The Charm (4.3) Programming Language Manual

September 9, 1994

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.

Contents

1 Introduction **5**

1.1 Charm Execution Model 5

1.2 Syntax of a Charm program 6

1.3 Structure of this Manual 8

2 Basic Charm Calls **9**

2.1 Basic Primitives 9

2.1.1 Creating Chares 9

2.1.2 Sending Messages 10

2.1.3 Chare IDs 10

2.2 Memory Allocation 11

2.2.1 Ordinary Memory Allocation 11

2.2.2 Memory Allocation for Messages 11

2.2.3 Variable Sized Messages 11

2.2.4 Memory Allocation for Prioritized Messages 12

2.2.5 Copying Messages 13

2.2.6 Deallocating Memory 13

2.3 Input/Output Primitives 14

2.4 Quiescence Detection 14

2.5 Charm Exit Calls 15

2.6 Function Calls 15

2.7 Miscellaneous Utility Routines 16

3 Branch Office Chares **18**

3.1 Syntax of a Branch Office Chare 18

3.2 Basic System Calls for Branch Office Chares 18

3.3 Spanning Tree System Calls 20

4 Information Sharing Abstractions **22**

4.1 Read Only Variables and Messages 22

4.2 Accumulator Variables 23

4.3 Monotonic Variables 25
4.4 Write Once Variables 27
4.5 Dynamic Tables 30

5 Message Packing **33**

6 Prioritized Execution **36**

7 Multiple Charm Source Files **39**

8 Dagger **42**
8.1 Syntax of a Dag Chare 42
8.2 Condition Variable Declaration 42
8.3 Entry Point Declaration 42
8.4 Specifying Dependences in When Blocks 43
8.5 Dag Synchronization Calls 43
8.6 Reference Numbers and Messages 44

9 Charm Libraries **45**
9.1 Process Groups 45
9.2 Reduction 49
9.3 Barrier 54
9.4 Accumulators 54

10 Advanced Features **58**

A Installing and Running Charm Programs **59**
A.1 Installing Charm 59
A.2 Compiling Charm Programs 60
A.3 Executing Charm Programs 62
A.4 Command Line Options 63
A.4.1 Additional Uniprocessor Command Line Options 64
A.4.2 Additional Network Command Line Options 64
A.5 Queueing and Load Balancing Strategies 65
A.5.1 Queueing Strategies 65

A.5.2 Load Balancing Strategies 65
A.5.3 Modes of Execution 66

B Charm System Types 67

C Changes in this Version 68

D Common Programming Mistakes in Charm 69

E Known Bugs/Problems 70

F Further Information 71
F.1 Related Publications 71
F.2 Associated Tools 71
F.3 Contacts 71

INDEX

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.

Shared	**Nonshared**
Encore Multimax	Thinking Machines CM-5
Sequent Symmetry	nCUBE
Unix based uniprocessor	Intel Paragon
	Intel i860
	IBM SP-1
	Network of UNIX workstations

Table 1: Machines on which Charm is supported.

Table 1 shows the list of MIMD machines on which Charm is currently supported. 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 F.

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.

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

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" 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, but otherwise it is free to schedule them in any way consistent with a queueing strategy that may be chosen at link time.

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 F 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 1 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 2. 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 3) 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 3 and Section 4, respectively.

```
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

1.3 Structure of this Manual

This manual describes the various Charm calls and how to run Charm programs. Section A describes the procedure for installing Charm, compiling Charm programs and running them on different machines. Charm allows the user to choose from alternative run-time strategies for memory management, queuing, and load balancing. These are also described in Section A. Basic Charm calls are described in Section 2. Charm provides specific information sharing abstractions. The syntax of these abstractions and the related system calls are described in Section 4. 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 3, we describe the syntax and system calls related to branch office chares. In nonshared memory machines, pointers inside messages (if used) must be "packed" before the message can be sent across address-space boundaries. Calls for *message packing* are described in Section 5. Prioritized execution of messages is described in Section 6. We discuss the syntax of Charm programs with multiple modules in Section 7. Commonly used Charm features are now being provided as libraries. In Section 9, we describe the libraries currently available. Section 8 describes Dagger, a coordination language (running on top of Charm) that allows dependences between messages and computations to be expressed.

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 described in Appendix B.

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 3 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 3).

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 2.2) 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.

2.1.3 Chare IDs

The ID of a chare can be obtained through the two system calls — **MyChareID** and **MainChareID**. These IDs can be used by the **SendMsg** primitive to send messages to entry points at chare instances specified by those IDs. Note that these IDs are currently implemented as structures, so user manipulation of chare IDs (such as printing) is not possible.

```
MyChareID(chareid)
        ChareIDType *chareid;

MainChareID(chareid)
        ChareIDType *chareid;
```

These calls take a pointer to an already allocated structure of type **ChareIDType**, chareid. It is the responsibility of the user to allocate space for chareid before the call is made. The appropriate fields in the structure are modified as a result of these calls so that after the **MyChareID** call, the parameter points to the ID of the currently executing chare. Similarly, after the **MainChareID** call, the parameter points to the ID of the main chare.

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**.

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 2.2.3 below for further discussion of variable sized messages.

2.2.3 Variable Sized Messages

In Section 5 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 3.

```
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

In the example in Figure 3, 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.

2.2.4 Memory Allocation for Prioritized Messages

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

The **CkAllocPrioMsg** call is specifically for allocating space for messages that have **integer** or **bit-vector** priorities. The call allocates a message of type **MSG_TYPE** and a priority field of size prioSize words. A pointer is returned to the allocated space, if successful; otherwise it returns NULL. This call is different from **CkAllocMsg()** in that additional information is kept by the system to facilitate the handling of the message with *priorities* by other system routines. Any data to be used as a prioritized message and passed to system routines MUST be allocated using **CkAllocPrioMsg()**. See Section 6 for more details on different types of priorities and how to access them.

msg: A pointer to the message being allocated. The This usage is the same as that for non-prioritized messages.

prioSize: The size of the priority field (in words).

sizes_array: An optional array parameter used only for variable sized messages to specify the size of the message. This usage is the same as that for non-prioritized variable sized messages. See Section 2.2.3 for further discussion of variable sized messages.

2.2.5 Copying Messages

MSG_TYPE *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.

2.2.6 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()
unsigned int CkUTimer()
int CkHTimer()

The **CkTimer**, **CkUTimer** and **CkHTimer** calls are used to time programs. **CkTimer** and **CkUTimer** return the time since the process began execution on that node. **CkTimer** returns the time in milliseconds, while **CkUTimer** returns process time in microseconds. It is important to remember that to print out the value in microseconds one must use the %u option with **CkPrintf**, instead of %d. Since, **CkUTimer** rolls overs for an execution of 1 hour in our implementation, a **CkHTimer** call is provided to give the number of hours since the program started executing. This call should be used in conjunction with the **CkUTimer** call. **CkHTimer** returns -1 on machines on which it hasn't been implemented.

int McTotalNumPe()

McTotalNumPe returns the total number of processors on which the user program is being run. In versions of Charm prior to 4.1, this command was called **McMaxPeNum** which is still supported for purposes of backward compatibility.

int McMyPeNum()

McMyPeNum returns the processor number of the processor on which the **McMyPeNum** call was made. The numbers assigned are in the range 0..**McTotalNumPe()**-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 4. 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 2.7).

