The
Charm++ (4.3)
Programming Language
Manual

September 8, 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. The current version, Charm(4.0), has been developed by Attila Gursoy, Sanjeev Krishnan, Amitabh B. Sinha and Laxmikant Kale.

Contents

1 Introduction

This manual describes CHARM++, an object oriented portable parallel programming language based on C++. Its program structure and execution model, language constructs and runtime system calls are described[1]. The current version of CHARM++ has some minor syntax changes as compared to the previous version. This manual overrides older syntax descriptions (such as those in the OOPSLA 1993 paper).

CHARM++ is an explicitly parallel language consisting of C++ with a few extensions. It provides a clear separation between sequential and parallel objects. The execution model of CHARM++ is message driven, thus helping one write programs that are latency-tolerant. CHARM++ provides specific modes for sharing information between parallel objects. Extensive dynamic load balancing strategies are provided. CHARM++ supports both irregular as well as regular, data-parallel applications. It is based on the CHARM parallel programming system, and its runtime system implementation reuses most of the runtime system for CHARM. For a description of the C-based CHARM parallel programming system, please refer to the CHARM programming language manual and the tutorial introduction to CHARM.

All software and documentation related to the CHARM system are available by anonymous ftp from a.cs.uiuc.edu (128.174.252.1), in the directory /pub/CHARM.

Currently the parallel platforms supported by CHARM++ are the TMC CM-5, nCUBE/2, Intel Paragon, Sequent Symmetry, and a single workstation or network of workstations. CHARM++ programs can run without change on all these platforms.

[1]For a description of the underlying design philosophy please refer to the OOPSLA 1993 paper on CHARM++.

2 The CHARM++ language

2.1 Structure of a CHARM++ program

A CHARM++ program is structurally similar to a C++ program. Many components of a CHARM++ program *are* C++ code fragments [2]. The main syntactic units in a CHARM++ program are function and class definitions. A CHARM++ program can possibly be distributed across several source code files.

There are five disjoint categories of objects (classes) in CHARM++ :

- Sequential objects : as in C++
- Chares [3] (concurrent objects)
- Branched chares (a form of replicated objects)
- Shared objects (specific information sharing abstractions)
- Messages (communication objects)

2.2 Basic CHARM++ constructs

This section describes the syntax of message, chare and branched chare definitions. Please see Section 5 for a description of shared objects.

Messages : Message definitions are similar to structure definitions in C++.

```
message MessageType {
    // List of data members as in C++
} ;
```

A message pointer has type `(MessageType *)` as in C++, and can be used as a normal pointer. Messages may optionally have two function members : *pack* and *unpack* (see Section 6), which must be defined by the user[4].

Chare classes : Chares are concurrent objects. A chare definition has the form

```
chare class ChareName [: superclass names] {
    // Private data and member functions as in C++
    // One or more entry point definitions of the form
 entry:
```

[2]**The C++ code cannot, however, contain global or static variables.**

[3]Chare (pronounced **chär**, ä as in cart) is Old English for chore.

[4]Messages can be extended to general classes, with data and function members, and inheritance. This extension may be incorporated in a future version of CHARM++.

```
    EntryPointName1(MessageType1 *MsgPointer)
        { C++ code block }
    EntryPointName2(MessageType2 *MsgPointer)
        { C++ code block }
} ;
```

Chares are medium grained units of work. Chares can be dynamically created on any processor ; there may be thousands of chares on a processor. The location of a chare is usually determined by the dynamic load balancing strategy ; however, once a chare commences execution on a processor, it does not migrate to other processors. Chares do not have a "thread of control" : the entry-point functions in a chare execute in a message driven fashion upon arrival of a message.

The entry point definition specifies a function that is executed *without interruption* when a message is received and scheduled for processing. Only one message per chare is processed at a time. Entry points are defined exactly as functions, except that they cannot return values and they have exactly one argument which is a pointer to a message. Currently entry points cannot be overloaded (i.e. there cannot be two entry points of the same name in a chare definition).

Each chare instance is identified by a *handle* which is unique across all processors. The handle of a chare of type `ChareName` has type (`ChareName handle`). Chares can be accessed only through their handles. The user can construct inheritance hierarchies of chares ; however, currently private and protected base chare classes are not supported.

