CHARM++ : A Portable Concurrent Object Oriented System Based On C++*

Laxmikant V. Kale
Department of Computer Science
University of Illinois, Urbana-Champaign
email : kale@cs.uiuc.edu

Sanjeev Krishnan
Department of Computer Science
University of Illinois, Urbana-Champaign
email : sanjeev@cs.uiuc.edu

Abstract

We describe Charm++, an object oriented portable parallel programming language based on C++. Its design philosophy, implementation, sample applications and their performance on various parallel machines are described. 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. The language supports multiple inheritance, dynamic binding, overloading, strong typing, and reuse for parallel objects. Charm++ provides specific modes for sharing information between parallel objects. Extensive dynamic load balancing strategies are provided. It is based on the Charm parallel programming system, and its runtime system implementation reuses most of the runtime system for Charm.

1 Introduction

In the last decade, parallel processing has emerged as a powerful new technology. Many large scale commercial parallel machines are available today, such as Intel's iPSC/860 and the Paragon, NCUBE's 1024 processor machines, the CM-5, KSR, etc. A large class of applications, in science, engineering, operations research, and even artificial intelligence, can potentially benefit from parallel processing, and enable solution of important problems.

However, programming these machine remains a substantial hurdle. This is partly due to the new issues and difficulties that must be faced in writing parallel programs, such as scheduling, load balancing, synchronization and race conditions, the necessity of having to deal with communication latencies, and so on. In addition, parallel programming is made difficult by the fact the software developed for one parallel computer cannot be reused easily on another. These software hurdles must be overcome before the technology can be used effectively.

In the same decade, the object oriented methodology has emerged as a promising way of developing and organizing software, and permitting its reuse. This methodology has been brought into common practice to a large extent by pragmatic languages such as C++. Can object orientation help solve the parallel programming problem? It appears to be a natural choice for many reasons. The notion of state and persistence, which is one of the central features of object oriented methodology, naturally leads to the notion of a process, which is a central notion in parallel processing. Processes encapsulate local data and interact with other processes only via defined interfaces, such as messages. Other notions such as inheritance and polymorphism can

*This research was supported in part by the National Science Foundation grants CCR-91-06608 and CCR-90-07195.

be seen to acquire a new importance in the parallel world, where library modules must deal with data defined by other modules which may be distributed differently in different applications.

Even independent of the question of whether object orientation can be a help in solving problems of parallel programming, a separate argument can be made for developing a parallel object oriented system. Parallel processing and object oriented methodology have emerged, independently, as promising and popular new technologies. It can be predicted that both of these will be pursued and used extensively in years to come. It therefore makes sense to provide a bridge between the two powerful technologies, to combine their benefits.

These considerations have led to substantial research on concurrent object oriented systems, some of which are summarized in Section 2. The work presented in this paper is based on the parallel programming concepts developed in the Charm parallel programming system. The essential philosophy of Charm is discussed in Section 3, along with the guiding principles that we used for synthesizing these parallel programming constructs with object oriented constructs. We chose to use C++[27] as the underlying language, although the basic parallel constructs are equally applicable in other object oriented languages as well. Section 4 describes the resultant language, including the structure of the program, concurrent classes, abstractions for specifically shared objects, and features that support modularity. It also describes a simple example which demonstrates some concepts. Charm++ has been implemented to run on many parallel machines, including shared memory machines (e.g. Sequent Symmetry), non-shared memory machines (e.g. the nCUBE/2), uniprocessors, and networks of workstations. The translator used in this implementation, as well as relevant aspects of the runtime support system, are briefly discussed in Section 5. Section 6 describes a small set of applications implemented using Charm++, along with their performance results on parallel machines. Finally, Section 7 presents some conclusions and a discussion of future work. An example which uses dynamically created concurrent objects and shared objects is illustrated in Appendix A.

2 Previous Work

In this section we briefly discuss work done previously by other researchers in object oriented concurrent programming.

The notion of "actors" was described by Hewitt [17] and further developed by Agha [1]. One of the first implementations of Actors on commercial parallel machines was carried out recently using Charm [18]. Actors, which are concurrent objects, communicate with each other solely via messages, and allow concurrency even within a single actor. The execution model for actors is message driven, which is helpful for latency tolerance (see Section 3). However, we believe that concurrency within an object is difficult to manage, for the programmer as well as for the system. Moreover, using messages as the sole mechanism for information exchange diminishes the expressiveness of the language as well as its efficiency. Charm++ supports specific kinds of "shared objects" in addition to messages for this reason (Section 4.3). Other Actor like languages include Cantor [3].