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 McSpanTreeRoot()

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

int McSpanTreeParent(node)
    int node;

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

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

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

int McNumSpanTreeChildren(node)
    int node;

The **McNumSpanTreeChildren** 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 5, `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 $\times$ 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 6. 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 7, `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 8, 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 9 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 10.

```
message {
    int dummy;
} SomeMsg;

message {
    WriteOnceID wovID;
} WovMSG;

chare foo {
    entry E1:  (message SomeMsg *dummy) {
        int theVar = 20;
        ChareIDType myID;

        MyChareID(&myID);
        WriteOnce(&theVar, sizeof(int), foo@E2, myID);
    }
    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 % McTotalNumPe()); /* 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 11.

```
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

Charm provides many alternative strategies for managing its queues of messages (section A.5.1). Some of them (such as stack, fifo, ... etc.) do not require any explicit priority attached to individual messages. When it is required to give priorities to messages based on their intrinsic merit (as opposed to the time at which they were created) one must use priorities explicitly. Priority of a message is its importance in relation to other messages. It is the responsibility of the user to generate and assign priorities to messages.

Priorities are very useful in any application where the user would like to have the messages executed in some order of priority. As an example, priorities can be effectively used to reduce the number of nodes expanded in a branch and bound solution of the Traveling Salesperson Problem by assigning the cost of the branch and bound nodes as the nodes' priority.

Priorities may be of several different types as explained below. Based on the type of priority and the queueing strategy selected, Charm attempts to assign resources first to high priority messages.

Priorities can be of two types:

1. **Integer Priorities**:

   This priority type uses 32-bit integers where priority decreases as its value increases. So a message with priority 0x0 is considered the highest priority message and the lowest priority message will have a priority value of 0x7fffffff.

2. **Bit Vector Priorities**:

   If priorities longer than 32 bits are needed, then a bit vector priority may be used. A bit vector priority (or priority vector) is a sequence of binary digits of arbitrary length. A message with priority P_1 is *higher* in priority than P_2, if P_1 is *lexicographically* smaller than P_2 (e.g. $P_1 =$ 101011000 is at higher priority than $P_2 =$ 101101).

   Bit Vector Priority Format: The most significant *byte* of the first 32-bit word encodes the number of bits used in the bit vector. Currently, a maximum of 256 bits can be used to encode priorities to messages. The bit vector priority begins with the second byte of the first word as shown in Figure 12, thus using the 24 bits <23:0> in the first word.

   As the number of bits needed increases, multiple long words can be allocated to attain the required bit vector length. The vector length in the most significant byte of the first word must be updated accordingly.

   The lexicographical comparison to compare two bit vectors begins with bit 23 of the first word and proceeds left to right.

Charm provides function calls and macro calls to access and assign the above priorities. The following calls are used to access the priority field in a message allocated using the **CkAllocPrioMsg()** call described in Section 2.2.

PVECTOR *CkPriorityPtr(msgPtr)
    void *msgPtr;

First word:

31	23		0
Vector Length in No. of Bits			

Successive words if required:

31			0

Figure 12: Bit vector priority format

The calls return a pointer (of type PVECTOR) to the priority field in the message msgPtr. This can be used to assign priorities to the message in the following manner:

```
message {
    int cost;
    unsigned int isDefined[MaxSquare];
    unsigned int in[MaxSquare];
} MSG1;
...
MSG1 *msg;
...
msg = (MSG1 *) CkAllocPrioMsg(MSG1, sizeof(int));
x = (int *) CkPriorityPtr(msg);
*x = cost;
```

The above example shows how to assign integer priorities. Bit vector priorities are assigned in a similar fashion.

int CkPrioritySize(priorityPtr, Bits)
PVECTOR *priorityPtr;
int Bits;

This call is used only for bit vector priorities. The call takes a pointer to a bit vector priority and the number of bits Bits that may need to be appended to this vector to generate a new bit vector priority. It returns the size in number of full words needed as appending the bits may go over the word boundary. This size may then be used to allocate a message using the call **CkAllocPrioMsg()**.

PGEN_GET_NEW_PVECTOR(priorityPtr1, priorityPtr2, Bits, rel_prio)
PVECTOR *priorityPtr1;
PVECTOR *priorityPtr2;
int Bits;
int rel_prio;

This *macro* call is used only for bit vector priorities. It takes a pointers to a two bit vector priorities priorityPtr1 and priorityPtr2, and the number of bits Bits to encode the relative priority rel_prio. It copies the bit vector priority of priorityPtr1 to priorityPtr2 and appends the rel_prio to priority pointed by priorityPtr2 using Bits number of bits. Memory space must be allocated for priorityPtr2 using the **CkAllocPrioMsg()** call and the pointer is obtained using the **CkPriorityPtr()** call described above.

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 13 and 14 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 2.6 and Section 2.1).

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 13: 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 14: 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 15. 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 15: 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 7. 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.

```
interface module PG {
   CreateRootGroup() ;
   CreateRootGroupMsg() ;

   Partition() ;
   Multicast() ;
   Synchronize() ;

   PgMySpanTreeParent() ;
   PgMyNumSpanTreeChildren() ;
   PgMySpanTreeChildren() ;
   PgGroupSize() ;
   PgMyRank() ;

   message {} CREATE_ROOT ;
   message {ChareNumType rootBocNum ;} ROOT_GROUP_CREATED ;
   message {int newGid ;} PARTITION_CREATED ;
}
```

Figure 16: The format for the Process Groups (PG) library interface

PG::CreateRootGroup()

PG::CreateRootGroupMsg(ep, boc)
    EntryPointType ep;
    ChareNumType boc;

Before any group operations can be performed, a root group must be created from which to partition. **CreateRootGroup()/ CreateRootGroupMsg()** perform the same operation, and only differ in where they should be called from. CreateRootGroup() should only be called from inside CharmInit, and will immediately return the identifier (ChareNumType) of the new PG manager. CreateRootGroupMsg() can be called from anywhere in the user program (by one processor only, usually) and will return the identifier to the specified entry point and BOC in a message of type ROOT_GROUP_CREATED with a single field rootBocNum.

These calls also provide the user with an initial root group to partition. **This initial root group is assumed to always be 0.**

PG::Partition(coord, gid, copy, partition, ref, ep, boc)
ChareNumType coord;
int gid;
int copy;
int partition;
int ref;
EntryPointType ep;
ChareNumType boc;

This call is used to specify which group each branch (or processor) wishes to join. `gid` specifies which group is being partitioned. If partitioning the initial root group consisting of all processors (branches), gid will always equal 0. It is also possible to partition user-created groups into subgroups, in which case gid will equal the identifier returned by a previous call to **partition**.

`copy` and `partition` together specify a unique group to join. copy is used to specify which type of group you are creating. partition is used to specify which group in the specified copy you wish to join. In both cases, the values chosen are entirely up to the user, as long as they are integer values.

For example, you may wish to partition a matrix into groups of rows and columns. Each processor would thus call partition twice: once to specify which row to join (for example using *copy=0, partition=myRow*), and once to specify which column to join (for example using *copy=1, partition=myCol*).