Branched chare classes : These are modeled after the "branch office chares" of CHARM. A branched chare is a special chare that has a branch on each processor, with each branch having its own data members. Branched chares can have public data and function members in addition to entry points and private members. A branched chare is identified by a unique handle that is common to all its branches. Branched chares have a definition syntax similar to normal chares :

```
branched chare class ChareName [: superclass names ]
{   // Private data and member functions as in C++
    // Public data and member functions
    // Entry points as for normal chares
} ;
```

Sequential objects, normal chares and other branch chares can access public members of the branch of a branched chare *on their processor* by
`ChareHandle[LOCAL]->DataMember` and
`ChareHandle[LOCAL]->FunctionMember()`. Thus a dynamically located chare can call a public function of a branched chare without needing to know which processor it is itself on : the function call executes in the local branch of the branched chare.

Branched chares can be used to implement data-parallel operations easily. Messages can be broadcast to all branches of a branched chare as well as sent to a particular branch.

There can be many instances corresponding to a branched chare type : each instance has a different handle and its set of branches on all processors.

2.3 CHARM++ Execution Model

Every CHARM++ program must have a chare type `main`. There can be only one instance of this chare type, which executes on a single system selected processor. This must have the public function `main`. Execution of a CHARM++ program begins at this function. Typically, this function starts the computation by creating chares and branched chares. It can also be used to initialize shared objects.

CHARM++ program execution can be terminated by the *CharmExit()* call. The user can specify an entry point (using the *CStartQuiescence()* call, see Section 4) to be called when the computation has become quiescent (i.e., when no processor is executing an entry point and all messages that have been sent have also been consumed). This entry point is useful in computations where the user cannot forsee when the program is going to become quiescent. It is also useful for computations which proceed in phases — the quiescence entry point can be used to start new phases of computation.

The basic unit of parallel computation in CHARM++ programs is the chare. A chare is similar to a process, an actor, an ADA task, etc. A chare is created by the new_chare() system call. At runtime, many chares may be active. Chares may send messages to one another. Conceptually, the system maintains a "work-pool" consisting of seeds for new chares (which are created when new_chare() calls are made), and messages for existing chares. The runtime system may pick multiple items, non-deterministically, from this pool and execute them. It will not process two messages for the same chare concurrently, but otherwise it is free to schedule them in any way.

3 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, allocate memory and free memory. Various other calls for input-output and exiting from CHARM++ programs are also described in this section.

If a CHARM++ call returns a value, the value must be cast to the proper type.

3.1 Creating Chares

The call *new_chare* permits a user to create a new instance of a chare. The syntax of the call appears below :

new_chare (ChareName, &(ChareName::EntryPoint), MsgPtr [, &VirtualHandle [, DestPE]])

The call deposits the *seed* for a chare in a pool of seeds and returns immediately. The chare will be created later on some processor, as determined by the dynamic load balancing strategy.

The call can have three, four or five parameters as shown above. The parameters are described below.

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

EntryPoint: The entry point in the chare from where the execution is to start, after creation of the chare. The value expected for this parameter is a pointer to a valid entry point in chare type ChareName. Entry point pointers may be assigned to variables of the system defined type EntryPointType ; these variables can then be passed to other functions (even possibly across processors) and used wherever an entry point pointer can be used.

MsgPtr: A pointer to a message of the type associated with EntryPoint in chare ChareName.

&VirtualHandle: This is a pointer to a handle which is set by the *new_chare* routine. This optional argument can be used if the user desires to have a *virtual* handle to the instance of the chare that will be created. This handle is useful for sending messages to the chare, even though it has not yet been created on any processor. Messages sent to this virtual handle are either queued up to be sent to the chare after it has been created, or simply redirected if the chare has already been created. For performance reasons, therefore, virtual handles should be used only when absolutely necessary. Virtual handles are otherwise like normal handles, and may be sent to other processors in messages.

DestPE: When a chare is to be created at a specific processor, the optional parameter DestPE is used to specify that processor. If no virtual handle is required to be returned, the fourth parameter can be *NULL_HANDLE*. 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 branched chares should be used.

3.2 Creating Branched Chares

The call *new_branched_chare* permits a user to create a new instance of a branched chare. The syntax of the call appears below :

new_branched_chare (ChareName, &(ChareName::EntryPoint), MsgPtr [, ReturnEP, ReturnHandle])

This call creates a branch of the branched chare on every processor and executes the entry point EntryPoint in that branch. The first three arguments are similar to the first three arguments of the *new_chare* call. The MsgPtr argument is the message that is broadcast to all branches.

Branched chares are usually created in the main entry point of the main chare, in which case the *new_branched_chare* call returns the handle of the newly created branched chare. When branched chares are dynamically created (from outside the main entry point), the handle is returned in a message to the entry point ReturnEP in the chare whose handle is ReturnHandle. The ReturnEP must accept a message which has its first element as a ChareName handle.

