeed, somewhat more often than we update this manual. If you need a current list, check the ftp site `ftp://a.cs.uiuc.edu/pub/CHARM` and the web page `http://charm.cs.uiuc.edu/`.

This manual describes the Charm programming language and its installation and usage procedure. For more information about related publications and how to obtain Charm, see Section ??.

1.1 Charm Execution Model

The basic unit of parallel computation in Charm programs is a *chare*[1]. A chare is similar to a process, a concurrent object, an actor, an ADA task, etc. Note that there is a distinction between the definition of a chare and an instance of a chare. A chare definition merely specifies the behavior of a class of possible chare instances. A chare instance is created by the **CreateChare** system call.

The execution of a Charm program begins at the **CharmInit** entry point of the **main** chare. The **CharmInit** entry point is used to read in input, to create new chares, and to initialize branch office chares and specifically shared variables, e.g., read only variables, accumulators, monotonics and dynamic tables. The parallel program execution begins (conceptually) after this entry point is executed.

Calls to create chares are allowed in the **CharmInit** entry point. At runtime, many chare instances may be active. Chare instances may send messages to one another using the **SendMsg** system call. Conceptually, you may imagine that the system maintains a "work-pool" consisting of "seeds"

[1]Chare (pronounced **chär**, ä as in cart) is Old English for chore, and is borrowed from Robert Keller and the REDIFLOW project

for new chares (which are created when **CreateChare** calls are made), and messages for existing chares (which are created when **SendMsg** calls are made). The system may pick multiple items, non-deterministically, from this pool and execute them. It will not process two messages for the same chare instance concurrently, otherwise, it is free to schedule them in any way (but see also Section ?? on priorities, which can influence the scheduler's decisions).

Executing a message for an existing chare involves retrieving its data-area (which has its local variables) and executing the entry-point code specified in the **SendMsg** call, after setting the message-variable of the entry-point to point to the message. Executing a seed for a new chare involves allocating the data area for the chare, and beginning execution at the entry-point specified in the **CreateChare** call. When execution begins at the beginning of an entry-point, it does not block until it finishes executing the code at that entry point.

In addition to the two calls mentioned above (and their variations), Charm provides many other basic calls, which are described in this document. These calls include support for handling flow of control, memory management and I/O.

Section ?? gives references to other publications that discuss the Charm execution model in more detail.

1.2 Syntax of a Charm program

A Charm program is structurally similar to a C program. Many components of a Charm program *are* C code fragments. **The C code cannot, however, contain global or static variables.** Figure ?? shows the syntax of a Charm program.

A Charm program consists of modules. Each module is defined in a separate file. A module consists of a declaration section and a code section. In the declaration section, there are message and information sharing definitions, in addition to the definition of constants and inclusion of files. An example of a message definition appears in Figure ??. The message type defined in the figure can be referred to either as MSG_NAME or **message** msg_name.

The code section consists of definitions of chares, branch-office chares and C functions. A chare definition starts with the name of the chare, a declaration of local variables of the chare, entry-point definitions and definition of functions **private** to the chare. Entry points and **private** functions (and **public** functions, introduced in Section ??) can access the local variables of the chare. Each entry point definition includes the declaration of a message variable and an arbitrary C-code-block. The C-code-block may contain Charm calls in addition to the usual C code. The syntax of branch office chares and information sharing abstractions is discussed in Section ?? and Section ??, respectively.

1.3 Structure of this Manual

This manual describes the various Charm calls and how to run Charm programs. Basic Charm calls are described in Section ??. Charm provides specific information sharing abstractions. The syntax of these abstractions and the related system calls are described in Section ??. In addition to chares, Charm provides another type of process called branch office chare, which is essentially a chare replicated on each processor. In Section ??, we describe the syntax and system calls related to branch office chares. In nonshared memory machines, pointers inside messages (if used) must

```
module Module1 {

    Type declarations
    Message definitions, information sharing declarations

    chare main {
        Local variable declarations

        entry CharmInit:  C-code-block
        Other Entry points, functions and their code
    }

    chare Example1 {
        Local variable declarations

        /* Entry Point Definitions */

        entry EP1:  (message MESSAGE_TYPE1 *msgPtr)
            C-code-block
        ..
        entry EPn:  (message MESSAGE_TYPEn *msgPtr)
            C-code-block

        /* Local Function Definitions */

        private Function1(..)
            C-code-block
        ..
        private Functionm(..)
            C-code-block
    }

    BranchOffice Chares and other function declarations

}
```

Figure 1: Syntax of a Charm program

```
message msg_name {
    /* declaration of variables inside */
    /* the message structure.  */
    int x;
    float y[100];
} MSG_NAME;
```

Figure 2: Example of a Message Definition

be "packed" before the message can be sent across address-space boundaries. Calls for *message packing* are described in Section ??. Prioritized execution of messages is described in Section ??. We discuss the syntax of Charm programs with multiple modules in Section ??. Commonly used Charm features are now being provided as libraries. In Section ??, we describe the libraries currently available. Section ?? describes Dagger, a coordination language (running on top of Charm) that allows dependences between messages and computations to be expressed.

Finally, information about how to install Charm, how to compile and run programs on various machines are summarized in Section ??. Charm allows the user to choose from alternative compile-time and run-time options for queuing, load balancing, and performance analysis. These options are described in the same section. The complete description of installing Charm, compiling and running Charm programs is given separately in the *Installation Manual.*

2 Basic Charm Calls

This section contains the basic system calls in Charm. These calls can be used to create new chares, send messages to existing chares, and allocate and free memory. Various other calls for input-output and exiting from Charm programs are also described in this section. The system types used in the following Charm call descriptions are defined by the Charm system. The definitions may change in future versions of Charm.

2.1 Basic Primitives

2.1.1 Creating Chares

The most basic call is **CreateChare.**

CreateChare (charename, entry, msgptr, [virtualID [, destPE]])
identifier charename;
EntryPointType entry;
void * msgptr;
ChareIDType *virtualID;
PeNumType destPE;

CreateChare creates a new instance of a chare. (See Section ?? for a discussion of a special form of chare that is replicated on every processor, the *branch-office chare.*) The call can have three, four or five parameters as shown above. The parameters are described below.

charename: The name of the chare that is to be created.

entry: The entry point in the chare to which the message pointed to by msgptr will be sent, and where execution in this chare will start. The value expected for entry is a valid entry point in charename and may assume one of several possible forms:

1. The literal name of an entry point, possibly prefixed by a chare name. If the entry point belongs to another chare, the chare name prefix is required. The separator between the chare name prefix and the entry name is the @ character, as in *charename@entryname.*
2. A variable of type **EntryPointType**, to which an entry point has been assigned. For example,

```
x = charename@entryname;
 :
CreateChare(charename, x, ....)
```

msgptr: A pointer to the initial message to be sent to the created chare. The type of the message should match the message type associated with entry point entry.

virtualID: For certain applications, it is necessary for the user to have the **CreateChare** call return a virtual ID, virtualID, which is useful for sending messages to the chare, even if it has not yet been created on any processor. When messages are sent to the ID virtualID, they are

either queued up to be sent to the chare when it is created, or simply redirected if the chare has already been created. The **CreateChare** call fills the virtualID in place. If desired, the virtualID may then be sent to other chares via the **SendMsg** system call. If no virtual ID is required this may be indicated by setting the virtualID parameter to **NULL_VID**. It should be noted that additional overhead is incurred by the virtual ID feature.

destPE: When a chare is to be created at a specific processor, destPE is used to specify that processor. Note that, in general, for good load balancing, the user should let Charm determine the processor on which to create a chare. Under unusual circumstances however, the user may want to choose the destination processor. If a process replicated on every processor is desired, then one should instead use the branch office chare abstraction (described in Section **??**).