ABCL/1[29] also has concurrent, message driven objects. An object processes one message at a time, and can selectively receive messages. Communication between objects can be blocking, non-blocking or future based. An express mode of messaging allows high priority messages to be processed quickly. ABCL/1 also supports delegation. A low-latency implementation of ABCL was recently done[28] on the Fujitsu AP1000.

Concurrent Smalltalk (CST) [11] is an experimental language designed to run on (fine grained) message-driven processors. The CA (Concurrent Aggregates) language [8, 9] supports a fine-grained model of parallelism, originally intended for the J-machine. An aggregate is a collection of objects that has a single name - a call to an aggregate may be sent to any one of its members. This idea has some similarities with the idea of "branch offices" (see Section 4.1) of chares, implemented in Charm around the same time that CA was defined [21]. CA offers concurrency within objects, which can be distributed across more than one processor, allowing hierarchies of parallel abstractions. CA also supports delegation, first class messages and continuations.

pC++ [14, 5] is a parallel C++ language largely oriented toward data-parallel and SPMD programs, and is based on CA and High Performance Fortran (HPF) ideas. It has a single thread of control, which can be forked into parallel threads (one on each processor) via a concurrent call. The concurrent objects share arrays which are distributed using HPF-like constructs such as alignments and distribution-templates. Originally designed for shared memory machines, pC++ has recently been implemented on the CM-5. C** [24] is another data-parallel C++ language.

Mentat is a noteworthy portable C++ based language [15]. Concurrent objects are specified separately from sequential objects, which is a desirable feature, because it gives the programmer control over parallelism. Mentat, however, overloads constructs for method invocation and message sending, which makes the cost of a call unclear to the programmer. Mentat supports futures, and manages object location, and synchronization and forwarding of communication automatically based on compiler analysis of dependences. Mentat does not appear to provide any special constructs for programming regular, data-parallel computations.

pSather [12] is based on the Eiffel object oriented language. It supports a shared-memory model, based on clusters of shared-memory processors. Objects are localized to a cluster, and can have multiple concurrent threads, with synchronization provided by a monitor construct. pSather has data-parallel constructs for distribution and alignment.

ESP-C++ [25] has concurrent objects, transparent remote method invocation, and blocking as well as non-blocking, future based messaging. Amber [7], and its predecessor Presto [4], are designed for a network of shared-memory multiprocessors. They provide a uniform network-wide object space. Amber objects are passive entities, and thread objects invoke operations on them. The programmer can control object location by migration primitives. Both ESP and Amber do not seem to have been ported to commercial massively parallel computers.

There are several other efforts in this area. CC++ [6] provides various parallel constructs while using C++ for the sequential portions of the codes. POOL-T [2] has parallel objects that can be dynamically created, with a blocking model of communication between objects. Parallel-C++ [19] provides a restricted "co-begin, co-end" model of process creation. Only sequential objects are allowed, which can be "migrated" to other processors. OOMDC/C [10] has a message driven computing model built on C, and does not appear to fully support object oriented features like inheritance and dynamic binding.

Charm++ differs from most, if not all, of the above in the following respects : Charm++ supports abstractions for specific modes of information sharing, in addition to supporting communication via messages. It incorporates a message-driven scheduling strategy, which is essential for latency tolerance. It provides extensive support for dynamic load balancing, and prioritization of messages. It supports a novel replicated type of object called a "branched chare" which has a sequential as well as parallel interface, and which can be used for efficiently programming data parallel applications. Charm++ does not depend on an operating system provided threads package, hence avoids the corresponding overhead and non-portability. It is also one of the few systems that has been implemented on many commercial shared as well as large distributed memory machines.

3 Design Philosophy

We will first describe the parallel programming concepts developed in the Charm parallel programming system[21, 13, 20], and then discuss issues involved in synthesizing them with object oriented notions.

3.1 The Charm Parallel Programming Philosophy

The rationale used in the design of Charm can be summarized as follows.

Portability is an essential catalyst for large scale development of parallel software. Charm programs run without change on all MIMD machines.

The system must allow dynamic creation of parallel work, and support it with *dynamic load balancing* strategies. Such a facility is necessary to handle many irregular parallel computations (such as AI search computations, discrete optimization, irregular finite-element computations, adaptive grid refinements, etc.).