The ReturnEP–ReturnHandle mechanism : Since all CHARM++ calls are non-blocking, calls that require remote computations cannot return values immediately. Instead, the user is required to specify a ReturnEP–ReturnHandle pair, which is like a *continuation*, to which the value is sent later in a message. This mechanism will be used in many calls.

3.3 Sending Messages

A message may be sent to a chare using the notation :

ChareHandle=>EntryPoint(MessagePointer)

This sends the message in MessagePointer to the chare whose handle is ChareHandle at the entry point EntryPoint, which must be a valid entry point of that chare type. This call is asynchronous and non-blocking ; it returns immediately after sending the message. The EntryPoint could also be a variable of type *EntryPointType*. Currently ChareHandle cannot be a field inside a message; it must be a handle variable declared in some enclosing scope.

A message may be sent to a particular branch of a branched chare using the notation :

ChareHandle[Processor]=>EntryPoint(MessagePointer)

This sends the message in MessagePointer to the branch of the branched chare ChareHandle on processor number Processor at the entry point EntryPoint, which must be a valid entry point of that chare type. This call is asynchronous and non-blocking ; it returns immediately after sending the message.

A message may be broadcast to all branches of a branched chare (i.e., to all processors) using the notation :

ChareHandle[ALL]=>EntryPoint(MessagePointer)

This sends the message in MessagePointer to all branches of the branched chare ChareHandle at the entry point EntryPoint, which must be a valid entry point of that branched chare type. This call is asynchronous and non-blocking ; it returns immediately after sending the message.

Note that the programmer relinquishes control of a message after sending it : further access to the message field can cause runtime errors.

Currently, the handles must be variables in an enclosing scope ; they cannot be fields inside aggregate types. This restriction may be removed in a future version.

3.4 Chare Addresses

The keyword thishandle holds the handle of the chare whose entry point or public function is currently executing. It can be used to set a handle variable, which may be in a message. This mechanism allows chares to send their handles to other chares. The keyword mainhandle holds the handle of the single instance of the main chare.

3.5 Message Creation and Deletion

Messages are allocated using the call

new_message(MessageType [, sizes_array])

This call allocates space for a message of message type MessageType. If successful a pointer is returned to the allocated structure; otherwise it returns NULL. This call is different from the C++ *new* call in that extra information is kept by the system to facilitate the handling of the message by other system routines. Any structure to be used as a message and passed to system routines **must** be allocated using *new_message*.

In Section 6.1 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 keyword VARSIZE to declare messages with *single dimensional* arrays whose sizes may be known only at run time when the message is allocated. [5] The translator generates code to handle the pointers (so as to maintain validity) for such messages. The optional parameter sizes_array in the *new_message* call is a pointer to an array of integers, and is used to specify variable sized messages. An example of usage of a variable sized message appears below:

[5] The array element type must not be a pointer type, i.e., an array of pointers cannot be a variable sized array. To use messages with multi-dimensional arrays and arrays containing pointers, see Section 6.1.

```
message MSG {
    int size1, size2;
    float field1[VARSIZE];
    float field2[VARSIZE];
} ;
...
...
int sizes[2];

sizes[0] = Size1;
sizes[1] = Size2;
MSG *msg = (MSG *) new_message(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;
```

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

void *new_prio_message(MessageType, PrioSize [, sizes_array])

The *new_prio_message* call is specifically for allocating space for messages that have **integer** or **bit-vector** priorities. The call allocates a message of type MessageType 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 *new_message* 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 *new_prio_message*. See Section 6 for more details on different types of priorities and how to access them. The usage of the optional parameter sizes_array is same as that of non-prioritized variable sized messages.

When a message is sent to a chare, the programmer relinquishes control of it ; the space allocated to the message is freed by the system. When a message is received at an entry point it is not freed by the runtime system. It may be reused or deleted by the programmer.

Messages can be deleted using the call :

void delete_message(MessagePtr)

This call call frees the space associated with a message. The space should have been originally allocated with *new_message* or *new_prio_message*. The routine takes a pointer to the user message as a parameter.

3.6 Input/Output Primitives

The CHARM++ input / output calls are similar to their C or C++ counterparts, except that they guarantee atomicity, e.g. outputs from two output calls on two different processors will not get interleaved.

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

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

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

CScanf 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 may be specified.