2.1.2 Sending Messages

Chares can communicate with each other by using the **SendMsg** call.

SendMsg(entry, msgptr, chareid)
EntryPointType entry;
void *msgptr;
ChareIDType *chareid;

SendMsg sends the specified message (allocated by **CkAllocMsg**–see Section **??**) to the specified chare at the specified entry point.

entry: The entry point in the chare to which the message will be sent and where execution is to start. entry must be a valid entry point name. See the discussion of the entry parameter for the **CreateChare** command above for details.

msgptr: A pointer to the message to be sent. Its type must match that expected by the entry point as specified in the entry point definition.

chareid: The ID of the chare to which the message at msgptr is to be sent. For a chare instance to obtain its own ID, it can access the implicitly-present local variable **ThisChareID**. For a chare instance I1 to obtain the ID of another chare instance I2, it needs I2's cooperation: I2 must obtain its own ID, then must somehow transmit it to I1. This process is analogous to the passing of object-pointers in C.

2.2 Memory Allocation

The following system calls are used to allocate and deallocate memory. **Important note:** The memory allocation calls for messages and other data items are distinct. **CkAlloc** must not be used to allocate messages, and **CkAllocMsg** should not be used for non-message data items.

2.2.1 Ordinary Memory Allocation

void *CkAlloc(size)
int size;

The **CkAlloc** call allocates memory and returns a pointer to the allocated space, if successful. If unsuccessful, it returns NULL. **CkAlloc** attempts to take advantage of allocation/deallocation patterns that commonly arise in parallel programs and should be used in place of **malloc**.

size: The number of bytes to allocate.

2.2.2 Memory Allocation for Messages

void *CkAllocMsg(msg [, sizes_array])
MSG_TYPE msg;
int * sizes_array;

The **CkAllocMsg** call allocates space for a message. If successful, a pointer is returned to the allocated space, otherwise NULL is returned. Any data to be used as a message and passed to system routines **must** be allocated using **CkAllocMsg**, not **CkAlloc**. (Note: see also **CkAllocPrioMsg** in section **??**.)

msg: A pointer to the message being allocated. The Charm system uses the size of the MSG_TYPE message type to determine how many bytes to allocate.

sizes_array: An optional array parameter used only for variable sized messages to specify the size of the message. See Section **??** below for further discussion of variable sized messages.

2.2.3 Variable Sized Messages

In Section **??** we explain how messages can contain arbitrary pointers, and how the validity of such pointers can be maintained across processors in a distributed memory machine. However, we also provide the user a construct called **varSize** to declare messages with (one-dimensional, non-pointer) arrays whose sizes may be known only when the message is allocated. The translator generates code to handle the pointers (so as to maintain validity) for such messages. The optional parameter sizes_array in the **CkAllocMsg** call is used to specify variable sized messages. An example of usage of a variable sized message appears in Figure **??**.

In the example in Figure **??**, a message with two variable sized arrays has been declared. The sizes array is used to specify the sizes (i.e., number of elements) of the arrays in the order in which they appear in the message declaration. Notice that the user must maintain the sizes of the arrays (as done with size1 and size2 fields in this example) if such information is needed at a later stage.

```
message {
    int size1, size2;
    varSize float field1[];
    varSize float field2[];
} MSG;
⋮
int sizes[2];
MSG * msg;

sizes[0] = size1;
sizes[1] = size2;
msg = (MSG *) CkAllocMsg(MSG, sizes);
msg->size1 = size1;
msg->size2 = size2;
for (i=0; i<size1; i++)
    msg->field1[i] = i;
for (j=0; j<size2; j++)
    msg->field2[j] = j;
```

Figure 3: Use of a variable sized message

2.2.4 Copying Messages

void *CkCopyMsg(msg)
MSG_TYPE *msg;

This system call allows the user to make an exact copy of an existing message specified by the parameter. It returns a pointer to a new copy of that message. Space for the new message is allocated by **CkAllocMsg**. The return value is a pointer of the same message type as the parameter being passed.

There are two situations in which this function is particularly useful:

1. When a **SendMsg** is inside a loop, the message must be reallocated and filled each time through the loop.
2. When the user does not know the structure of the message, but needs to allocate and fill a copy to be sent.

msg: A pointer to the message being copied.

2.2.5 Deallocating Memory

void CkFree(ptr)
void *ptr;

The call **CkFree** frees up the space allocated by a previous **CkAlloc**. The user may not split an allocated block and free it in parts.

ptr: A pointer to the space to be freed. This pointer must have been obtained via a previous call to **CkAlloc**.

void CkFreeMsg(msgPtr)
void *msgPtr;

The **CkFreeMsg** call frees up the space associated with a message. This function must be used (rather than **CkFree**) when the item to be freed is a message.

msgPtr: A pointer to the space to be freed. This pointer must have been obtained via a previous call to **CkAllocMsg** or **CkAllocPrioMsg**.

2.3 Input/Output Primitives

int CkPrintf(format [, arg]*)
char *format;

CkPrintf places output on the standard output stream. The output format specified by format works exactly as in the **printf** statement in C.

int CkScanf(format [, arg]*)
char *format;

CkScanf reads input from the standard input stream. The input format specified by format works exactly as in the **scanf** statement in C. Currently at most 16 arguments can be specified.

Other C primitives such as **fscanf and fprintf** functions may be used as in C.

2.4 Quiescence Detection

Charm provides the user with the ability to detect quiescence in the user computation. Quiescence is defined as the state in which there are no chares executing and no messages in transit or in queues (and the system has not been requested to exit). Quiescence detection is initiated by calling

the function **StartQuiescence**, which asks the system to begin detecting quiescence. If and when quiescence is detected, a message is sent to a designated entry point of a designated chare. Note that the user can have the quiescence message reported to multiple chares and entry points by calling **StartQuiescence** multiple times (once for each chare-entry point to be notified). Care must be exercised if this is done however, because only the *initiation of the sending* of the messages is done synchronously. That is, on only *one* processor is the quiescence message *necessarily* the first message processed after quiescence has been detected. That processor could then send other messages that are processed before the quiescence message on other processors.

Note also that quiescence detection can be initiated for more than one phase of computation (i.e. once quiescence is detected, quiescence detection may be "restarted").

void StartQuiescence(entry, chareid)
EntryPointType entry;
ChareIDType *chareid;

StartQuiescence causes a message to be sent to the specified entry point when quiescence is detected. The message type QUIESCENCE_MSG is defined defined by the Charm system in the header file **"qd.h"** which must be included in any files in which QUIESCENCE_MSG is used.

ep: The entry point to which the message will be sent when quiescence is detected.

chareid: The ID of the chare to be notified when quiescence is detected.

2.5 Charm Exit Calls

void ChareExit()

The **ChareExit** call is used to inform the system to release any memory allocated for the local variables in the chare *after the execution of the current entry point is finished.* The entry point code must explicitly relinquish control. At present we do not handle messages sent to chares that have exited by making a call to **ChareExit**; these messages constitute runtime error, and may lead to unpredictable behavior.

void CkExit()

Similar to the regular Unix **exit()**, except that this routine will kill all running processes and then exit itself. The **CkExit** routine does not stop execution of the code at the calling entry point immediately. It simply tells the system that after the current entry point finishes execution, it should take care of any housekeeping activities required, and ask all other running processes to terminate (again, non-preemptively).