Simple Charm programs, therefore, specify parallel processes called *chares*[1]. Chares can create new chares, and send messages to each other. (It is possible to have a large number of chares per processor.) However, messages are only a restricted mode in which chares share information with each other. Parallel computations in general require processes to share data in a few specific but distinct modes. Recognizing this, Charm provides additional *data abstractions for information sharing*. These include constructs such as read-only data, accumulators, monotonic variables (used in branch-and-bound computations), and distributed tables.

Charm also supports a special form of chare called *branch-office chare* (BOC). A BOC instance has one branch on every processor. Each branch can send and receive asynchronous messages as a chare, but may also provide sequential public functions. A BOC instance has a single global name. So, a dynamically load balanced chare can call a BOC function without knowing which processor it is on; the call is always handled by the local branch of the BOC instance[2]. BOCs support regular, data parallel computations easily and efficiently.

Latency Tolerance: Remote data always takes more time to access than local data, on any scalable parallel machine. Moreover, the arrival of remote data can be further delayed due to runtime conditions and computations on remote processors. The parallel programming system should make it possible to tolerate this latency of communication. Message driven execution — instead of the traditional blocking-receive-based communication — is employed in Charm to attain this goal. In message driven execution, all computations are initiated in response to messages being received.

All system calls in Charm are non-blocking. So access to remote data is always done in a "split-phase" fashion. Along with message driven execution, this induces better latency tolerance : while one process is waiting for remote data, another process, which has a message directed to it, may be scheduled on the same processor. What is more, a single process may wait for multiple data items simultaneously. Hence split-phase remote access allows overlap of communication and computation.

Futures are sometimes used for latency tolerance in specific contexts. A process spawns another process to compute a value to be stored in a future. The calling process (or others) may do other computations and then when they need the value of the future, make a call to access it. If the value in the future has been computed, the calling process continues with this value; otherwise it blocks until this value is available. Compared to a (synchronous) remote procedure call (RPC), futures are better at overlapping communication and computation. However, futures do not allow a single process to wait for multiple messages simultaneously.

[1] chare is Old English for chore.

[2] In contrast, a call to a concurrent aggregate in CA goes to a system chosen representative on some processor via an RPC-like mechanism.

Also, futures do not adapt to the runtime variation of remote response time: a programmer must decide how much computation to do before blocking on a future. Finally, transparent implementation of futures [15] hides the true cost of remote invocation from the programmer, because all invocations appear to have the same blocking syntax and semantics.

3.2 Synthesizing Parallelism with Object Orientation

For completeness, we briefly restate some of the essential properties of object oriented programs. An object oriented program (e.g. a C++ program) consists of class definitions, global variable declarations and function definitions. Each class encapsulates some data, and defines public and private functions to access and modify this data in specific ways. In addition to this data abstraction, an object oriented language supports *inheritance*, and *dynamic binding*. A class may be defined as a subclass of another, thus inheriting all the functions defined for the superclass. A class may also redefine some of the functions defined by the superclass. Dynamic binding refers to the ability to determine the function to be called at run time. The programmer calls a function of an object belonging to a base class, which at runtime may have been specialized to an instance of a derived class.

Charm++ disallows unrestricted global variables and static variables in classes. Arbitrary operations (mutators) on such variables cannot be implemented with satisfactory (e.g. serializable) semantics on parallel machines. Instead, we permit only global variables that are instances of predefined information sharing classes (discussed in Section 4.3). Processor specific global variables are supported as public variables of branch-office chares. Other than this, we decided to retain all the features of C++. This allows one to include and interface with sequential C++ code.

Charm++ consists of the following categories of objects:

1. *Sequential objects*

2. *Concurrent objects* naturally arise from the processes (chares) of Charm. These are localized units of work. Each chare object has its own local data and the code for handling messages are methods of these objects.

3. *Replicated objects* are the branch-office chares of Charm, and consist of a branch on every processor.

4. Next, the specific information sharing modes in Charm each become a template for a *shared object*. These objects may encapsulate data of any type, but have specific operations that can be performed on them. Shared objects are not localized to any particular processor, and may constitute a distributed data structure.

5. *Communication objects* represent entities that can be sent as messages between concurrent objects.