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

3.7 CHARM++ Exit Calls

void ChareExit()

The *ChareExit* call is used to to release any memory allocated for the local variables in the chare. It does not stop program execution at the point of call ; hence the user must ensure that no access is made to chare-local variables after the *ChareExit* call. At present the runtime does not handle messages sent to chares that have exited by making a call to *ChareExit*; these messages constitute runtime errors, and may lead to unpredictable behavior.

void CharmExit()

Similar to the regular Unix *exit*, except that this routine will kill all running processes and then exit itself. The *CharmExit* routine does not stop execution of the program in 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).

3.8 Quiescence Detection

The call *CStartQuiescence* permits a user to detect quiescence. The syntax of the call is :

CStartQuiescence (&(ChareName::EntryPoint), ChareHandle)

Quiescence is defined as a state in which no processor is executing an entry point and all messages that have been sent have also been consumed.

When quiescence occurs, the system will send a message to the entry point EntryPoint in the chare of type ChareName whose handle is ChareHandle. The message is of type QUIESCENCE_MSG, which is predefined by the system in the file `quiescence.h`. After quiescence is detected, the user can start the next phase of the computation, and if necessary, detect quiescence again by making another *CStartQuiescence* call. This can be done indefinitely many times if required.

3.9 Function Addresses

In C++, 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 go across 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 *CFunctionNameToRef* and *CFunctionRefToName*. They are described below.

FunctionRefType CFunctionNameToRef(function-name)

The system call *CFunctionNameToRef* takes a function name, 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 *CFunctionRefToName* system call to get the local address (on the calling processor) of the function, function-name.

CFunctionRefToName((FunctionRefType)function-ref)

The system call *CFunctionRefToName* takes a function reference, function-ref, as parameter and returns a pointer to the function corresponding to that reference index. Currently the above two calls are supported only for global functions.

3.10 Other Calls

int CTimer()

unsigned int CUTimer()
int CHTimer()

The *CTimer*, *CUTimer* and *CHTimer* calls are used to time programs. *CTimer* and *CUTimer* return the time since the process began execution on that node. *CTimer* returns the time in milliseconds, while *CUTimer* 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 CPrintf, instead of %d. Since, *CUTimer* rolls overs for an execution of 1 hour in our implementation, a *CHTimer* call is provided to give number of hours since the program started executing. This call should be used in conjunction with the *CUTimer* call. *CHTimer* returns -1 on machines on which it hasn't been implemented.

int CMaxPeNum()

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

int CMyPeNum()

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

3.11 Spanning Tree System Calls

CHARM++ provides a default organization of processors as a logarithmic height spanning tree. This is useful for implementing global operations with branched chares. The spanning tree routines are used to access details of the spanning tree formed by the nodes in the system. Node 0 is the root of the spanning tree (message communication patterns can be made to proceed along the spanning tree arcs to avoid bottlenecks at a single node).

int CSpanTreeRoot()

CSpanTreeRoot returns the processor number of the root of the spanning tree.

int CSpanTreeParent(int proc)

CSpanTreeParent returns the processor number of the parent (in the spanning tree) of processor proc.

int CNumSpanTreeChildren(int proc)

CNumSpanTreeChildren is used to determine the number of spanning tree children of processor, proc.

CSpanTreeChild(int proc, int *children)

CSpanTreeChild is used to determine the children in the spanning tree for a processor, proc. The integer array children is filled with the processor numbers of all children.

4 Information Sharing Abstractions

Messages are too general and low level a mechanism for information sharing between chares. Moreover, communication overheads can be optimized if the pattern of information sharing is known. Hence CHARM++ provides specific abstract data types, referred to as *specifically shared variables*, for sharing data amongst chares. 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 which is obtained only after the program begins execution, and which does not change subsequently. These variables can be accessed from any chare on any processor ; the cost incurred is that of a local reference. Read only variables can be declared only at file scope. Read only declarations have the following form :

readonly Type readname;
readonly MessageType *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 a read only message of type MessageType. Pointers are not allowed to be read only variables, unless they are pointers to message types, in which case the message will be correctly initialized on every processor.

Read only variables and messages can be assigned values only in the main function of the main chare. The value of a read only variable is the value it has at the end of the main entry point. The assignment to a read only variable is like a normal variable assignment. In case of read only messages, the user must allocate space for the message using the *new_message* call and initialize the message.

Read only variables can be used after initialization like normal variables. Currently, the compiler does not strictly disallow assignments to read only variables ; it is the user's responsibility to make sure that no assignments occur.

4.2 Accumulator Variables

The accumulator type is like a counter – it can be used to accumulate results from chares when the result of the accumulation is needed only once (perhaps, at the end of computation). It is optimized for fast update : the cost incurred is that of a local reference.