2.6 Function Calls

In Charm, addresses of functions can be passed as parameters to function calls, or in a message to an entry point in a chare. In distributed memory machines, where messages can span address boundaries (across processors), a message containing a function address may have a valid address only on the processor where the address was first determined. Charm provides a mechanism by which one can pass a reference index to a function, instead of the function name, and accesses to the function can be made through the reference index. The necessary functions are **FunctionNameToRef** and **FunctionRefToName**. They are described below.

FunctionRefType FunctionNameToRef(function-name)
identifier function-name;

The system call **FunctionNameToRef** takes a function name as parameter and returns a reference index. This reference index may be passed in a message to other chares on other processors. The index can be dereferenced through the **FunctionRefToName** system call to get the local address (on the calling processor) of the function, function-name.

function-name: The function name for which a reference index will be computed.

typedef int (*FUNCTION_PTR)();

FUNCTION_PTR FunctionRefToName(function-ref)
FunctionRefType function-ref;

The system call **FunctionRefToName** takes a function reference as parameter and returns a pointer to the function corresponding to that reference index.

function-ref: The function reference for which a function pointer will be computed.

2.7 Other Calls

PrivateCall(FunctionName([arg [,arg]*]))

This call allows one to call a C subroutine *FunctionName* declared as a **private** function to the chare definition. The call may only be made from an entry point in the chare or from another private function defined in that chare, enabling a form of information hiding. The other advantage of this call is that private functions can directly access variables defined for that chare as though it were in the same scope. This eliminates the need for passing a chare's local data as parameters to private functions. The call returns whatever FunctionName returns.

int CkTimer()

The **CkTimer** call is used to time programs. **CkTimer** return the time since the process began execution on that node. in milliseconds. **CkTimer** is based on the Converse timer call *double CmiTimer()* which returns elapsed time in seconds.

int CkNumPes()

CkNumPes returns the total number of processors on which the user program is being run.

int CkMyPe()

CkMyPe returns the processor number of the processor on which the **CkMyPe** call was made. The numbers assigned are in the range 0..**CkNumPes()**-1, and are not necessarily identical to processor numbers assigned by a specific parallel machine.

3 Branch Office Chares

A *branch office chare* is a special type of chare–it has a representative branch process on each node.

3.1 Syntax of a Branch Office Chare

The syntax of a branch office chare appears in Figure ??. The syntax of a branch office chare is almost identical to that a chare, except that it may have **public** functions, which can be called from outside the branch office chare.

```
BranchOffice <branch-office-chare-name> {

      Local Variables

      entry EPB1 : (message MESSAGE_TYPE1 *msg)
            C-code-block
      ..
      entry EPBY : (message MESSAGE_TYPEy *msg)
            C-code-block

      [private|public] FunctionB1(..)
            C-code-block
      ..
      [private|public] FunctionBZ(..)
            C-code-block
}
```

Figure 4: Syntax of a Branch Office Chare

3.2 Basic System Calls for Branch Office Chares

A branch office chare can be created using the **CreateBoc** system call. The syntax of the call follows:

```
ChareNumType CreateBoc (bocname, entry, msg[,ReturnEP, ReturnID])
      identifier bocname;
      EntryPointType entry;
      char *msg;
      EntryPointType ReturnEP;
      ChareIDType * ReturnID;
```

The **CreateBoc** call creates a branch process for a branch office chare on each node. The call

results in a user defined message, msg, being sent to the entry point, entry, in the branch office chare. The **CreateBoc** call returns an identifier of type ChareNumType, if called from within the **CharmInit** section of the main chare. At any other point the **CreateBoc** call may return an illegal identifier. Outside the **CharmInit** entry point, the identifier may be received in a message at the specified entry point, ReturnEP, and address, ReturnID. This user-defined message can have any structure, but its first element must be of type ChareNumType.

Communication between branch processes of various branch office chares are carried out through the two message passing primitives: **SendMsgBranch** and **BroadcastMsgBranch**. These calls are described below.

SendMsgBranch (entry, msg, penum [, boc])
EntryPointType entry;
void * msg;
PeNumType penum;
ChareNumType boc;

The **SendMsgBranch** call is used to send a user defined message msg to the entry point entry in the branch process on node penum. If the optional parameter is specified, then the message is sent to the branch office chare identified by boc, otherwise it is sent to the branch of calling branch office chare.

BroadcastMsgBranch(entry, msg [, boc])
EntryPointType entry;
void *msg;
ChareNumType boc;

The **BroadcastMsgBranch** call is used to send a user defined message msg to all the branches at the entry point entry. If the optional parameter is specified, then the message is sent to the branch office chare identified by boc, otherwise it is sent to the calling branch office chare.

The system call **PrivateCall** can also be used from inside a branch office chare to call a function declared to be of the type **private function.** The syntax is the same as that of the **PrivateCall** for chares (see Section ??).

PrivateCall(fn([arg [, arg]*]))

The system call **BranchCall** is used by chares to call (branch office) functions on the same node as the chare. The syntax of the call is as follows:

BranchCall (boc, bocname@fn([arg [, arg]*]))
ChareNumType boc;

The **BranchCall** call is used to call a **public** function bocname@fn in a branch process on the same node as the process that made the call. **BranchCall** can also be made by passing a pointer to the public function fn in bocname. The branch process belongs to the branch office chare named

bocname identified by boc. A **BranchCall** returns whatever the function *bocname* is supposed to return.

ChareNumType MyBocNum()

The **MyBocNum** call returns the branch office chare number for the currently executing branch office chare. It should be called *only* from within a branch office chare.

MyBranchID(pChareID)
ChareIDType *pChareID;

The call **MyBranchID** takes a pointer to a pre-allocated data area of type ChareIDType and copies in the identity of the corresponding branch of the branch office chare. The identity of the branch can be used to send messages to an entry point in it using the **SendMsg** system call. Its useful when messages need to be sent to a branch office chare from a chare, when the **SendMsgBranch** call cannot be used, e.g., the user can pass the address of a branch in the **Find** request made from inside a branch office chare so that the message is sent back to an entry point inside the branch office chare.

3.3 Spanning Tree System Calls

Charm defines a spanning tree organization of the processor nodes along with routines for accessing elements of that tree. The spanning tree routines come in handy when writing programs with branch office chares because message communication patterns can be made to proceed along the spanning tree arcs to avoid bottlenecks at a single node.

int CkSpanTreeRoot()

The **CkSpanTreeRoot** returns the processor number of the root of the spanning tree.

int CkSpanTreeParent(node)
int node;

The **CkSpanTreeParent** call returns the processor number of the parent in the spanning tree for node node.

CkSpanTreeChild(node, children)
int node, *children;

CkSpanTreeChild is used to determine the children in the spanning tree for a node node.

int CkNumSpanTreeChildren(node)
int node;

The **CkNumSpanTreeChildren** routine is used to determine the number of spanning tree children of node **node**.