We feel that it is important to clearly distinguish between sequential and concurrent objects. First, on most parallel machines, the cost of sending a message to a remote object is significant - several tens to hundreds of microseconds. Hence it is important for programmers to have a clear cost model for the actions in the program. In particular, they should understand which parts of their code involve expensive remote actions, and which ones are simply local function calls. Second, as stated above, Charm philosophy necessitates split-phase transactions, or asynchronous invocation of methods. Thus, when one places a “call” to a concurrent object, it is simply deposited into the runtime system, and the caller continues executing subsequent code. Eventually, the system schedules the target object to process the given message. That object may then initiate some other concurrent computations, collect their results, and send them in a message to the caller object. Clearly, providing the same interface as that for sequential objects will be misleading because the programmer might expect the call to immediately execute, and return the result. Third, a clean separation between sequential and parallel objects helps algorithm design : the "parallel paradigm" (which would be common for a class of algorithms) can be encoded in libraries using parallel objects to coordinate sequential processes. For each particular algorithm then, only the sequential part has to be specified, and this can be done using the best sequential algorithm.

While multiple inheritance, dynamic binding, and overloading are supported for sequential objects by C++, Charm++ extends these concepts for concurrent and replicated objects. Thus we allow inheritance hierarchies of parallel classes. Dynamic binding is supported by allowing run time determination of the remote parallel object type whose code is to be executed. Finally, overloading is supported by allowing message types to determine the code to be executed on remote objects.

4 The Charm++ Language

Charm++ is basically C++ without global variables, and with a few extensions to support parallel execution. Operations and manipulation of chares are restricted (as compared to sequential objects) to conform with parallel execution requirements.

4.1 Structure and execution model

A Charm++ program consists of modules. Each module is defined in a separate file. It can contain declarations and definitions of the following entities:

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 have two function members : *pack* and *unpack* (see Section 4.6), which have to be defined by the user[3].

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:
  EntryPointName(MessageType *MsgPointer)
    { C++ code block }
} ;
```

As chares can exist on remote processors, method invocation may involve remote access. However, the philosophy of non-blocking remote access dictates that all remote access be through messages. Hence chare definitions cannot have public members, unlike C++ class definitions. The entry point definition specifies code that is executed *without interruption* when a message is received and scheduled for processing. Only one message per chare is executed 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. Each chare instance is identified by a *handle* which is unique across all processors. The handle of a chare has type `ChareName handle`.

Branched chare classes : These are modeled after the "branch office chares" of Charm. They are special replicated chares that have a branch on all processors, 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 similar definition syntax as 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
} ;
```

[3] 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++.

Sequential objects, normal chares and other branch chares can access public members of the branch of a branched chare *on that processor* by
`LocalBranch(ChareHandle)->DataMember` and
`LocalBranch(ChareHandle)->FunctionMember()`.

Specifically shared objects : These are discussed in detail in Section 4.3.

Global functions : These are defined as in C++.

Sequential classes : Charm++ allows hierarchies of classes as in C++.

Every Charm++ program must have a chare type `main`. There can be only one chare instance of this chare type, which executes on a single system selected processor. This must have the entry point `main`. Execution of a Charm++ program begins at this entry point. Typically, this entry point is used to initialize specifically shared variables and create new chares and branched chares. Charm++ program execution can be terminated by the CharmExit() call. The `main` chare has an optional `Quiescence` entry point which is executed when the parallel 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).

4.2 Basic Charm++ calls

The Charm++ calls that provide support for parallel execution are described below.
Chares are created with the call
`new_chare(ChareName,EP,MsgPointer)`.
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 causes the chare to be initialized by executing its entry point `EP` with the message contained in `MsgPointer`. Using an optional argument the user can also specify a processor to create the chare on, thereby overriding the dynamic load balancing strategy. `new_chare()` does not return any value. The user may, however, obtain a *virtual handle* (virtual because the chare has not yet been created) to the chare by specifying another optional argument. This handle may be used for sending messages to the chare or may be passed to other objects (see also the `MyChareHandle()` call).

Branched chares are created with the call
`new_branched_chare(ChareName,EP,MsgPointer)` This creates a branch on every processor and initializes it by executing `EP`. Branched chares are usually created in the `main` entry point of the `main` chare, in which case this call returns the handle of the newly created branched chare. When branched chares are dynamically created, the user can specify an entry point and chare handle at which the handle of the newly created branched chare can be received.