Before a partition is complete, all members of the group being partitioned must join exactly one partition in each copy. When all members of the root group have made their call to partition, the library will return a new group identifier for that group to the specified entry point and BOC in a message of type PARTITION_CREATED with a single field newGid. This message will also be tagged with the reference number supplied by the *ref* parameter, and can be retrieved from the PARTITION_CREATED message via the **GetRefNumber()** function. This is particularly useful when you want to have one entry point handling group creation messages for several different group types.

PG::Multicast(coord, gid,msg, ep, boc)
ChareNumType coord;
int gid;
void *msg;
EntryPointType ep;
ChareNumType boc;

Multicast() sends the message specified by msg to all members in the group specified by gid. If gid equals 0 (the default root group containing all branches), then multicast performs the same basic function as **BroadcastMsgBranch()**. Communication to members in the group is optimized via spanning trees built during the partition phase.

PG::Synchronize(coord, gid, key, msg, ep ,id)
ChareNumType coord;
int gid;
int key;
void *msg;
EntryPointType ep;
ChareIDType id;

Synchronize() can be used to create a barrier synchronization among group members. In addition, each branch can supply a message to be delivered to the specified entry point and ChareID upon synchronization. Synchronize is totally non-blocking, so that a processor can go ahead and perform other work while the rest of the processors are still reaching the barrier point. The "key" parameter is used so that different synchronizations do not overlap and conflict.

In addition to the high level functions described above, there are several functions which allow the user program to determine the structure of the process group as it was built. This is useful for other programs which may wish to build on top of the process groups library, and thus need to know something about the structure of the groups.

Each group is represented as a spanning tree. Global communication among group members can thus be performed in time logarithmic to the number of group members. The following functions are provided in the PGlib to help the user program determine the structure of the trees after they have been built. In all cases, `coord` means the ChareNumType of the PG manager returned by either **CreateRootGroup** or **CreateRootGroupMsg**, and gid stands for the group identifier. In all cases, the functions return immediately, and each function can only determine the values for itself (you cannot, for example, directly find out the children of another member in your group without having that member make the call and pass the value(s) to you).

int PG::PgMySpanTreeParent(coord, gid)
ChareNumType coord;
int gid;

The PE number of the parent is returned as an integer. If the calling processor is the root of the tree, then the return value will be -1.

int PG::PgMyNumSpanTreeChildren(coord, gid)
ChareNumType coord;
int gid;

This function will return an integer value corresponding to the number of children that the current processor has in this group. If the value is 0, then the member is a leaf - otherwise it is either the root or an intermediate node.

int PG::PgMySpanTreeChildren(coord, gid, children)
ChareNumType coord;
int gid;
int *children;

This function takes an additional parameter which is a pointer to an array of integers. The function will fill in this array with the PE numbers of the span tree children. Note that the only valid entries in the return array will be the first n elements, where n is the value returned by **PgMyNumSpanTreeChildren()**.

int PG::PgGroupSize(coord, gid)
ChareNumType coord;
int gid;

This function can be used to determine the size of a group. The group (gid) must be already formed, and the calling processor must be a member of the group in question.

PG::PgMyRank(coord, gid)
ChareNumType coord;
int gid;

The function `PgMyRank()` returns a unique integer value in the range $0 \ldots G-1$, where G is the vaue returned by `PgGroupSize()`. A processor will have a unique and possibly different rank for each group that it is in.

Figure 17 shows a BranchOffice chare example in which processors are split into groups of even and odd numbered PE's. They exchange messages with all the members in their group via a multicast, each group synchronizes with it's other members, and the program comes to an end.

9.2 Reduction

The reduction library contains modules for global reduction operations including maximum, minimum, sum, and product on vectors of data of type int, float or double. In addition, a count operation is available which returns the number of non zero elements in an integer vector.

The reduction operations are designed to accept partial reductions from each processor in a BranchOffice chare. One reduction-request must be made from each branch, and these requests are combined and the result is distributed back. In addition, functions to solve the local portion of the reduction (the partial reduction) are provided.

The following features are found in the Charm reduction library:

1. Reductions may be performed over all processors (every member of a BranchOffice chare) or over a group defined by the Process Groups library (Section 9.1).

```
#include "pg.int"
module pg_example {
#include "qd.h"
message { ChareNumType pgCoord ;} START_MESSAGE ;
message { char evenodd[6] ;} TEST_MESSAGE ;
BranchOffice TestBoc {
    int me,odd,finalGid,left ;
    ChareIDType myBocID ;
    ChareNumType myBocNum, coordBoc ;

    entry start : (message START_MESSAGE *inMsg) {
        coordBoc = inMsg->pgCoord ;
        me = McMyPeNum() ;
        leftToFinish = McTotalPeNum() ;
        myBocNum = MyBocNum() ;
        MyBranchID(&myBocID) ;
        StartQuiescence(qdetect, myBocID) ;
        if (me % 2 == 0) odd = 0 ;
        else odd = 1 ;
        PG::Partition(coordBoc, 0, 0, odd, 1, OddEvenPart, myBocNum) ;
    }
    entry OddEvenPart : (message PG::PARTITION_CREATED *inMsg)
    {
        int i, totalSize ;
        TEST_MESSAGE *outMsg ;
        finalGid = inMsg->newGid ;
        outMsg = (TEST_MESSAGE *)CkAllocMsg(TEST_MESSAGE) ;
        SetRefNumber(outMsg, me) ;
        if (odd) strcpy(outMsg->evenodd, "odd") ;
        else strcpy(outMsg->evenodd, "even") ;
        PG::Multicast(coordBoc,finalGid,outMsg,deliverEp,myBocNum) ;
    }
    entry deliverEp : (message TEST_MESSAGE *msg) {
        int ref ;
        ref = GetRefNumber(msg) ;
        CkPrintf("I am PE %d and I am %s, as is my sender %d?\n",
                  me, msg->evenodd,ref) ;
    }
    entry qdetect : (message QUIESCENCE_MSG *qmsg) {
        if(me==0) CkPrintf("Quiescence detected\n") ;
        PG::Synchronize(coordBoc, finalGid, 101, qmsg, groupSync, myBocID) ;
    }
    entry groupSync : (message QUIESCENCE_MSG *qmsg) {
        CkPrintf("Group %d on pe %d has synchronized.\n",finalGid,me) ;
if(--leftToFinish == 0)
            CkExit() ;
  }
}
}
```

Figure 17: An example using the Process Groups library

2. The type of reduction library to be created, and the scope over which it is created is decided at the time of the creation of the library.

3. Concurrent reductions are supported by passing a vector into the **DepositData()** functions.

4. The size of the vector is specified at the time the library is used, instead of created, allowing the same library instance to perform different sized reductions.

5. Reduction operations using the same library instance can overlap by specifying a "key" or "reference number".

Table 2 outlines the types of reductions currently supported by the reduction library, along with the interface file which must be included, and the name of the module.