4 Information Sharing Abstractions

Charm is a parallel programming system that allows users to write programs that run on both shared-memory and message-passing machines. Rather than provide a generalised mechanism for information sharing between chares, Charm provides a variety of specific abstract data types which the users can choose from, depending on the type of data that is needed by the program. These are referred to as *specifically shared variables*, and they provide as efficient an implementation for sharing data as the underlying machine can provide.

4.1 Read Only Variables and Messages

Read only variables and messages are used to share information that is obtained only after the program begins execution, and which does not change subsequently. The **readonly** declarations are in the declaration section of the program, and have the following form:

readonly Type readname;
readonly MsgType *readmsgname;

In the first declaration, the variable readname is declared to be a read only variable of type Type. In the second declaration, the variable readmsgname is declared to be read only message of type MsgType.

A variable or message that has been declared **readonly** needs to be assigned the values it is going to hold for the rest of the program and then initialised in the **CharmInit** entry point of the main chare. The initialization of a read only variable is done by the **ReadInit** system call.

ReadInit (readname)
Type readname;

The above call creates the read only variable readname. The read only variable is initialized with its value at the time the **ReadInit** call was made. No further writes can be made to the variable readname after the **ReadInit** call is made.

Similarly, a read only message is initialized by the **ReadMsgInit** system call.

ReadMsgInit (readmsgname)
MsgType * readmsgname;

Prior to the **ReadMsgInit** call, the user has to allocate the message readmsgname, and make the appropriate assignments to the message fields. Again, all this is to be done in the **CharmInit** entry point of the main chare.

The call establishes the messages as a read only variable on all the processors. The value of a read only variable or a readonly message can be read at a later stage by invoking the **ReadValue** system call.

ReadValue (readname)
Type readname;

The **ReadValue** call is used to obtain the value of the read only variable readname.

In the example in Figure ??, count is declared to be an integer read only variable. It is initialized using the **ReadInit** call to have a value 10. The message MSG is declared as a read only message, and is initialized using the **ReadMsgInit** call.

```
message {
    int x;
} MSG;

readonly int count;
readonly MSG *msg;
..
int n = 5;
count = 10;
ReadInit(count);
msg = (MSG *) CkAllocMsg(MSG);
msg->x = n;
ReadMsgInit(msg);
CkPrintf("%d", ReadValue(count));
CkPrintf("%d", ReadValue(msg->x));
```

Figure 5: Example of a readonly variable and message

The **ReadValue(x)** call can be thought of as a variable with the same type as x, and having the value of the read only variable. Thus the **ReadValue(count)** call can be used in expressions as a variable of type **int** with the value of the read only variable count,
e.g., n += 10 × ReadValue(count).

4.2 Accumulator Variables

An accumulator variable is like a counter – it's value increases monotonically in a fixed metric, and is needed only once (perhaps, at the end of computation). It's useful, for example, in counting the number of nodes searched in a search problem, where the count is an index of the difficulty of the search.

The **accumulator** abstract data type has associated with it a message containing the data area of the accumulator data type, an initialization function (that is used to set up the message) and two user defined commutative-associative operators.

The initialization function is called on every node (upon the invocation of the **CreateAcc** call).

The first operator *adds* to the accumulator variable in some user defined fashion, while maintaining commutativity and associativity, and is invoked when the user wants to add to the accumulator (using the **Accumulate** call).

The second operator *combines* two accumulator variables element by element, again in a commutative-

associative manner. This is used when the final value of the accumulator is being determined (as a result of the **CollectValue** call).

```
accumulator acc_type {
    Message_Type *acc;
    Message_Type *initfn (void * msg)
        C-code-block
    addfn ([arg[,arg]*])
        C-code-block
    combinefn (Message_Type * acc2)
        C-code-block
} ACC_TYPE;
```

Figure 6: Structure of an accumulator abstract data type

The structure of an accumulator abstract data type is shown in Figure ??. In this declaration, the accumulator is of type `ACC_TYPE`, the associated message is of type `Message_Type`, the initialization function is `initfn` and the two operators are `addfn` and `combinefn`. The names of the functions, `initfn`, `addfn` and `combinefn`, are keywords, and serve to identify the nature of the functions. Currently , their order cannot be changed. The scope of these functions is the block defined by the accumulator variable. An accumulator variable is initialized using the **CreateAcc** call.

typedef INT16 AccIDType;

AccIDType CreateAcc (ACC_TYPE, msg [, ReturnEP, ReturnID])
char * msg;
EntryPointType ReturnEP;
ChareIDType *ReturnID;

The **CreateAcc** call is used to create an accumulator of type ACC_TYPE. The call results in the `initfn`, defined in the accumulator, being invoked on every node with a single parameter msg, where msg is a message of type Message_Type. `initfn` must allocate space for the message for that accumulator type, initialize the contents of the message for the metric that the accumulator is defined over, and finally return a pointer to the created message. The **CreateAcc** call returns a value of type AccIDType if the call has been made inside the **CharmInit** entry point of the **main** chare. Otherwise optional parameters can be used to receive the identifier at the entry point ReturnEP at address ReturnID. The message received can be defined by the user to have any structure, except the first field must be of type AccIDType.

A note of caution: The value returned by a call to **CreateAcc** in any place other than **CharmInit** may not be a valid accumulator index.

Any chare can add to an accumulator variable by making the **Accumulate** call.

Accumulate (accid, addfn([arg [, arg]*]))
AccIDType accid;

The **Accumulate** call is used to call addfn, defined for that accumulator type, to add a value to the accumulator identified by accid (care should be taken to ensure that the accumulator index accid is legal at the time the call is made).

The correct value of an accumulator can be collected by using the **CollectValue** call.

CollectValue (accid, entry, chareid)
AccIDType accid;
EntryPointType entry;
ChareIDType * chareid;

The **CollectValue** call is used to collect the value of the accumulator identified by accid, and send the message, associated with that accumulator, to the chare chareid at entry point entry. The call can be used only once for each accumulator variable; later usages will result in an run time error.

In Figure ??, A1 is declared to be an accumulator, which is an array of integers. The two operators are addfn and combinefn; addfn adds to elements inside the accumulator, and combinefn takes an accumulator variable as a parameter and combines it (element by element) with the current value of the accumulator.

4.3 Monotonic Variables

A monotonic variable is similar to an accumulator in the sense that its value also increases monotonically in some metric. However, it differs from an accumulator variable in that its value may be accessed frequently during the course of the computation. Monotonic variables are especially useful in branch-and-bound computations. In this case every chare needs to know the current best bound, and this bound might change if some chare found a *better* bound. Repeated supply of the same bound would have the same effect as supplying it once .

A **monotonic** abstract data type has associated with it a message containing the data area of the monotonic data type, an initialization function (that is used to set up the message) and an update operator, which is *idempotent, commutative, associative* and *monotonic.*

In Figure ??, a monotonic data type of type MONO_TYPE is declared. The message msg associated with the monotonic variable is of type Message_Type. The initialization function for the monotonic variable is initfn, and updatefn is the idempotent, monotonic, commutative and associative operator. The names of the functions, initfn and updatefn, are keywords, and the scope of the functions is inside the block defined by the declaration of the monotonic variable.