Messages can be sent to chares by the call:
`ChareHandle=>EP(MsgPointer)`.
This sends the message pointed to by `MsgPointer` to the chare having handle `ChareHandle` at the entry point `EP`, which must be a valid entry point of that chare type. Note that this is different from sequential method invocation in that it is non-blocking[4].

Messages to a branch of a branched chare on a processor P can be sent by
`ChareHandle[P]=>EP(MsgPointer)`.

Messages to branched chares can be broadcast to all branches of that chare (on all processors) by
`ChareHandle[ALL]=>EP(MsgPointer)`
where `ALL` is a system defined reserved constant.

Auxiliary calls

The `MyChareHandle(ChareHandle)` call fills in the handle of the currently executing chare in `ChareHandle`. `MainChareHandle(ChareHandle)`, similarly, obtains the handle of the `main` chare. These

[4]The syntax of the Charm++ calls described above is intentionally different from the analogous operations on sequential objects as in C++. This is so that the programmer can clearly distinguish the semantics of parallel objects. "=>" instead of "->" for sending messages is used to emphasize the difference between method invocation and message sending : the latter is non-blocking. Similarly, the new_chare() call has an analogous function to the "new" operator in C++ ; however, it does not return any pointer to an object, and moreover, it is non-blocking.

handles should be used in preference to virtual handles because virtual handles involve an extra level of redirection of messages.

Since messages are handled differently by the system, they are allocated memory using a different call : void *new_message(MESSAGE_TYPE). The programmer relinquishes control of a message when it is sent. However, a message received at an entry point stays persistent : it may be reused or freed by the programmer.

The ChareExit() call tells the system to release memory of the currently executing chare *after the execution of the current entry point is completed.* Charm++ does not provide automatic garbage collection of chares. The CharmExit() call causes the system to terminate all processes (non-preemptively, as above).

Terminal input and output from any processor is accomplished by the CPrintf() and CScanf() calls, which are similar to their C counterparts. They guarantee atomicity, ensuring that outputs from two CPrintf() calls on different processors do not get mixed.

Since pointers are not valid across processors in general, function pointers are meaningless if passed in a message. Charm++ provides calls to convert function pointers to and from function reference indices, which can be passed in messages :

FnRefType FunctionNameToRef(fn-pointer) converts a function pointer to a reference index, and

FN_POINTER FunctionRefToName(fn-ref) does the reverse.

Charm++ provides a few other calls to query processor numbers and for timing computations in a portable manner.

4.3 Specifically shared objects

Since global variables cannot be provided in a parallel execution environment, Charm++ provides specific abstract object types for sharing data amongst chares, as in Charm. Each abstraction for information sharing may be thought of as a template, with predefined functions whose code is to be provided by the user. These objects are created and initialized in the main entry point of the main chare, after which they can be accessed only through their specific modes, on all processors. This section presents a brief description of these abstractions.

- **Read-Only objects** These objects hold information (e.g. problem parameters) that is obtained immediately after the program begins execution, but which does not subsequently change. Read-only objects can be accessed from any chare on any processor, without any need for "split-phase" access.

- **Write-Once objects** These are dynamic versions of read-only objects. They can be initialized once, from outside the main chare, using the results of a parallel computation.

- **Accumulator objects** An accumulator object has two operators on it : *add* which adds to the object in some user defined manner, and *combine*, which combines two objects. These operators must be commutative-associative. The accumulator template is predefined by the system ; however, the code for the add and combine operations for the particular type of accumulator has to be provided by the user. Accumulators are initialized from the main entry point of the main chare. Any chare can "add" to an accumulator variable. The final value can be accessed once for each accumulator variable. Accumulators are optimized for fast update, because they can be read only once.

- **Monotonic objects** A monotonic object is updated by an operation that is idempotent as well as commutative - associative. Moreover, the object takes on monotonically increasing "values". Like accumulators, the monotonic template is predefined, and the user has to provide code for the update operation. Monotonic variables are initialized in the main entry point of the main chare. Any chare can update the variable and also access its value more than once. This value may not be the current globally best value, but the system propagates the best value as soon as possible.

- **Distributed Tables** A distributed table is a set of entries, where each entry is a record with a key and data field. A distributed table type can be defined for any particular data type. Access to entries is through calls which *Insert, Delete* and *Find* entries in a particular table. All these calls are non-blocking, and their behavior depends on the status of the table and options specified in the calls.