The accumulator abstract data type is a class with only the following members :

- one data member : a message containing the data area of the accumulator data type
- a constructor. This must allocate space and initialize the above message to the zero for the user defined metric the accumulator is used for. The constructor has exactly one argument : a message which is used to carry the initialization data to all processors.

- an Accumulate() function : this operator "adds" to the accumulator variable in the user defined metric. It may have arbitrary arguments. Currently, the code for the Accumulate() function must be within the accumulator definition.
- a Combine() function : this operator "combines" values from two processors. It has exactly one argument which is a message of same type as the data member. This function is only called by the system. The Accumulate and Combine operators must be commutative-associative.
- a CollectValue() system defined function : this can be called by the user to obtain the final result of the accumulation. It has two arguments : a chare handle and a pointer to an entry point in that chare which accepts a message of the same type as the accumulator's data member. The accumulator result is returned to that entry point by the system. This function is "inherited" by all accumulator objects, so it should not be listed in the accumulator definition.

```
accumulator class AccType {
    MessageType1 *accmsg ;
        // The data member
public: AccType(MessageType2 *) ;
        // Constructor, to be defined by user
    void Accumulate ( /* arguments */ ) ;
        // To be defined by user
    void Combine (MessageType1 *) ;
        // To be defined by user
} ;
```

Figure 1: Structure of an accumulator abstract data type

An accumulator variable is initialized using the *new_accumulator* call.

new_accumulator (AccType, msg [, ReturnEP, ReturnHandle])

The above *new_accumulator* call creates an accumulator of type AccType. The call results in its constructor being invoked on every node with a single parameter msg, which is of type MessageType2. The *new_accumulator* call returns the handle of the newly created accumulator if the call has been made inside the main entry point of the main chare. Otherwise optional parameters can be used to receive the handle at the entry point ReturnEP (which is specified as &(ChareName::EP)) of the chare having handle ReturnHandle. The message received can be defined by the user to have any structure, except the first field must be of type AccType handle. A note of caution: the value returned by the *new_accumulator* call in any place other than main may not be a valid accumulator handle.

Any chare can "add" to an accumulator variable by calling the *Accumulate* method.

acchandle->Accumulate ([arg [, arg]*])

The final value of an accumulator can be obtained by calling the *CollectValue* method.

acchandle->CollectValue (&(ChareName::EntryPoint), charehandle)

This call results in the message associated with the accumulator acchandle being sent to the to the chare charehandle (of type ChareName) at entry point EntryPoint. The method can be called only once for each accumulator instance; later usages will result in an error being reported.

In Figure 2, `A1` is declared to be an accumulator, which is an array of integers. The two operators are `Accumulate` and `Combine`; `Accumulate` adds to elements inside the accumulator, and `Combine` takes an accumulator variable as a parameter and combines it (element by element) with the current value of the accumulator.

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

accumulator class A1 {
    MSG1 *msg;
public:
    A1(MSG1 *initmsg)
    {
        msg = (MSG1 *) new_message(MSG1);
        for (int i=0; i<MAX; i++)
            msg->a[i] = initmsg->a[i];
    }
    void Accumulate (int y, int z)
    {
        msg->a[y] += z;
    }
    void Combine (MSG1 *othermsg)
    {
        for (int i=0 ; i<MAX; i++)
            msg->a[i] += othermsg->a[i];
    }
} ;
```

Figure 2: Example of an accumulator definition

4.3 Monotonic Variables

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

The monotonic abstract data type is a class with only the following members :

- exactly one data member : a message containing the data area of the monotonic data type

- a constructor. This must allocate space and initialize the above message to the zero for the user defined metric the monotonic is used for. The constructor has exactly one argument : a message which is used to carry the initialization data to all processors.
- an Update() function : this operator "adds" to the monotonic variable in the user defined metric. It must be idempotent, commutative, associative and monotonic. It has exactly one argument which is a message of same type as the data member. It should return 1 if the value of the variable is actually updated, and 0 otherwise. Currently, the code for the Update() function must be within the monotonic type definition.
- a MonoValue() system defined function : this can be called by the user to obtain the current value of the monotonic variable. It has no arguments. The value returned is a pointer to a message of same type as the data member. This function is "inherited" by all monotonic objects, so it should not be listed in the monotonic definition.