A variable of monotonic data type can be created using the **CreateMono** call.

```
message {
    int a[MAX];
} MSG1;

accumulator a1 {
    MSG1 *msg;
    MSG1 * initfn (data)
    void *data;
    {
        int i;
        msg = (MSG1 *) CkAllocMsg(MSG1);
        for (i=0; i<MAX; ++)
            msg->a[i] = 0;
        return(msg);
    }
    addfn (y, z)
    int y; int z;
    {
        msg->a[y] += z;
    }
    combinefn (b)
    MSG1 *b;
    {
        int i;
        for (i=0 ; i<MAX; i++)
            msg->a[i] += b->a[i];
    }
} A1;
```

Figure 7: Example of an accumulator variable

```
monotonic mono_type {
    Message_Type *msg;
    Message_Type *initfn (void * a)
        C-code-block
    updatefn (Message_Type *msg2)
        C-code-block
} MONO_TYPE;
```

Figure 8: Structure of a monotonic abstract data type

typedef INT16 MonoIDType;

MonoIDType CreateMono (monotype, msg [, ReturnEP, ReturnID])
MONO_TYPE monotype;
char * msg;
EntryPointType ReturnEP;
ChareIDType *ReturnID;

The **CreateMono** call is used to create a monotonic variable of type MONO_TYPE. The call results in `initfn`, defined in the monotonic data type, being invoked on every node with a single parameter msg. `initfn` must allocate space for the message for that monotonic type, initialize the contents of the message, and finally return a pointer to the created message. The **CreateMono** call has to be made inside the **CharmInit** entry point of the **main** chare and usually returns an identifier of type MonoIDType. Alternatively, optional parameters can be used to receive the identifier at the entry point ReturnEP at address ReturnID. The message received can have any structure, but its first field must be of type **MonoIDType**.

A note of caution: The value returned by the call made in any place other than **CharmInit** may not be a valid monotonic data type index.

The value of a monotonic variable can be updated using the **NewValue** call.

NewValue (monoid, updatefn(arg))
MonoIDType monoid;
Message_Type arg;

The **NewValue** call is used to update the value of the monotonic variable identified by monoid (care must be taken to ensure that the identifier monoid is legal at the time the call is made) with arg by using the update function, defined for that monotonic data type. Notice that arg has to be of the same type as the message in the declaration for this monotonic type.

The value of the monotonic variable can be read using the **MonoValue** declaration.

Message_Type MonoValue (monoid)
MonoIDType monoid;

The **MonoValue** call is used to obtain the value of the monotonic variable indexed by monoid (care must be taken to ensure that the identifier monoid is legal at the time the call is made).

Note : The call may not return the current value of the monotonic variable.

Figure **??** contains the declaration of a monotonic data type, `M1`.

4.4 WriteOnce Variables

Write once variables are similar to read only variables, except that they are dynamically created. The system call that creates a write once variable is **WriteOnce**.

```
message {
    int x;
} MSG2;
monotonic m1 {
    MSG2 *msg;
    MSG2 * initfn (data)
    void *data;
    {
        msg = (MSG2 *) CkAllocMsg(MSG2);
        msg->x = (int) data;
        return(msg);
    }
    updatefn (update)
    MSG2 *update;
    {
        if (msg->x < update->x) {
            msg->x = update->x;
            return(1);
        } else return(0);
    }
} M1;
```

Figure 9: Example of a monotonic variable

typedef INT16 WriteOnceID;

void WriteOnce(dataPtr, dataSize, entry, chareid)
void *dataPtr;
int dataSize;
EntryPointType entry;
ChareIDType chareid;

This routine creates a write once variable. That is, the data given to the routine is distributed to all the other nodes in the system. Then, an identifier denoting the write once variable is passed back to the user at the entry point entry at address chareid. The user may then access the write once variable data on any node using the **DerefWriteOnce** routine and the identifier. Note that it is the user's responsibility to pass the identifier to any chare that wishes to access the write once variable.

The message type declared by the user for receiving the write once variable identifier is irrelevant as long as the first field in the struct is a variable of type WriteOnceID. An example of a write once variable usage appears in Figure **??**.

```
message {
    int dummy;
} SomeMsg;

message {
    WriteOnceID wovID;
} WovMSG;

chare foo {
    entry E1:  (message SomeMsg *dummy) {
        int theVar = 20;
        WriteOnce(&theVar, sizeof(int), foo@E2, &ThisChareID);
    }
    entry E2:  (message WovMsg *wov) {
        int *intPtr;
        /* At this point the write once variable
        has been installed and has been assigned
        a unique id given in wov->wovID */
        intPtr = DerefWriteOnce(wov->wovID);
        CkPrintf("the wov value is [%d]", *intPtr);
    }
}
```

Figure 10: Example of a WriteOnce variable usage

The **DerefWriteOnce** system call allows one to gain access to a write-once variable.

void *DerefWriteOnce(wovID)
WriteOnceID wovID;

The above call, given an identifier denoting a write once variable, returns a pointer to the actual data itself. No updates must be made to a write once variable. (**Note:** This is not currently enforced either by the compiler or the runtime system.) **DerefWriteOnce** simply returns a pointer to the local copy of the variable, so that any changes made to the data itself will be reflected in future accesses made through the DerefWriteOnce routine. Also, any changes will not be broadcast to other nodes, so the other nodes would not see the changes.

4.5 Dynamic Tables

Dynamic tables is an information abstraction that allows chares on different processors to access data in a *location-transparent manner* by using a key which is associated with the data.

A dynamic table is a set of entries, each entry being a record with a key and a data field. The key must be an integer, and the data might be information which is formatted in any manner.

A table `tablename` is declared as follows:

```
#include "tbl.h"
table [{
        hashfn(key)
        int key;
        {
           return (key % CkNumPes()); /* Or some other hash function */
        }
      }] tablename;
```

In the declaration, the programmer may *optionally* include a hash function **hashfn** to indicate the processor which must hold the data with a particular key. In the absence of a hash function, a default hash function is used in distributing the entries.

Access to the entries is provided through non-blocking calls like **TblInsert**, **TblDelete** and **TblFind**. Each of these calls can result in a message of type TBL_MSG being sent to a specified entry point. TBL_MSG has the following structure:

typedef struct {
int key;
char *data;
} **TBL_MSG;**

An access request if said to have a matching entry in the table if the values of the key in an entry and the key supplied by the request match. The matching entry is said to have *defined* data, if the data field of the entry has a non-null value.

The table management calls are described below. In each of the calls a message is sent as a reply only if the parameters entry and chareid are valid values. The user can choose to have no reply messages sent by setting either the value of entry as -1, or the value of chareid as NULL.

TblInsert(tablename, key, data, size_data, entry, chareid)
table tablename;
int key;
char *data;
int size_data;
EntryPointType entry;
ChareIDType *ChareID;

The **TblInsert** call is used to insert the entry with key key and data data of size size_data into the dynamic table tablename. In each of the following cases, a message *msg* of type TBL_MSG is sent back to the entry point entry at chareid, if entry and chareid are valid. The action taken on an **TblInsert** request is as follows:

1. *matching entry already present and data is defined:* The TblInsert request is ignored. A *msg* with the key and data fields set to the values of the existing entry is sent to the entry point.
2. *matching entry already present but data is undefined:* The data field of the matching entry is defined to be data, and the key and data fields of *msg* are key and data, respectively.
3. *matching entry not already present:* An entry is inserted with the key and data fields as key and data. A *msg* is sent immediately with the key and data fields as key and data, respectively.

TblDelete(tablename, key, entry, chareid, option)
table tablename;
KEY_TYPE key;
EntryPointType entry;
ChareIDType *chareid;
int option;

The **TblDelete** call is used to delete the entry with key key from the dynamic table tablename.