4.4 Software reuse through modules

In sequential programs, modularity is enforced through encapsulation and clearly defined interfaces. The modularity features of Charm++, like those in Charm, are motivated by the need for reusing independently developed, previously compiled modules that can control the visibility of the entities they contain. This necessitates constructs in addition to those provided by C++ or C.

A Charm++ program is written as a set of modules, which can be separately compiled. A complete chare definition cannot be split across modules. A module is similar to an ADA package; it encapsulates a set of related functions and classes and can control what entities it exports and imports using *interface statements*. Interface statements for a module include prototypes of all chares, shared objects, and functions that are to be exported by the module. A module imports other modules by including their interface statements. Name resolution is done by the scope resolution operator "::". Thus entry point EP in chare type ChareName in module Module1 is referred to as `Module1::ChareName::EP`. Remote creation and message passing require chare, entry point and message names (or *ids*) to be sent across processors. Modules help to ensure consistency of ids across separately compiled program units, especially in case of dynamic binding and inheritance (see Section 5).

Supporting modularity and reuse is more difficult in a parallel context than in a sequential context. The issues here are :

1. For scalability, modules must be able to exchange data in a fully distributed fashion, without the need for centralized transfer. This is achieved in Charm++ with the branched chare abstraction. Since a branched chare has a branch on each processor, two branched chares in separate modules can exchange data scalably. Also, entities in one module may deposit data in distributed tables which can be accessed by other modules.
2. A reusable module cannot manipulate function pointers because it may need to pass the function in a message, to be invoked on a different processor. (E.g. comparison function pointers may not be enough for a parallel sorting module). Charm++ provides function reference indices (Section 4.2 above) which can be passed across processors in messages.
3. A reusable module must not assume an input data distribution. (E.g. a matrix multiplication module must not assume a particular distribution of the matrices across processors). This is supported by branched chares which can encapsulate a data distribution, and have public functions.

4.5 Load balancing and memory management strategies

Charm++ allows the user to select from a number of dynamic load balancing strategies depending on the requirements of the application. The strategies are implemented as modules on top of the basic runtime system. Some of the load balancing strategies supported include[26]

1. Random : new chares are sent to random processors
2. Adaptive Contracting Within Neighborhood : new chares are balanced within a neighborhood, leading to a global balance.

3. Central manager : All new chares are sent to a central manager which redistributes them among processors. This strategy is meant for use with prioritized task creation.

4. Token : This is a more sophisticated, scalable strategy for use with prioritized task creation.

4.6 Other Charm++ features

Prioritized Execution : Charm++ provides many strategies for managing queues of messages waiting to be processed. Some of them (FIFO, LIFO, etc) are based solely on the temporal order of arrival of messages. The user can also assign priorities to messages depending on the application requirements. Charm++ supports integer priorities as well as bit-vector priorities (with lexicographical comparison of bit-vectors determining order of execution).

Conditional Message Packing : Charm++ allows arbitrarily complex data structures in messages. On non shared memory systems, pointers are not valid across processors, hence it is necessary to copy (*pack*) the structure into a contiguous block without pointers before sending the message. However, packing is wasteful if the message is not sent to another processor, or on shared memory systems. Hence for messages involving pointers the user is required to specify two functions for packing and unpacking messages that are called *by the system* before sending and after receiving a message, respectively. Thus only messages that are actually sent to other processors are packed.

4.7 An example

The following example illustrates inheritance and dynamic binding in a parallel context. The program is a regular, mostly data parallel algorithm for iterative solution of partial differential equations using the Jacobi method. A blocked data partitioning is used to distribute the 2-dimensional domain among processors.

Consider a server chare of type `ReductionAnd` [5]

```
module interface Reduction {
  enum Boolean { FALSE, TRUE };
  message BooleanObject {
    Boolean convergeValue;
  };

  // message used for initializing the
  // ReductionAnd branched chare
  message ReductionInit { /* parameters */ };

  branched chare class Receiver {
  entry :
    // This is called in a broadcast when the
    // reduction is completed.
    virtual ReceiveResult(BooleanObject *result);
  };

  branched chare class ReductionAnd {
  entry:
    Initialize(ReductionInit *parameters);
    // Called when this chare is created by a
    // new_branched_chare call
  public:
    void StartRedn(BooleanObject *local,
                   Receiver handle h);
    // Called by clients to start a global AND
  };
};
```

Figure 1: **Interface for the Reduction library module, which is imported in the Jacobi program**

(Figure 1) which provides a global AND reduction operation over all processors. This service is needed by many other types of client chares. The library module `Reduction` that defines `ReductionAnd` also defines and exports the message type `BooleanObject` and a class `Receiver` containing a virtual entry point `ReceiveResult` that receives results from `ReductionAnd`.