DataType	Function	InterfaceFile	ModuleName
integer	maximum	imaxredn.int	IMaxRedn
integer	minimum	iminredn.int	IMinRedn
integer	sum	isumredn.int	ISumRedn
integer	product	iprodredn.int	IProdRedn
integer	count	icountredn.int	ICountRedn
float	maximum	fmaxredn.int	FMaxRedn
float	minimum	fminredn.int	FMinRedn
float	sum	fsumredn.int	FSumRedn
float	product	fprodredn.int	FProdRedn
double	maximum	dmaxredn.int	DMaxRedn
double	minimum	dminredn.int	DMinRedn
double	sum	dsumredn.int	DSumRedn
double	product	dprodredn.int	DProdRedn

Table 2: The table for interface file and module names for the reduction library

The interface files, listed in Figure 18, specify some data structures and functions to be used in the calls to the reduction library.

The functions provided by the reduction library are non-blocking, that is, they immediately return. Eventually, the reduction library replies with the result. Five functions are provided: **Create()**, **CreateOverGroup()**, **LocalReduction()**, **DepositData()**, and **DepositDataMsg()**.

ChareNumType Create()

```
interface module ModuleName {
    message { int ref ;
        varSize DataType data[];
    } REDN_MSG;
    message {
        ChareNumType rednBoc ;
    } REDUCTION_CREATED ;
    Create();
    CreateOverGroup();
    LocalReduction();
    DepositData();
    DepositDataMsg();
}
```

Figure 18: The format for the reduction library interface

ChareNumType CreateOverGroup(gid, pgBoc, retEP, retBoc)
 int gid;
 ChareNumType pgBoc;
 EntryPointType retEP;
 ChareNumType retBoc;

An instance of the reduction module is created by one of these calls.

Create() creates a reduction library instance over all processors, and all processors must participate in subsequent reductions using that instance. The call *must* be invoked in the **CharmInit** entry point, and it immediately returns an identifier (of type *ChareNumtype*) which must be used in subsequent calls.

If you have already created Process Groups using the Process Groups library, then you can also create a reduction instance over any valid group using **CreateOverGroup()**. You need to supply the group identifier of the group over which the reduction will take place, the *ChareNumType* of the process groups coordinator returned by the **CreateRootGroup()** command in the PG library, and the entry point and BOC number at which the new ChareNumType of the created reduction library will be returned. The return entry point should accept a message of type `redntype::REDN_CREATED`, (where `redntype` is the name of the reduction module), and it will have a single field called `rednBoc` which is the BOC number of the new reduction.

Note: `CreateOverGroup()` can also be used to create a reduction instance over all processors outside of CharmInit(). The process group library must be at least created (even if no partitions are formed), and a group identifier value of 0 should be used.

Usually a reduction operation will consist of two steps:

- Performing the local portion of the reduction to get a partial reduction for each processor.

- Taking the partial reductions and completing the reduction operation via communication and comparison with other processors.

In this reduction library implementation, each element of the vector (possibly a single element) passed into the **DepositData** functions is reduced element by element with the corresponding elements from the other processors. Thus, if an array is being reduced which was larger than N, where N is the number of processors in the reduction, then a partial (local) reduction must be performed prior to the final step.

DepositDataMsg(rednboc, x, nel, ref, ep, id)
ChareNumType rednboc;
DataType x[];
int nel;
int ref;
EntryPointType ep;
void *id;

Each node contributes nel elements of data pointed by x to the reduction by calling this function. rednboc is the reduction identifier returned by the **Create** call. The reduction library sends a message (of type ModuleName::REDN_MSG) to the entry point specified by entry point ep, and address id. The reduced data is identified by `data->ref` (originally supplied by the user), and accessed by `msg->data`. The return id must be specified to be the id of that return object. Note that branch office chares must be distinguished from ordinary chares for this purpose. I.e., if the calling chare is a branch office chare then to return to that chare the parameter id must be initialized with the **MyBranchID()** system call, otherwise the **MyChareID()** call should be used. If the id is NULL, then no message will be sent the caller (this feature can be used to have the result of the reduction delivered to just one (or some) specific processors, for example).

DepositData(boc,x,z,nel,ref,fptr,id)
ChareNumType boc;
DataType x[],z[];
int nel ;
int ref ;
void (*fptr)() ;
void *id;

This call is like the **DataDepositMsg** call, but with a different reply mechanism. The reduction library eventually replies by calling the function specified by the function pointer fptr, and the pointer to the chare identifier id. The function pointed to by fptr must have a formal parameter of type **ChareNumType**. This parameter will have the id of the reduction module. This is useful if more than one instances of reduction module (concurrent) calls back the same function. The result will be copied to the area pointed by y just before the reply call.

A reduction branch office chare created by the **Create** call can be used to carry out multiple consecutive reductions. However, care must be taken to assign correct reference numbers so that overlapping reductions do not get mixed up. Also, the user should take care to make sure that the vectors are the same size for any given reduction. Unpredictable results can occur otherwise.

In Figure 19, a sample program is listed to illustrate the usage of the reduction library. In this example, two different reduction modules are used concurrently

9.3 Barrier

In addition to these global operations, a *barrier* module is also available in the reduction library. The *barrier* module is the same as a reduction module except there is no data to be reduced. The interface file for the *Barrier* module is `barrier_redn.int`, and it exports following functions, analogous the functions exported by the reduction modules:

- **Create()**
- **Signal(rednboc,fptr,id)**
- **SignalMsg(rednboc,ep,id)**

To use this library, the program must include the file "barrier_redn.int" which is shown below:

```
interface module Barrier {

message {ChareNumType id;} BARRIER_MSG;
Create();
Signal();
Signal_msg();
}
```

To use the barrier reduction, first create an instance of the barrier reduction library using **Create()**, followed by each processor issuing either **Signal()** or **SignalMsg()** command. The only difference between these two commands is the method by which the program is notified when the barrier is complete.

If **Signal()** is used, then the function specified by `fptr` is called. The function pointed to by `fptr` must have a single formal parameter of type **ChareNumType**. This parameter will have the `id` of the reduction module used. This is useful is more than one instance of the reduction module calls back the same function.

if **SignalMsg()** is used, then a message is sent to the entry point and chare (or branch) specified by *ep* and *id*. The message of type BARRIER_MSG is returned to the specified entry and chareID. The single field `id` is uninportant, but contains the Boc number of the reduction library if the user program needs it.