If a matching entry is found, a message of type TBL_MSG is sent to the entry point entry at chareid with the key field as key and the data corresponding to that of the deleted entry. (Note that as in the case of **TblInsert**, the message is sent only if entry and chareid are valid.) If a matching entry is not found, a message of type TBL_MSG is sent to the same address, if the option field is TBL_REPLY. The data field in this message is *NULL*, and the key field is key. No message is sent back if option is TBL_NOREPLY.

TblFind(tablename, key, entry, chareid, option)
table tablename;
KEY_TYPE key;
EntryPointType entry;
ChareIDType *chareid;
int option;

The **TblFind** call is used to determine whether or not the entry with key key exists in the dynamic table tablename. In some of the following cases message *msg* of type TBL_MSG is sent to the entry

point entry at chareid if entry and chareid are valid. The values of the key and data fields in *msg* are described below:

1. *a matching entry exists and data is defined:* The key and data fields of *msg* are those of the matching entry.
2. *a matching entry exists but data is NULL:* The following actions are taken depending on the value of option:
   - If option = $TBL\_WAIT\_AFTER\_FIRST$ or option = TBL_ALWAYS_WAIT, then the **TblFind** request is suspended till such time as the data is available, when the message, *msg*, is sent then with key and data fields as key and the available data, respectively.
   - If option = TBL_NEVER_WAIT then the key and data fields of *msg* are key and *NULL*, respectively.
3. *a matching entry did not exist:* A table entry is created for the entry with the key, key, and data as NULL. The following actions are taken depending on the value of option:
   - If option = TBL_ALWAYS_WAIT, then the **TblFind** request is suspended till such time as the data is available. The message, *msg*, is sent then with key and data fields as key and the available data, respectively.
   - If option = TBL_NEVER_WAIT or option = $TBL\_WAIT\_AFTER\_FIRST$ then the key and data fields of *msg* are key and *NULL*, respectively.

   In all the above calls, if data is NULL, then the size of the data must be 0.

5 Message Packing

In many programs the data structure to be passed in messages may be large and complex. As a simple concrete example, consider a dynamically created array being passed in a **CreateChare** message. On a shared memory system, the message need only store a pointer to the base of the array and its size. However, on nonshared memory systems, a pointer is not valid across processors. So the whole array must be copied in each message. A chare may also want to send a dynamically created data structure, such as a graph or a tree. Again, it must be copied (or "packed") into a contiguous structure without pointers before sending it in a message.

Charm encourages a programming style that counters the unpredictability of available work by creating many small chares in the hope of being able to distribute them as needed. So in a nonshared memory version of Charm, many chares that are created are not actually sent out to any other processor, but executed locally. When the system is in saturation (i.e. all the processors have sufficient work), this happens to most chares. This behavior is desirable because it offers the flexibility of responding to load fluctuations as they arise.

In this context, packing every message in a format suitable for across-processor transmission seems quite wasteful. Moreover, such packing is clearly unnecessary and wasteful on shared memory machines. It would be preferable to pack only those messages that actually leave address-space boundaries, leaving other messages free to contain pointers. This poses two problems. First, the programmer doesn't know which one of the created chares (or messages) will end up going to another processor, as load distribution is dynamic and entirely the responsibility of Charm. Second, the Charm cannot pack an application-specific message, since its structure is unknown.

The Charm language provides a solution to this problem by postponing packing until it is absolutely necessary. Messages are allowed to contain pointers as for shared memory machines. However, if the kernel decides to send a message to another processor, it calls appropriate code in the *user* program to pack the message in a contiguous space and eliminate any explicit pointers. Conversely, before a message received from outside is scheduled for execution, the system calls another entry point to unpack the message.

The **pack** and **unpack** functions are associated with the message type. If the message is just a flat record, and does not need any packing, the user does not need to define any **pack** and **unpack** functions. However, if the message contains pointers and dynamically allocated data structures, the user must supply the associated **pack** and **unpack** functions.

When variable-sized arrays (specified as **varSize**) are used in a message, the system automatically generates corresponding **pack** and **unpack** routines. The user would find it difficult to generate the **pack** and **unpack** routines for messages which combine both **varSize** fields with dynamically allocated arrays. Hence it is advised that users not use **varSize** when using other dynamically allocated structures in their messages.

The structure of a message with **pack** and **unpack** routines is as follows:

```
message [message-identifier] {

    <Field Declarations>

    pack pack-identifier (in, out, length)
    char *in;
    char **out;
    int *length;
        <Pack-Function-Body>

    unpack unpack-identifier (in,out)
    char *in;
    char **out;
        <Unpack-Function-Body>
} message-identifier;
```

In the above syntax, the keywords **pack** and **unpack**, identify the pack and unpack routines, respectively, associated with the message. The **pack** routine takes three parameters: `in`, the unpacked message, `out`, the packed message and `length`, the length of the packed message which is set inside the **pack** routine. Inside the **pack** routine the user unravels all the pointers in the incoming message, `in`, and creates a contiguous message area, `out`, with the length `length`. The **unpack** routine takes two parameters: `in`, the packed message, and `out`, the un-packed message. Inside the **unpack** routine the user re-creates the structure with pointers in the unpacked message, `out`, starting with the packed message, `in`.

It is important that the user free the unpacked (original) message in the **pack** routine, but the packed (original) message should not be freed in the **unpack** routine (the system after taking care of various necessary message parameters will free the original message). The **CkAllocPackBuffer** call is used to generate the contiguous message area in which the unpacked message can be packed. The syntax of the **CkAllocPackBuffer** call is as follows:

```
void * CkAllocPackBuffer(msg, size)
        void *msg;
        int size;
```

The **CkAllocPackBuffer** call takes two parameters, msg, which is the unpacked message, and size, which is the size of the allocated pack buffer message. This is the only call that should be used to allocate packed messages – it is important because the call also transfers system relevant information from the unpacked message to the packed message.

An example of a message with **pack** and **unpack** function appears in Figure **??**.

```
message {
    ChareIDType chareid;
    int size;
    int count;
    int *x;
} MSG1;

message [MSG2] {
    int *x;
    int size;
    int count;
    ChareIDType chareid;

    pack jam(in ,out, length)
    MSG2 *in;
    MSG1 **out;
    int *length;
    {
        int i;

        (*out) = (MSG1 *) CkAllocPackBuffer(in, sizeof(MSG1));
        for (i=0; i<in->size; i++)
            (*out)->x[i] = in->x[i];
        (*out)->size = in->size;
        (*out)->count = in->count;
        (*out)->chareid = in->chareid;
        *length = sizeof(MSG1);
        CkFreeMsg(in);
    }

    unpack unjam(in, out)
    MSG1 *in;
    MSG2 **out;
    {
        int i;

        (*out) = (MSG2 *) CkAllocMsg(MSG2);
        (*out)->x = (int *) CkAlloc(sizeof(int)*in->size);
        for (i=0; i<in->size; i++)
            (*out)->x[i] = in->x[i];
        (*out)->size = in->size;
        (*out)->count = in->count;
        (*out)->chareid = in->chareid;
    }

} MSG2;
```

Figure 11: A message with pack and unpack functions

6 Prioritized Execution

If one does not specify otherwise, charm will process the messages you send in roughly FIFO order. For most programs, this behavior is fine. However, some programs need more explicit control over the order in which messages are processed. Charm allows you to control queueing behavior on a per-message basis.