Client chare types may be defined in separate modules. Chare types wanting the `ReductionAnd` service import `Receiver` in the module they are defined in and list `Receiver` as one of their base classes. The clients may choose to inherit the virtual entrypoint `ReceiveResult`, or redefine it for

[5] By having general message objects that can define their own combining functions, we can have a general reduction module that supports arbitrary combining operations and data types.

their specific purpose. Dynamic binding ensures that the proper ReceiveResult function will be called. Note that the clients may be of any type, as many different clients may require the same service.

Figure 2 shows part of the message and sequential class definitions for the Jacobi program. In Figure 3, Jacobi is a branched chare which is a subclass of the Receiver branched chare, allowing it to use the ReductionAnd class to do global convergence testing. Figure 4 has the main chare for the Jacobi program.

The Jacobi branched chare is created from the main entry point of the main chare. It is initialized by sending a message to the BranchInit entry point (which is executed by each branch on each processor). BranchInit starts the computation by sending the boundaries of its local subdomain to neighboring processors. All processors receive boundaries from their neighbours at the recvBoundary entry point. Note that messages from neighbours can be received in any order, hence the maximum possible overlap of computation (copyBoundary()) with communication is possible. After receiving messages from all neighbours (as measured by the variable count), the iterate private function does the local computation and checks for local convergence. The local convergence value is then passed to the Reduction module by the StartRedn call. Note that after the local StartRedn call is made, the local processor is free to do other work, which would not be possible in a system supporting only blocking calls. After global reduction is complete, the result is returned to the ReceiveResult entry point. This checks if the computation has globally converged, and initiates the next iteration otherwise, by sending its new boundaries to the neighbors. At this point the local branch might already have received the boundaries from all its neighbors (e.g. because they received their reduction results earlier). If so, it calls iterate immediately. Otherwise iterate will be called from the recvBoundary entry point when messages from all neighbors have been received.

Although the above Jacobi code is somewhat less concise than code written in a blocking style, it has significant performance advantages, and moreover, would be only a small part of the actual application code which includes the sequential functions not shown in the figures.

The ReductionAnd chare could be used similarly by, say, a 3-dimensional grid module. Also, the ReductionAnd class may implement global AND in different ways, even possibly supporting multiple (pipelined) concurrent operations.

```
module JacobiModule {

// nxn is the size of each processor's sub-domain
const int n = 32;

enum Direction { NORTH, SOUTH, EAST, WEST };

// Message holding boundary values that have
// to go across processors
message BOUNDARY {
  Direction from;
  float boundary[n+2];
};

// Message to initialize Jacobi branched chare
message JacobiInit { ReductionAnd handle redn; };

class TwoDGrid { // sequential class
  // The array storing values of points on this
  // processor's sub-domain.  There are two
  // extra rows and columns for boundaries.
  float A[n+2][n+2];

public:
  // Constructors and other public functions for
  // operations on 2-D grids such as update and
  // local convergence.

};
```

Figure 2: **Message and sequential class definitions for the Jacobi program**

5 Implementation

Charm++ has been implemented as a translator and a runtime system. The translator converts Charm++ constructs into C++ constructs and

```
branched chare class Jacobi :  public Receiver {
  TwoDGrid P, Q;
  int count, numOfNeighbours, K, mypenum;
  Boolean reductionDone;
  ReductionAnd handle rednHandle;
  Jacobi handle myHandle;

public:
  Jacobi() {}

entry:
  BranchInit(JacobiInit *msg)
  { rednHandle = msg->redn;
    delete_message(msg);
    myHandle = MyChareHandle();
    initialize();
    sendBoundaries();

    count = numOfNeighbours;
    reductionDone = TRUE;
  }
  recvBoundary (BOUNDARY *msg)
  { copyBoundary(msg);
    if (--count == 0 && reductionDone)
      iterate();
  }
private:
  void iterate()
  { P.update(&Q);

    BooleanObject *local = (BooleanObject *)
                       new_message(BooleanObject);
    local->convergeValue = P.LocalConverge(&Q);
    reductionDone = FALSE;
    count = numOfNeighbours;
    LocalBranch(rednHandle)->StartRedn(local,
                                     myHandle);
  }
entry:
  ReceiveResult(BooleanObject *global)
  { if ( global->convergeValue )
      P.PrintResult(mypenum,K);
    else { // start next iteration
      reductionDone = TRUE;
      P.copy(&Q);
      sendBoundaries();
      if (count == 0) iterate();
    }
  }
  void initialize();
  void sendBoundaries();
  void copyBoundary(BOUNDARY *msg);
};
```