9.4 Accumulators

The accumulator library contains definitions of the two most commonly used types of accumulators: counters and histograms. There are three types of counter and histogram accumulators: integer, float and double. In order to use the accumulator library, one needs to include the proper interface

```
#include "isumredn.int"
#define REF1 0
#define REF2 1
module REDNTEST {
message {
    ChareNumType isumboc;
    ChareNumType fmaxboc;
} MSG;
chare main {
    entry CharmInit : {
        MSG *msg;
        msg = (MSG *)CkAllocMsg(MSG);
        msg->isumboc = ISumRedn::Create();
        msg->fmaxboc = FMaxRedn::Create();
        CreateBoc(test,test@init,msg);
    }
}
BranchOffice test {
    ChareNumType isumboc, fmaxboc ;
    int x ;
    float fx, r_fx ;
    ChareIDType  mycid ;
    ChareNumType myBocNum ;
    entry init : (message MSG *msg) {
        isumboc = msg->isumboc ;
        fmaxboc = msg->fmaxboc ;
        MyBranchID(&mycid);
        myBocNum = MyBocNum() ;
        x = McMyPeNum() ;
        fx = (float)McMyPeNum() ;
        waitFor = 2 ;
    ISumRedn::DepositDataMsg(isumboc,&x,1,REF1,isumrecv,&mycid) ;
    FmaxRedn::DepositData(fmaxboc,&fx,&r_fx,1,REF2,fmaxfcn,&myBocNum) ;
    }
    public fmaxfcn(rednlib)
    ChareNumType rednlib ; {
        CkPrintf("FMAX result is %f\n",r_fx) ;
        if (--waitFor==0) CkExit() ;
    }
    entry isumrecv : (message ISumRedn::REDN_MSG *msg) {
        CkPrintf("ISUM result(s) for ref %d is %d ",msg->ref,msg->data[0]);
        if (--waitFor==0) CkExit() ;
    }
}}
```

Figure 19: Usage of the reduction library

```
interface module ModuleName{
        message { int data; } MSG;
        Create();
        Add();
        Collect();
}
```

Figure 20: interface file: iacc_count.int

```
interface module ModuleName{
        message { int size; varSize double data[]; } MSG;
        Create();
        Add();
        Collect();
}
```

Figure 21: interface file: dacc_hist.int

file and refer to external names with the name of the module. Tabler 3 shows the names of the interface files and modules for the accumulator corresponding to its accumulator type (counter or histogram) and its data type (integer, float or double).

Accumulator Type	**DataType**	**InterfaceFile**	**ModuleNam e**
counter	integer	iacc_count.int	ICount
counter	float	facc_count.int	FCount
counter	double	dacc_count.int	DCount
histogram	integer	iacc_hist.int	IHistogram
histogram	float	facc_hist.int	FHistogram
histogram	double	dacc_hist.int	DHistogram

Table 3: The table for interface file and module names for accumulator library

Figure 20 shows the interface file for an integer counter accumulator, and Figure 21 shows the interface file for a histogram accumulator where each entry is of type double. In the following description `X` stands for I (Integer), F (Float), or D (Double).

The counter accumulator can be created using the following call:

```
acc1id = XCount::Create()
```

The histogram accumulator can be created using the following call:

```
        entry Collect: (message Dummy *msg) {
            CharеIDType myid;

            MyChareID(&myid);
            XCount::Collect(acc1id, Recv1, &myid);
            XHistogram::Collect(acc2id, Recv2, &myid);
        }

        entry Recv1: (message XCount::MSG *msg){
            CkPrintf("Total number of nodes = %X\n", msg->data);
        }

        entry Recv2: (message XHistogram::MSG *msg){
            int i;

            for (i=0; i<msg->size;i++)
            if (msg->data[i] >0)
                CkPrintf("Number of nodes at level %d = %X\n", i, msg->data[i]
);
        }
```

Figure 22: Collecting the value of the counter and histogram accumulator

```
        acc2id = XHistogram::Create(size)
```

In the above call `size` is the number of histogram levels.

The counter and the histogram accumulators can be incremented as follows:

```
        XCount::Add(acc1id, incr);
        XHistogram::Add(acc2id, level, incr);
```

The first call increments the counter by `incr`; the second call increments by `incr` the count of index `level` in the histogram, where $0 \leq level < size$.

The values of a counter accumulator and a histogram accumulator can be collected as shown in Figure 22. Note that the message types for the accumulators are external to the program, hence they must be referred to as `XCount::MSG` and `XHistogram::MSG`. The message definitions appear in the interface file.

10 Advanced Features

Although not supported in the current release of Charm(4.3), future releases of the Charm parallel programming system will allow users to write customized load balancing strategies, queue managers and memory managers. For further information regarding this or other system-related features, mail may be sent to the Parallel Programming Laboratory at **ppl@cs.uiuc.edu**.

A Installing and Running Charm Programs

In this section, we describe how to install Charm, how to compile and execute Charm programs, and the available command line options. We also describe queueing and load balancing strategies, and the various modes of execution of Charm programs.

A.1 Installing Charm

Charm is available by anonymous ftp from the site a.cs.uiuc.edu (128.174.252.1) in the directory pub/CHARM/CHARM.4.3. The code names for the machines on which Charm currently runs are:

Machine	Machine Name
Encore Multimax	max
IBM SP-1	sp1
Intel iPSC/860	i860
Intel iPSC/860 with remote host	i860R
Intel Paragon running OSF	paragon
Intel Paragon running SUNMOS	sunmos
nCUBE without host	ncube3
Sequent Symmetry	sequent
Thinking Machines CM-5	cm5
Network of Sun SPARC workstations	network-sun
Network of Suns running Solaris	network-solaris
Network of HP workstations	network-hp
Network of IBM RS6000 workstations	network-rs6000
Sun uniprocessor running SunOS	uniprocessor-sun
Sun uniprocessor running Solaris	uniprocessor-solaris
HP uniprocessor	uniprocessor-hp
IBM RS6000 uniprocessor	uniprocessor-rs6000

To install Charm for a particular machine, the following files need to be obtained from the aforementioned directory at the ftp site :

- INSTALL : A C shell script which installs Charm.
- sources.tar.Z : A tarred, compressed file which contains the translators and other utilities
- `<machine name>`.tar.Z : A tarred, compressed file which contains the Charm runtime system for the machine `<machine name>`.

In addition, the manuals and other documentation for Charm are in the file doc.tar.Z, and Projections, SummaryTool, and DagTool tools are in the file tools.tar.Z. See the README file in the CHARM.4.3 directory or Appendix F for more information on these.

After obtaining the three files listed above, Charm can be installed in the current directory by typing :

```
INSTALL charm <machine name>
```

This uncompresses, untars and "makes" the system for the machine `<machine name>`. After installation is complete, the following directories should be present :

- bin : this contains the translators and other utilities.
- translators : the sources for the translators
- charm_pgms : sample Charm programs which illustrate some Charm features
- charm++_pgms : sample Charm++ (based on C++) programs
- `<machine name>` : the Charm runtime

A.2 Compiling Charm Programs

An easy way to get started with compiling and running Charm programs is :

- include the Charm `bin` directory in your `path` environment variable (alternatively, create a symbolic link to **charmc**
- copy one of the sample programs directories (along with the Makefile, etc)
- type "make". This should produce the Charm executable, which is run using the commands described in Section A.2.

The Charm compiler is called **charmc**, and now does linking as well as compiling. It is installed in the Charm `bin` directory.

All Charm source code files must have the extension `.p`. The Charm compiler is be invoked for one Charm source file at a time. The **charmc** command recognizes the following arguments specially. The first group of arguments are to compile-time options while the second group are link-time options.