The simplest call available to change the order in which messages are processed is `CkSetQueueing`.

void CkSetQueueing(message, queueingtype)
MSG_TYPE message;
int queueingtype;

where queueingtype is one of the following constants:

CK_QUEUEING_FIFO
CK_QUEUEING_LIFO
CK_QUEUEING_IFIFO
CK_QUEUEING_ILIFO
CK_QUEUEING_BFIFO
CK_QUEUEING_BLIFO

The first two options, `CK_QUEUEING_FIFO` and `CK_QUEUEING_LIFO`, are used as follows:

```
msg1 = CkAllocMsg(...);
CkSetQueueing(msg1, CK_QUEUEING_FIFO);

msg2 = CkAllocMsg(...);
CkSetQueueing(msg2, CK_QUEUEING_LIFO);
```

When message `msg1` arrives at its destination, it will pushed onto the end of the message queue, as usual. However, when `msg2` arrives, it will be pushed onto the *front* of the message queue.

The other four options involve the use of priorities. To attach a priority to a message, one needs to set aside space in the message's buffer to hold the priority. To achieve this, we provide `CkAllocPrioMsg`:

void *CkAllocPrioMsg(msg, prioSizeInBits [, sizes_array])
MSG_TYPE msg;
int prioSizeInBits;
int * sizes_array;

This creates a message with a buffer inside it to hold the priority. The size of the buffer is specified in bits (!), although the size you specify is always padded to an integral number of `int`s. A pointer to the priority part of the message buffer and its size can be obtained with these calls:

unsigned int *CkPrioPtr(msg)
MSG_TYPE msg;

unsigned int CkPrioSizeBits(msg)
MSG_TYPE msg;

unsigned int CkPrioSizeBytes(msg)
MSG_TYPE msg;

unsigned int CkPrioSizeWords(msg)
MSG_TYPE msg;

There are two kinds of priorities which can be attached to a message, these are termed "integer priorities" and "bitvector priorities".

Integer priorities are quite straightforward. One allocates a message, setting aside enough space in the message to hold the priority, which is an integer. One then stores the priority in the message. Finally, one informs the system that the message contains an integer priority using `CkSetQueueing`:

```
msg = CkAllocPrioMsg(MsgType, CK_INT_BITS);
*CkPrioPtr(msg) = prio;
CkSetQueueing(msg, CK_QUEUEING_IFIFO);
```

The constant `CK_INT_BITS` is defined to be the number of bits in an integer, usually `(sizeof(int)*8)`. (Remember, the size of the priority is specified in bits). The constant `CK_QUEUEING_IFIFO` indicates that the message contains an integer priority, and that if there are other messages of the same priority, they should be sequenced in FIFO order (relative to each other). Similarly, a `CK_QUEUEING_ILIFO` is available. Note that `MAXINT` is the lowest priority, and `NEGATIVE_MAXINT` is the highest priority.

Bitvector priorities are somewhat more complicated. Bitvector priorities are arbitrary-length bit-strings representing fixed-point numbers in the range 0 to 1. For example, the bit-string "001001" represents the number $.001001_{\text{binary}}$. As with the simpler kind of priority, higher numbers represent lower priorities. Unlike the simpler kind of priority, bitvectors can be of arbitrary length, therefore, the priority numbers they represent can be of arbitrary precision.

Arbitrary-precision priorities are often useful in AI search-tree applications. Suppose we have a heuristic suggesting that tree node N1 should be searched before tree node N2. We therefore designate that node N1 and its descendants will use high priorities, and that node N2 and its descendants will use lower priorities. We have effectively split the range of possible priorities in two. If several such heuristics fire in sequence, we can easily split the priority range in two enough times that no significant bits remain, and the search begins to fail for lack of meaningful priorities to assign. The solution is to use arbitrary-precision priorities, aka bitvector priorities.

To assign a bitvector priorities, two methods are available. The first is to obtain a pointer to the priority field using CkPrioPtr, and to then manually set the bits using the bit-setting operations

inherent to C. To achieve this, one must know the format of the bitvector, which is as follows: the bitvector is represented as an array of unsigned integers. The most significant bit of the first integer contains the first bit of the bitvector. The remaining bits of the first integer contain the next 31 bits of the bitvector. Subsequent integers contain 32 bits each. If the size of the bitvector is not a multiple of 32, then the last integer contains 0 bits for padding in the least-significant bits of the integer.

The second way to assign priorities is only useful for those who are using the priority range-splitting described above. The root of the tree is assigned the null priority-string. Each child is assigned its parent's priority with some number of bits concatenated. The net effect is that the entire priority of a branch is within a small epsilon of the priority of its root.

To achieve this, the following call is available:

```
void CkPrioConcat(msg1, msg2, delta)
        MSG_TYPE msg1;
        MSG_TYPE msg2;
        unsigned int delta;
```

Where `msg1` must be a message containing a valid priority, and `msg2` must be an allocated message buffer with at least as many priority bits as `msg1`. The priority bits from `msg1` are concatenated with enough bits from delta to fill the priority buffer in `msg2`. The extra bits are taken from the least-significant end of delta.

It is possible to utilize unprioritized messages, integer priorities, and bitvector priorities in the same program. The messages will be processed in roughly the following order:

- Among messages enqueued with bitvector priorities, the messages are dequeued according to their priority. The priority "0000..." is dequeued first, and "1111..." is dequeued last.
- Unprioritized messages are treated as if they had the priority "1000..." (which is the "middle" priority, it lies exactly halfway between "0000..." and "1111...").
- Integer priorities are converted to bitvector priorities. They are normalized so that the integer priority of zero is converted to "1000..." (the "middle" priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority.
- Among messages with the same priority, messages are dequeued in FIFO order or LIFO order, depending upon which queuing strategy was used.

A final warning about prioritized execution: charm always processes messages in *roughly* the order you specify, it never guarantees to deliver the messages in *precisely* the order you specify. However, it makes a serious attempt to be "close", so priorities can strongly affect the efficiency of your program.

7 Multiple Charm Source Files

A Charm file typically consists of a module, which contains names (and definitions) of typedefs, specifically shared variables, chares, branch office chares and C functions. These names are *internal* to the module. Some of these names are *external* to other modules. Modules can interact with each other by referencing *external* names (defined in other modules). *External* names that a module references are referred to as the *imports* of the module, while its *internal* names that are referenced *externally* are referred to as its *exports.*

First, we discuss how a user specifies an *external* name declared in another module. These *external* names have to be specified in a manner that should indicate the module and the name being referenced. A name in another module can be referred to by prefixing the name with the *module-name::*. Therefore, for e.g., accesses to a function *F*, or a chare *C* or an entry point *E* inside chare *C*, defined in a module *M* would be made as *M::F*, *M::C* and *M::C@E* respectively.

Each module must begin with a declaration of its *imports* and *exports.* These are specified by using the **interface** construct. Figures ?? and ?? shows examples of the **interface** construct. The example illustrates how chares, functions, messages, accumulators, monotonics, tables, and read only names can be specified for external use. The specification of branch office chares is similar to that of chares, except that the keyword **chare** is replaced with the keyword **BranchOffice.** Functions inside branch office chares can be specified like global functions, except that the prefix **public** must be present. Type definitions are retained in their original.