```
monotonic class MonoType {
    MessageType1 *monomsg ;
        // The data member
public: AccType(MessageType2 *) ;
        // Constructor, to be defined by user
    int Update (MessageType1 *) ;
        // To be defined by user
} ;
```

Figure 3: Structure of an monotonic abstract data type

An monotonic variable is initialized using the *new_monotonic* call.

new_monotonic (MonoType, msg [, ReturnEP, ReturnHandle])

The above *new_monotonic* call creates a monotonic variable of type MonoType. The call results in its constructor being invoked on every node with a single parameter msg, which is of type MessageType2. The *new_monotonic* call returns the handle of the newly created monotonic if the call has been made inside the main entry point of the main chare. Otherwise optional parameters can be used to receive the handle at the entry point ReturnEP (which is specified as &(ChareName::EP)) of the chare having handle ReturnHandle. The message received can be defined by the user to have any structure, except the first field must be of type MonoType handle. A note of caution: the value returned by the *new_monotonic* call in any place other than main may not be a valid monotonic handle.

Any chare can update a monotonic variable by calling the *Update* method.

monohandle->Update ((MessageType1 *) msg)

The current value of a monotonic variable can be obtained by calling the *MonoValue* method.

MessageType1 * currentvalue = monohandle->MonoValue ()

This call results in the message associated with the monotonic monohandle being returned. This value may not be the globally (i.e., over all processors) current value of the variable, but it is guaranteed to *eventually* be the globally current value. However, the value returned will always be monotonically greater than or equal to the values returned by previous calls. The cost of a MonoValue call is that of a local procedure call.

In Figure 4, `GMax` is declared to be a monotonic, which contains a single integer. The functionality this monotonic implements is equivalent to multiple concurrent non-blocking global "max" operations, with the global maximum being continously updated and available on all processors.

```
message MSG1 {
    int x;
} ;

monotonic class GMax {
    MSG1 *msg;
public:
    GMax(MSG1 *initmsg)
    {
        msg = (MSG1 *) new_message(MSG1);
        msg->x = initmsg->x;
    }
    int Update (MSG1 *updatemsg)
    {
        if (msg->x < updatemsg->x) {
            msg->x = updatemsg->x;
            return(1) ;
        }
        return(0) ;
    }
} ;
```

Figure 4: Example of a monotonic type definition

4.4 WriteOnce Variables

Write once variables are similar to read only variables, except that they are dynamically created. Write once variables may be thought of as untyped global objects which can be initialized once from anywhere in the program, and then referenced from all processors, with the cost of a reference being that of a local procedure call. A write once variable handle has type writeonce handle. The system call that creates a write once variable is *new_writeonce*.

void new_writeonce(void *DataPtr, int DataSize, ReturnEP, ReturnHandle)

This routine creates a write-once variable of size DataSize which has the data pointed to by DataPtr. A handle denoting the write once variable is passed back to the user at the entry point ReturnEP of the chare whose handle is ReturnHandle. The message type declared by the user for receiving the write once variable handle is irrelevant as long as the first field in the struct is a variable of type writeonce handle.

The user may access the write-once variable data on any node using the DerefWriteOnce method :

void *dataptr = whandle->DerefWriteOnce()
writeonce handle whandle;

The above call returns an untyped pointer to the actual data itself. Note that it is the user's responsibility to pass the handle (using messages or any other mechanism) to any chare that wishes to access the write once variable.

4.5 Distributed Tables

A Distributed Table is a set of entries, each entry being a record with an integer key and an untyped data field. Access to the entries are provided through the non-blocking calls *Insert*, *Delete* and *Find*. These user accessible functions of the distributed table class are defined by the system as :

```
class table {
public:
        void Insert(int key, void *data, int size_data, ReturnEP, ReturnHandle, int option) ;

        void Delete(int key, ReturnEP, ReturnHandle, int option) ;

        void Find(int key, ReturnEP, ReturnHandle, int option) ;
} ;
```

Each of the table class functions can result in a message of type TBL_MSG being sent to a specified entry point. TBL_MSG has the following structure :

```
message TBL_MSG {
        int key;
        void *data;
} ;
```

In each of the functions a message is sent as a reply only if the parameters ReturnEP and ReturnHandle are valid values. The user can choose to have no reply messages sent by setting the value of ReturnEP as NULL_EP and the value of ReturnHandle as NULL_HANDLE.

The key must be an integer. An element is said to match a request if the values of the key in the element and the key supplied by the request match. The matching element is said to have *defined* data, if the data field of the element has a non-null value. A distributed table `tablename` is declared as follows:

#include "dtable.h"