Argument	Meaning	Possible Values
-c	Compile only (don't link after compilation)	(none)
-cc	Name of C compiler (for Charm programs)	(user defined)
-c++	Name of C++ compiler (for Charm++ programs)	(user defined)
-D	Set macro value as if by #define	(user defined)
-language	Select Charm or Charm++	charm, charm++
-machine	Compile for machine	(see machine code names above)
-O	Set optimization level of compiler	(defined by C or C++ compiler)
-save	Save translator's temporary files	(none)
-verbose	Give verbose feedback about compilation process	(none)

Argument	Meaning	Possible Values
-balance	Load balancing strategy	rand, acwn, mngr, tok, btok, replay, recv
-execmode	Execution mode	notrace, summary, projections, record, replay
-g	Compile for run-time debugging	(none)
-l	Link with a library	(user defined)
-language	Language	charm, charm++
-ld	Name of linker	(user defined)
-machine	Target machine	(see machine code names above)
-memory	Memory management	table, unix, leak
-o	Name object file	(user defined)
-queue	Message queueing strategy	fifo, stack, fl, bfifo, ififo, bstack, istack, rand, replay
-s	Strip executable file	(none)

In reality, **charmc** does not enforce the meaning of some of these options. Those arguments, if present, are passed to the preprocessor (-D), to the C or C++ compiler (-cc-option, -c++-option, -D, -g, -O) or to the linker (-ld-option, -D, -g, -s). Any other arguments not enumerated above (other than the source files) are passed to both the C/C++ compiler and the linker. Those options that are directed to the Charm system per se are explained in more detail below.

An example of a use of **charmc** is as follows:

charmc -c pgm.p

This results in the object file pgm.o being created. Linking is suppressed by the **-c** option.

The Charm compiler translates Charm code to C constructs and runtime system interface code. Currently, some errors in the Charm program are detected by the C compiler; in such cases the error message generated may not be meaningful, however, the line number in the error message can be used to get the location of the error in the program.

The Charm linker takes in a list of object files and Charm runtime options, and produces an executable file. For example:

charmc -g -balance mngr pgm1.o pgm2.o -o pgm

This links pgm1.o and pgm2.o with the Charm runtime to produce the executable pgm, where pgm1.o and pgm2.o can be either Charm object files or C object files. The Charm runtime system options are those arguments that select the behavior of the Charm runtime system. Those arguments and their default values are -balance rand, -queue fifo, -execmode notrace, and -memory table. The -machine attribute is used to select the target machine. It is necessary only for sites having more than one machine type installed in the same directory. For example, if uniprocessor-sun and network-sun are installed on the same workstation, the -machine attribute is used to select the target machine. The default target machine is the machine that was installed first.

Section A.5 describes each of the runtime options.

A.3 Executing Charm programs

The Charm linker produces one executable file. On machines with a host (such as a network of workstations), a link to the proper host program **charmhost** is created in the user program directory. Sample execution examples are given below (the executable is called pgm). Exact details will differ from site to site. The list of Charm command line options is in Section A.4.

- **CM-5:**

  pgm

  will run program pgm interactively from one of the partition managers. For programs with large resource requirements, use the utilities **jrun** and **jstat**. See the man pages at your site for more information.

- **nCUBE/2:**

  xnc -d4 pgm

  runs pgm on a hypercube of dimension 4 (i.e. on 16 processors).

- **Paragon:**

  pexec pgm -sz 4

  runs pgm on four processors.

- **Network of workstations:**

  charmhost pgm +p4

  executes pgm on 4 nodes. In a network environment, Charm must be able to locate the directory of the executable. If all workstations share a common file name space this is trivial. If they don't, Charm will attempt to find the executable in a directory with the same path from the **$HOME** directory. Pathname resolution is performed as follows:

  1. The system computes the absolute path of **pgm**.
  2. If the absolute path starts with the equivalent of **$HOME** or the current working directory, the beginning part of the path is replaced with the environment variable **$HOME** or the current working directory.
  3. The system tries to locate this program (with modified pathname) on all nodes.

  The list of nodes must be specified in a file, with one node entry per line, each entry being in the format:

  nodename [username] [passwd] [setup_command]

  The nodename must be specified completely (e.g. sparc1.cs.uiuc.edu). The username is optional; if the login id on that node is the same as the current login id, it need not be specified. The password argument is ignored in the current implementation. The setup_command is optional; if specified, the command is executed on the corresponding node before the program begins execution. The * character may be used as a placeholder for an absent argument. The number of nodes specified in the nodes file must not be less than the number of nodes specified with the **+p** command line option. If the number of nodes specified in the nodes file is less than the number of nodes specified with the **+p** option, **+p** entries in the nodes file will be used in succession, starting with the first entry in the file. The name of the nodes fil itself is obtained by **charmhost** in the following order:

1. From the command line option **++nodesfile**.
2. If the **++nodesfile** option is not given, the value of the environment variable **NODES** is taken to be the nodes file.
3. If the environment variable **NODES** is not set, the file .nodes in the user's home directory is taken to be the nodes file.
4. If the above file does not exist, the file .nodes in the current directory is used as the nodes file.

The user is required to set up remote login permissions on all nodes using the .rhosts file in the home directory.

Note that the Charm linker will provide the correct executable. The user, however, needs to know how programs are run for the particular machine.

A.4 Command Line Options

A Charm program accepts the following command line options:

+pN Run the program with N processors. The default is 1. Note that on some nonshared memory machines, e.g., nCUBE/2, the user must specify the number of processors using the command provided for that machine (e.g. **xnc** on the nCUBE/2). In such cases the **+p** option is ignored.

+mM Run the program with M Kwords of memory per processor. The default is 50 Kwords of memory per processor.

+mmM Run the program with M Kwords of memory for processor 0.

+ss Print summary statistics about chare creation. This option prints the total number of chare creation requests, and the total number of chare creation requests processed across all processors.

+cs Print statistics about the number of create chare messages requested and processed, the number of messages for chares requested and processed, and the number of messages for branch office chares requested and processed, on a per processor basis. Note that the number of messages created and processed for a particular type of message on a given node may not be the same, since a message may be processed by a different processor from the one originating the request.

+mems Print the Memory Usage Statistics at the end, including the number of memory allocation requests and memory free requests, based on the size of the memory allocated or freed.

user_options Options that are be interpreted by the user program may be included after all the system options. However, **user_options** cannot start with +. The **user_options** will be passed as arguments to the user program via the usual **argc/argv** construct to the **main** entry point of the main chare. Charm system options will not appear in **argc/argv**.

A.4.1 Additional Uniprocessor Command Line Options

The uniprocessor versions can be used to simulate multiple processors on a single workstation. Any number of processors between 1 and 32 can be simulated by using the **+p** option, limited only by the available memory on the uniprocessor workstation. By default, the uniprocessor versions handle a single message from each processor, going in order from processor 0 thru $P-1$ (where P is the number of processors) repeatedly. If the user supplies the **+seed** command line option followed by an integer value, the processors will be accessed in a random (but deterministic) order. **+seed** is only recognized by the uniprocessor version.

A.4.2 Additional Network Command Line Options

The following ++ command line options are available in the network version:

++debug Run each node under gdb in an xterm window, prompting the user to begin execution.

++debug-no-pause Run each node under gdb in an xterm window immediately (i.e. without prompting the user to begin execution).

++maxrsh Maximum number of **rsh**'s to run at a time.

++resend-wait Timeout before retransmitting datagrams (in msec).

++resend-fail Timeout before retransmission fails (in msec). This parameter can help the user kill "runaway" processes, which may not be killed otherwise when the user interrupts the program before it completes execution. Currently a bug exists in the network version that may cause programs to terminate prematurely if this value is set too low and **scanf** operations are being performed.