A suggestion on how both the *imports* and *exports* can be specified: Every module can have its own **interface module** file, in which all local names that are accessed by other modules are defined. This file is included in the module that *exports* these *external* names, and in all modules that *import* some or all of the *external* names. In the example, the program *m1.p* includes two interface constructs, and the module *M1.* The interface construct with the module name *M1* specifies the exports of the module *M1.* The other one specifies the imported variables from the module *M2.* The module *M1* exports a readonly variable, a table, an accumulator, an monotonic variable and a function. Note that, it exports the definition of the accumulator and monotonic variable. These definitions are used by the module *M2* to create instances of the accumulator and the monotonic variable. Once these instances are created, their addresses can be passed to and used in any module. The module *M1* imports a message definition and a chare from the module *M2.* The module *M2* also exports and imports in the same way.

Note that in addition to these static mechanisms for exporting names to other modules, it is possible to pass references to entities in one module (i.e., entrypoints, function pointers) as fields of messages across modules (see Section ?? and Section ??).

The user can access C functions (defined in files which contain pure C syntax) from within Charm modules by declaring them as **extern** functions with syntax similar to C. The declarations can appear anywhere inside the module except in the chare variable declaration section (i.e., where the local variables of a chare are declared).

file: m1.p

```
#include "m1.interface"
#include "m2.interface"

module M1 {

    readonly int r;
    table t;
    accumulator {...} acc;
    monotonic {...} mono;

    chare chare1 {
        entry e1 : (message MSG1 *msg) {
            M2::MSG2 *msg2;
            ...
            CreateChare(M2::chare2, M2::chare2@e2,msg2);
        }
    }

    int f() {...}
}
```

file: m1.interface

```
interface module M1 {
    readonly int r;
    table t;
    accumulator acc;
    monotonic mono;
    int f();
}
```

Figure 12: Module M1 and its interface

file: m2.p

```
#include "m1.interface"
#include "m2.interface"

module M2 {

    message {int i;} MSG2;

    chare chare2 {
        entry e2 : (message MSG2 *msg) {
            ....
            i = ReadValue(M1::r);
            CreateAcc(M1::acc,accmsg,AccEntry,chareid);
            CreateMono(M1::mono,monomsg,MonoEntry,chareid);
            Find(M1::t,key,TabEntry,chareid);
            M1::f();
            ...
        }
        ...

    }
}
```

file: m2.interface

```
interface module M2 {
    message {int i;} MSG2;
    chare chare2 {
        entry e2;
    }
}
```

Figure 13: Module M2 and its interface

8 Dagger

Dagger is a coordination language which helps to express dependences among messages and computations. It augments the Charm language with *dag chare*'s. A detailed explanation of the Dagger language and the semantics of the constructs have been given in the *Dagger Language Tutorial.* In order to use Dagger constructs, the header file **dagger.h** must be included in the program, and the Charm library has to be linked in to the executable at run time (using **-lcharm**).

8.1 Syntax of a Dag Chare

The syntax of a dag chare appears in Figure ??. The C code blocks, private and public functions are the same as in the chare definition. In addition, they may contain Dagger control statements.

```
dag [chare|BranchOffice] <dag-chare-name> {

      Local Variables
      Condition Variables
      entry point declarations

      when depn_list : {C code block}
      ...
      when depn_list : {C code block}

      [private|public] FunctionB1(..)
            C-code-block
      ..
      [private|public] FunctionBZ(..)
            C-code-block
}
```

Figure 14: Syntax of a Dag Chare

8.2 Condition Variable Declaration

Condition variables are declared as follows:

CONDVAR condition_varname;

8.3 Entry Point Declaration

Entry points are declared as follows:

Entry entryname[[varname]] [MATCH] : (message msgtype *msgname);

The optional parameter [varname] must be declared in the local declaration section as an integer if it is used. This variable is initialized in the **init** entry. The Dagger translator requires one of the entries to be named **init**, and it assumes this entry is the one specified for creation of the dag chare. Dag chares are created in the same way other chares are created (i.e., with CreateChare or CreateBoc calls, and the these calls must use the **init** entry as creation entry point)

8.4 Specifying Dependences in When Blocks

Each when-block is guarded with a list of dependences. These dependences include entry point names and condition variables:

when e1,e2,..,en,c1,c2,...cm : { `C code block` }

when e[ANY] : { `C code block` }

where e,e1,...,en are entry points, and c1,...,cm are condition variables. Inside the when-block, messages are accessed by using the msgname which is defined in the entry point declarations. In the dependence list, all the entry names must be of the same type. i.e., matching or non-matching (which is specified by the **MATCH** option).

8.5 Dag Synchronization Calls

Expect(entryname,reference_number)

Ready(condition_variable,reference_number)

Set(condition_variable,initial_value)

Decrement(condition_variable)

where the reference_number is a positive integer value or the constant **DAG_NOREF**. **Expect** is used to specify dependences between messages and computations, and **Ready** is used to express dependences among the when blocks within the same chare. **Set** and **Decrement** are a variant of **Ready**. A condition variable is assumed to be initialized to the *not-ready* state, and a **Ready** call makes it *ready*. However, with **Set**, a condition variable can be assigned to a positive integer value. Each **Decrement** operation decrements the value of the condition variable by one. When the value reaches zero, the condition variable becomes *ready*. There is a restriction with this case: the condition variables may not use reference numbers. The *Dagger Language Tutorial* explains the semantics and usage of these constructs in more detail.

8.6 Reference Numbers and Messages

Messages, as well as condition variables, may be labelled with a reference number. Two functions are provided for this purpose.

SetRefNumber(msg, ref_num)

GetRefNumber(msg)

SetRefNumber sets the reference number of its message parameter msg to the value in the parameter ref_num.

GetRefNumber returns the reference number of its message parameter.

9 Charm Libraries

Charm provides the user with libraries of commonly used operations and instances of shared variables. The libraries are treated as external modules, hence all references to names in the libraries use the syntax described in Section ??. In addition, each library has a corresponding interface file which must be included inside each module which uses the library. The libraries can be linked in with the user code by using the '-lcharm' linker option.

9.1 Process Groups

Some applications have a natural breakdown of groups of processes, with communication among common group members occurring often. The process groups library (PG) is provided in the Charm system to

- Give the user an easy interface for efficiently defining and create process groups.
- Facilitate optimal communication among group members.
- Provide "hooks" into the library such that other libraries or user code can be built on top of the PG library.

Currently, process groups may only be built out of branches of BOC's, with each branch acting one and only one member of any group. A branch can however be part of many different groups.

All of the calls are non-blocking, and communication among group members is optimized via dynamically built spanning trees.

In order to use the library, the user must include the interface file `pg.int` shown in figure ??

Process groups are created by partitioning larger groups into subgroups. For example, you may wish to partition the processors into two groups, one containing the odd numbered processors, and another containing the even numbered processors. The groups are completely formed only after every processor in the original group has made a request to join one of the subgroups.

The **CreateRootGroup()** and **CreateRootGroupMsg()** functions are used to create the Process Groups distributed manager (a BOC), as well as supply the user with an initial group containing all processors which can be partitioned. The identifier (a ChareNumType) of the new manager is returned in a message.

When a processor requests to join a group, it must specify both the copy number, and the partition number of the group it wishes to join. The copy number is used like a reference number, and simply specifies which partitioning event this request applies to. The partition number specifies which group in that copy this processor wishes to join. When the group is completely formed, the new group identifier (gid) is returned in a message.

Group functions such as Multicast(), Synchronize(), and other reduction-type operations can then be efficiently carried out among group members.