```
table tablename;
```

This declaration must be at file scope, similar to read only variables. Currently, distributed table objects can only be statically declared, there are no table handles and there is no facility to create a table instance dynamically (i.e., there is no "new_table" call).

The table class functions are described below.

tablename.Insert(int key, void *data, int size_data, ReturnEP, ReturnHandle, int option) ;

The *Insert* call is used to insert the element with key key and data data of size size_data into the distributed table tablename. In each of the following cases a message msg of type TBL_MSG is sent back to the entry point ReturnEP in the chare whose handle is ReturnHandle. The key and data fields of the message msg are set as described below :

1. *matching element already present and data is defined :* The key and data fields of msg are the same as those of the matching element.
2. *matching element already present but data is undefined :*
   - If data is not NULL, then the data field of the matching element is defined to be data, and the key and data fields of msg are key and data, respectively.
   - If data is NULL and option = $TBL\_WAITFORDATA$ then the Insert call is suspended until data is available, and the key and data fields of msg are key and the available data, respectively. Otherwise if option = $TBL\_NOWAITFORDATA$ then msg is sent immediately with the key and data fields as key and NULL, respectively.
3. *matching element not already present :* An element is inserted with the key and data fields as key and data. The data is considered undefined if data = $NULL$. If option = $TBL\_WAITFORDATA$ then the Insert call is suspended till data is available, and the key and data fields of msg are key and the available data, respectively. Otherwise if option = $TBL\_NOWAITFORDATA$ then msg is sent immediately with the key and data fields as key and NULL, respectively.

tablename.Delete(int key, ReturnEP, ReturnHandle, int option) ;

The *Delete* function is used to delete the element with key key from the distributed table tablename. If a matching element was found a message of type TBL_MSG is sent to the entry point ReturnEP at ReturnHandle with the key field as key and the data corresponding to that of the deleted element. Note that as in the case of *Insert* the message is sent only if ReturnEP and ReturnHandle are valid. If a matching element was 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*.

tablename.Find(int key, ReturnEP, ReturnHandle, int option) ;

The *Find* function is used to determine if the record with key key exists in the distributed table tablename. In some of the following cases message msg of type TBL_MSG is sent to the entry point ReturnEP at ReturnHandle. Again the message is sent only if ReturnEP and ReturnHandle are valid. The values of the key and data fields in msg are described below :

1. *a matching element exists and data is defined :* The key and data fields of msg are those of the matching element.
2. *a matching element 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 Find 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 element did not exist :* A table entry is created for the element 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 Find 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 Other CHARM++ Features

5.1 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 message. On a shared memory system, the message need only store a pointer to the base of the array and its size. However, on message passing systems (without a shared address space), 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 message passing version of CHARM++, many chares that are created are not actually sent out to any other processor, but executed locally. When the system is saturated (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, 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 in shared memory machines. However, if the runtime system decides to send a message to another processor, it calls an appropriate function 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 function 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.

Message Packing and varSize : When variable sized arrays, specified as varSize, are used in a message, the system automatically generates corresponding *pack* and *unpack* routines. Hence the varSize construct may not be used in a message type if the user defines the *pack* and *unpack* routines for that message type.

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

```
message MessageType {

    <Data member declarations>

    void *pack (int *length)
        <Pack-Function-Body>

    void unpack (void *in)
        <Unpack-Function-Body>
} ;
```

The *pack* routine takes one parameter: `length`, a pointer to the length in bytes of the packed message, which is set inside the *pack* routine. It returns a pointer to the packed message. Inside the *pack* routine the user allocates the packed message area of length `length` with the call *new_packbuffer()*. Then the user unravels all the pointers in the message, and initializes the contiguous packed message area. The *unpack* routine takes one parameter: `in`, the packed message. Inside the *unpack* routine the user re-creates the structure with pointers and sets the fields in the unpacked message.

It is important that the user free heap allocated structures in 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 *new_packbuffer* call is used to generate the contiguous message area in which the unpacked message can be packed. The syntax of the *new_packbuffer* call is as follows:

void * new_packbuffer(void *msg, int size)

The *new_packbuffer* 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.

5.2 Prioritized Execution

CHARM++ provides many strategies for managing queues of messages. Some of them (such as the *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. The 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 to high priority messages.

Priorities can be of two types:

1. **Integer Priorities** :

   This priority type uses 32 bit integers. Message with priority 0x0 is considered as the highest priority message. Priority decreases as its value increases. The lowest priority will have a value of 0x7fffffff. For example for shared memory machines, the queue strategies **istack.o** or **ififo.o** must be linked to use integer priorities. The **istack.o** queue strategy is a **priority queue** of **stacks** where messages with identical priorities are scheduled in a last-in-first-out manner. The **ififo.o** queue strategy is a **priority queue** of $fifo$ **queues** where messages with identical priorities are scheduled in a first-in-first-out manner.

2. **Bit Vector Priorities** :

   If priorities need to be longer than 32 bits then a bit vector priority may be used. For example for shared memory machines, the queue strategies **bstack.o** or **bfifo.o** must be linked to use bit vector priorities. The strategies follow the format of the corresponding integer priorities. Bit Vector Priority (or priority vector) is a sequence of binary digits of arbitrarily 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 below, thus using 24 bits $< 23 : 0 >$ in the first word.