++nodesfile File containing list of nodes.

If using the **++debug** option, the user must ensure the following:

1. **xterm** and **gdb** must be in the user's path.
2. The path must be set in the **.cshrc** file, not the **.login** file, because **rsh** does not run the **.login** file.
3. The nodes must be authorized to create windows on the host machine (see man page for **xhost**).

A.5 Queueing and Load Balancing Strategies

A.5.1 Queueing Strategies

Queueing strategies are used to schedule messages that are ready to be executed. The user can select from several queueing strategies. The prioritized queueing strategies require at least 50K words of memory per processor — for shared memory machines you would need to increment the size of the memory allocation only on node 0 to the total requirement, while on nonshared memory machines the memory requirement of each node must be incremented by 50K. Runtime commands to increment memory are discussed in Section A.4. The following queueing strategies are available:

1. **stack:** The queue is managed as *LIFO*.
2. **fifo:** The queue is managed as *FIFO*.
3. **fl (fifo-lifo):** The queue is managed as *FIFO* or *LIFO* depending on the length of the queue (with shorter queues managed in a *FIFO* manner).
4. **bstack (bit vector stack):** A bit vector queueing strategy in which elements are picked according to the highest priority, and elements with the same priority are picked as *LIFO*.
5. **bfifo (bit vector fifo):** A bit vector queueing strategy in which elements are picked according to the highest priority, and elements with the same priority are picked as *FIFO*.
6. **istack (integer stack):** An integer queueing strategy in which elements are picked according to the highest priority, and elements with the same priority are picked as *LIFO*.
7. **ififo (integer fifo):** An integer queueing strategy in which elements are picked according to the highest priority, and elements with the same priority are picked as *FIFO*.

A.5.2 Load Balancing Strategies

The following load balancing strategies are available for nonshared memory machines.

1. **rand (random):** Newly created chares are sent to random nodes.
2. **acwn (Adaptive Contracting Within Neighborhood):** Newly created chares are balanced within intersecting neighborhoods of processors.
3. **mngr (Manager):** This strategy works with a central manager to which all new chares are sent; it is the task of the manager to re-distribute the chares amongst other processors. The strategy is meant for use with prioritized task creation; however it can be used for applications without prioritized tasks.
4. **tok (Token):** This strategy is a more sophisticated load balancing strategy for prioritized task creation.

A.5.3 Modes of Execution

A Charm program can be executed in many different modes:

1. **notrace:** This mode of execution is the default, and produces no trace information about the execution of the program.
2. **summary:** This mode of execution produces summary traces of the execution of the program. They can be viewed using **SummaryTool**.
3. **projections:** This mode of execution produces traces for the **Projections** performance tool.
4. **record:** In this mode of execution, the system produces a record of the program's execution.
5. **replay:** The execution of a Charm program which has been recorded before using the **record** mode of execution, can be replayed using the **replay** mode of execution. In this mode, the program is re-executed on the basis of a recorded trace. The user selection of load balancing and queueing strategies are ignored in this mode of execution.

The current release may not support all of these modes on all machines. In particular, the record and replay modes of execution are not currently supported on shared memory machines.

B Charm System Types

```
typedef int INT16;
typedef INT16 PeNumType;
typedef INT16 ChareNumType;
typedef INT16 EntryPointType;
typedef unsigned char BOOLEAN;

typedef struct chare_id_type {
        PeNumType onPE;
        BOOLEAN isBOC;
        union chareboc {
             int chare_magic_number;
             ChareNumType boc_num;
        } chare_boc;
        union idblock {
             struct chare_block * chareBlockPtr;
             struct vid_block * vidBlockPtr;
        } id_block;
} ChareIDType;


typedef int (*FUNCTION_PTR)();


typedef INT16 AccIDType


typedef INT16 MonoIDType


typedef INT16 WriteOnceID;
```

C Changes in this Version

This version of Charm (4.3), differs from the previous version (4.1) in the following ways:

1. The **StartQuiescence** command has been added and is now necessary to use the quiescence detection feature of Charm. See Section 2.4. **Note: The addition of this feature requires existing programs that employ quiescence detection to be modified.**

2. Procedures for compiling and executing Charm programs have been simplified by the **charmc** script. See Section A.2.

3. Versions for the Thinking Machines CM-5, the IBM RS-6000, the IBM SP-1, and Hewlett-Packard workstations are now available.

4. Miscellaneous bug fixes and code improvements have been made.

D Common Programming Mistakes in Charm

1. **CkFreeMsg**, not **CkFree**, must be used to free a message.
2. Not reallocating a message when calling **SendMsg()** or other message calls which assume control of the message after the call.
3. Referencing a freed message.
4. Misunderstanding behavior of **CkExit()** and **ChareExit()** routines. In particular, execution does not stop immediately after calling these routines.
5. Failure to specify .nodes file on versions that use it.
6. Forgetting to include .interface files to other modules, or forgetting to preface identifiers from other modules with `ModuleName::`.
7. Assuming that Charm is a superset of C (e.g. Charm does not accept global variables or static declarations).
8. In a pack function, you must allocate a pack buffer. In an unpack function, you should not free the pack buffer.
9. For initializing a readonly variable that is a message, you must call **ReadMsgInit** instead of **ReadInit**.

E Known Bugs/Problems in Version 4.3

This version of Charm (4.3), has the following known bugs/problems:

1. **CkPrintf** statements are limited to 250 characters.
2. **CkScanf** statements are limited to 250 characters and 16 operands.
3. Some Charm keywords (e.g. **CreateBoc**) may be altered by the translator if they appear inside string literals (i.e. inside quotations).
4. All functions declared in a module may need to be forward declared as static.
5. The translator does not accept an extra comma at the end of a list, although this is legal C.
6. Only one word may appear inside the sizeof operator. The workaround is to use typedef.
7. Multiple consecutive *'s (as in pointers to pointers) are not parsed correctly by the translator. Again, the workaround is to use typedef.
8. When using multiple Charm source files, the interface exported by one module and imported by another module must agree exactly (i.e. must look identical).
9. Some operating system header files (e.g. varargs.h) are rejected by the Charm translator. The workaround is to include these header files in C source files instead of Charm source files.

F Further Information

This section lists some important documentation and how it may be obtained as well as how to obtain the Charm system itself.

F.1 Related Publications

1. For starters, *Reports and Publications: Parallel Programming Laboratory* catalogs all of the major papers written about Charm and its associated tools.

2. An overview of some of the issues involved in parallel programming and the Charm approach to addressing those issues is discussed in *Parallel Programming with CHARM: An Overview*. Also discussed are some of the more recent projects that have been developed by the Charm group.

3. The report *Tutorial: Introduction to Charm* is also available with the distribution of Charm, which introduces Charm via a series of examples.

4. For a design philosophy of the language, see the report *The Design Philosophy of Chare Kernel Parallel Programming System*, University of Illinois Department of Computer Science Report No. UIUCDCS-R-89-1555, November 1989.

5. The *Dagger Language Tutorial* gives further information on Dagger which was introduced in Section 8.

F.2 Associated Tools

Some tools which may also be of interest follow. See the Charm distribution sites (Section F.3) for information on obtaining them.