```
First Word:

  31          24 23                                         0
  ----------------------------------------------------------------
  |Vector Length |              |                 |               |
  |in No. of Bits|              |                 |               |
  ----------------------------------------------------------------


Successive Words if required:

  31                                                        0
  ----------------------------------------------------------------
  |              |              |                 |               |
  |              |              |                 |               |
  ----------------------------------------------------------------
```

As number of bits increase then multiple long words can be allocated for 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 the bit 23 of the first word and goes left to right.

5.3 Priorities: Access and Assignment

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 **new_priomessage()** call described in Section 3.5.

PVECTOR *CPriorityPtr(msgPtr)
void *msgPtr;

The call returns 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 MSG1 {
    // Data members as usual
} ;
...
MSG1 *msg;
...
msg = (MSG1 *) new_priomessage(MSG1, sizeof(int));
x = (int *) CPriorityPtr(msg);
*x = cost;
```

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

int CPrioritySize(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 **CAllocPrioMsg()**.

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 **CAllocPrioMsg()** call and the pointer is obtained using the **CPriorityPtr()** call described above.

5.4 Multiple CHARM++ Source Files

CHARM++ programs can be distributed across multiple source files, like C++ programs. Each file can have definitions of the five types of objects in CHARM++, along with the usual sequential function definitions at file scope. CHARM++ files can be separately compiled, allowing the user to build libraries easily.

When a CHARM++ program is run on a parallel machine, invocation of an entry point on a remote processor requires an encoded version of the entry point name to be sent across address spaces, since function pointers are not valid in general on a different processor. This encoding to entry point *ids* cannot be done at compile time because each separately compiled file needs to have non-overlapping ids. In the absence of a custom linker, the CHARM++ translator generates code to do this encoding during initialization at run time.

When there is just one source file, there is no problem, and the user need not take any explicit action to support this entry point / function encoding. However, when multiple source files are present, the user is required to supply a file module.list which lists all source (extension `.P`) file names involved in the program. The file names must be listed in this file one to a line, in any order (*with* the `.P` extension). The module.list file must be present (in the directory from where the translator is invoked) when the module containing the main chare's main entry point definition is compiled. In absence of this file, the translator will assume that the program contains only one source file.

Nested Scoping

Currently, CHARM++ object definitions cannot be nested inside other C++ constructs like classes and structs. All CHARM++ chare, branched chare, message and shared object definitions are thus required to be at file scope.

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++ (as well as Charm, its C based version) 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 **??** 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 **charm++c** 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:

charm++c -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:

charm++c -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 List of Keywords in Charm++

The following is the complete list of keywords in CHARM++ :

1. chare
2. branched
3. message
4. entry
5. handle
6. thishandle
7. mainhandle
8. ALL
9. LOCAL
10. VARSIZE
11. accumulator
12. monotonic
13. readonly
14. writeonce
15. `=>`

The following is the complete list of runtime system calls in CHARM++. Currently no user defined functions may have one of these names.

1. new_chare
2. new_branched_chare
3. new_message
4. new_accumulator
5. new_monotonic
6. new_writeonce
7. new_prio_message
8. new_packbuffer
9. delete_message

10. CFunctionNameToRef
11. CFunctionRefToName
12. CHTimer
13. CMaxPeNum
14. CMyPeNum
15. CNumSpanTreeChildren
16. CPrintf
17. CPriorityPtr
18. CPrioritySize
19. CScanf
20. CSpanTreeChild
21. CSpanTreeParent
22. CSpanTreeRoot
23. CStartQuiescence
24. CTimer
25. CUTimer
26. ChareExit
27. CharmExit
28. DerefWriteOnce

C Further Information

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

C.1 Related Publications

1. For starters, *Reports and Publications: Parallel Programming Laboratory* catalogs all of the major papers written about Charm/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 can be used to express dependences among messages and computations.

C.2 Associated Tools

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

Dagger is a coordination language which helps to express dependences among messages and computations.

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.

C.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/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/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