Dagger is described in this document in Section 8.

DagTool is a visual frontend to Dagger.

Projections is a performance visualization and feedback tool. The system has a much more refined understanding of user computation than is possible in traditional tools. Thus it can provide much specific feedback. Future versions will be able to provide recommendations/suggestions for improving performance.

Charm++ is a C++ based parallel language with all features of Charm, which supports multiple inheritance, late bindings, etc. for parallel objects too. It is documented in the *Charm++ Programming Language Manual*.

F.3 Contacts

The system is available (for research use and evaluation) by anonymous ftp, from a.cs.uiuc.edu, under the directory pub/CHARM. The files there include a manual, installation scripts, a README file that explains how to install the system on your machine, object files for supported machines, and many of the papers described above.

The above files are also available via the Charm Gopher server which runs on *dagger.cs.uiuc.edu*.

Further information about Charm is available via the World Wide Web server for the Charm group (accessible using **Mosaic**) at **http://charm.cs.uiuc.edu**.

If retrieval of a publication via these channels is not possible, please send electronic mail to **woodwort@cs.uiuc.edu** or postal mail to:

Joyce Woodworth
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, IL 61801

While we can promise neither bug-free software nor immediate solutions to all problems, Charm is a stable system and it is our intention to keep it as up-to-date and useable as our resources will allow by responding quickly to questions and bug reports. To that end, there are mechanisms in place for contacting Charm users and developers.

Discussion about Charm occurs through the Charm news group, **uiuc.cs.charm**, and the Charm mailing list. To join the Charm mailing list, send an electronic mail message to **listserv@cs.uiuc.edu** which contains the single line **subscribe charm <full-name>** (where your name is used in place of <full-name>). After receiving acknowledgement of your subscription to the list, mail to **charm@cs.uiuc.edu** will reach all subscribers of the list.

Finally, send bug reports to **charmbugs@cs.uiuc.edu**.

Index

AccIDType, 24, 72
Accumulate(), 24-25
accumulator, 23
Accumulator library, 55
ACWN load balancing strategy, 60, 70
Add(), 56

Barrier library, 53
bit vector priorities, 36
BOOLEAN, 72
BranchCall(), 19
BranchOffice chare, 8, 18
BroadcastMsgBranch(), 19

chare, 5
ChareExit(), 15
ChareIDType, 9, 72
ChareNumType, 72
charmc, 65
CharmInit(), 6, 19, 22, 24, 27
CkAlloc(), 11
CkAllocMsg(), 11
CkAllocPackBuffer(), 34
CkAllocPrioMsg(), 12
CkCopyMsg(), 13
CkExit(), 15
CkFree(), 13
CkFreeMsg(), 14
CkHTimer(), 16
CkMakeFreeCharesMessage(), 63
CkMeltFreeCharesMessage(), 63
CkPrintf(), 14
CkPriorityPtr(), 37
CkPrioritySize(), 37
CkScanf(), 14
CkSendLdbMsg(), 63
CkTimer(), 16
CkUTimer(), 16
cluster, 60
Collect(), 56
CollectValue()
combinefn, 24
command line options, 68
compiling Charm programs, 65
COPY_AND_SEND_TO, 63
copying messages, 13
counter, 56
Create(), 49, 53, 56
CreateAcc, 24
CreateBoc(), 18
CreateChare(), 5, 9
CreateMono(), 27
CreateOverGroup(), 49
CreateRootGroup(), 46
CreateRootGroupMsg(), 46

dag chare, 42
Dagger, 42
Deallocating Memory, 13
++debug, 69
++debug-no-pause, 69
Decrement, 43
DepositData(), 53
DepositDataMsg(), 52
DerefWriteOnce, 29-30
dynamic tables

entry, 42
entry point, 6
EntryPointType, 9, 72
execmode, 71
executing Charm programs, 67
Expect, 43

FunctionNameToRef(), 15
FUNCTION_PTR, 72
FunctionRefToName(), 16

GetRefNumber(), 44, 47

histogram, 56
Horizon Effect, 60

Information Sharing Abstractions, 22
initfn, 24
installation, 64
INT16, 72
integer priorities, 36
interface between modules, 39
interface with C, 39

LDB_ELEMENT, 59
LdbFillLDB(), 61

LdbInit(), 62
Ldb_NewMsg_FromLocal(), 61
Ldb_NewMsg_FromNet, 61
LdbPeriodicCheckInit(), 61
LdbProcessMsg(), 62
LdbProcessorIdle(), 61
LdbRecvUpdateStatus(), 62
LdbSentUpdateStatus(), 62
LdbStripLDB(), 61
LDB_UPTR, 62
libraries, 45
load balancing functions, 61
load balancing strategies, 59
Load balancing strategies, 70
load information, 59

main, 69
MainChareID(), 10
MANAGER load balancing strategy, 60, 70
++maxrsh, 69
McGetNodeNeighbours(), 62
McMyPeNum(), 62
McNumNeighbours(), 62
McNumSpanTreeChildren(), 20
McSpanTreeChild(), 20
McSpanTreeParent(), 20
McSpanTreeRoot(), 20
McTotalNumPe(), 62
modules, 39
MonoIDType, 27, 72
monotonic, 25
MonoValue(), 27
msg_macros.h, 62
Multicast(), 47
MyBocNum(), 20
MyBranchID(), 20
MyChareID(), 10

network command line options, 69
NewValue(), 27
.nodes, 68-69
nodes file, 68-69
++nodesfile

pack, 33-34
Partition(), 46
PeNumType, 9, 72
PGEN_GET_NEW_PVECTOR(), 38
PgGroupSize(), 48
PgMyNumSpanTreeChildren(), 48
PgMySpanTreeChildren(), 48
PgMySpanTreeParent(), 48
piggybacked, 59
prioritized messages, 12
priority, 36
private function, 6, 16, 19, 42
PrivateCall(), 16
Process groups, 45, 52
public function, 18-19, 39, 42

Qs_DeQMsg(), 62
Qs_EnQMsg(), 62
QsEnqUsrMsg(), 63
QsMyLoad(), 62
QsPickFreeChareMsg(), 63
Queueing strategies, 70

RANDOM load balancing strategy, 60, 70
ReadInit(), 22
ReadMsgInit(), 22
readonly, 22
ReadValue(), 22
Ready, 43
RECEIVER load balancing strategy, 60
Reduction library, 49
reference number, 43
++resend-fail, 69
++resend-wait, 69

+seed, 69
SEND_FIXED_TO, 63
SendMsg(), 6, 10
SendMsgBranch()
SEND_TO, 63
Set, 43
SetLdbMsgParameters(), 63
SetRefNumber(), 44
Signal(), 53
SignalMsg(), 53
spanning tree calls, 20
StartQuiescence(), 15
Synchronize(), 47

TblDelete(), 31
TblFind(), 31
TblInsert(), 31
TOKEN load balancing strategy, 60, 70

uniprocessor command line options, 69
unpack, 33-34
USR_MSG_LDB_PTR, 62

variable sized messages, 11
varSize, 12, 33

when block, 43
WriteOnce, 27
WriteOnceID, 29, 72