Figure 3: **JacobiModule continued : the Jacobi branched chare**

```
chare class main {

entry:
  main()
  { // Execution of the program starts here
    ReductionInit *m = (ReductionInit *)
                         new_message(ReductionInit);
    JacobiInit *msg = (JacobiInit *)
                            new_message(JacobiInit);

    // Create the branched chare for doing
    // global AND reduction
    msg->redn = new_branched_chare(ReductionAnd,
                                        Initialize,m);

    // Start the Jacobi iterations by creating
    // the Jacobi branched chare
    Jacobi handle jacobi_boc;
    jacobi_boc = new_branched_chare(Jacobi,
                                      BranchInit,msg);

  }
};

}; // End of JacobiModule
```

Figure 4: **JacobiModule : the main chare**

calls to the runtime system. The runtime system of Charm is reused with modifications to support C++ interfaces. Currently, a prototype translator with complete functionality but restricted error checking has been implemented. Several test and application programs have been successfully executed on various parallel machines.

The Charm runtime system [13] is written in C. It's lowest layer consists of a machine dependent set of routines which use the calls provided by the particular machine. On top of this is a machine independent set of routines that implement the various functions such as chare creation, message processing, performance measurements, quiescence detection, etc. The different strategies for queue and memory management and dynamic load balancing are written as modules that can be linked in at link time as specified by the user.

One important function of the Charm++ translator is to map parallel class and function names into consistent ids which can be passed to other processors. This is important when considering dynamic binding : when a sender sends a message to a chare C at an entry point E defined in C's base class, C must call its own definintion of E if it has been redefined, otherwise it must call its base class' definition of E. Also, multiple inheritance requires that ids of entry points inherited from different base classes do not clash. Again, separate compilation of modules means that ids cannot be assigned at compile time. On the other hand, all ids must be compile time constants if they are to be used in a switch statement (as opposed to an inefficient if-then-else structure) to execute the proper code at the destination of the message.

To meet these conflicting requirements the translator generates functions which assign globally unique indices to chare and entry point names during initialization at run time. The translator also generates a public *entry point selector function* for each chare class which correctly maps global ids to the appropriate chare-local constants and then uses a switch statement to call the proper local function. Thus unique global ids can be passed in messages across modules. Similarly, for creating new chare objects, the translator generates a *chare selector function* for every module.

One of the main units in the runtime system is the "pick and process" loop. This picks up incoming messages from the system message buffer, orders them according to a queueing strategy, and then processes them according to the type of message. The message for creating new chares includes a chare name index, which is used by the translator-generated chare selector function to create the appropriate chare object. The message for execution of an entry point of an existing chare includes an entry point index along with a message pointer and the chare object pointer. The index is used in the translator-generated entry point selector function to call the proper entry point in the proper chare.

6 Performance results

Charm++ can be used for regular as well as irregular parallel computations. Tables 1 and 2 present performance numbers in terms of speedups for three small applications on the nCUBE/2 (distributed memory multicomputers), and Sequent Symmetry (shared memory multiprocessor). Preliminary results for the CM-5 are presented in table 3.

- Jacobi : This is a regular, mostly data parallel algorithm for iterative solution of partial differential equations. The Jacobi method is used for 200 iterations of smoothening of a five-point stencil. The implementation mainly uses branched chares, with blocked partitioning of the 2-dimensional array. The speedups given are scaled speedups, obtained with a problem size of 32x32 grid points per processor. The timings for 1 processor are much lower than the rest because there is no communication at the boundaries of the grid.
- TSP : This is an irregular application consisting of a parallel branch and bound implementation of the asymmetric Traveling Salesperson Problem. The implementation uses integer message priorities (equal to the lower bound on cost of a branch and bound node) to guide search through the branch and bound tree. The cost of the best solution is maintained using a monotonic variable. The token based load balancing strategy [26] was used to distribute chares. Speedup levels off after 64 processors on the nCUBE/2 because of a large increase in the number of wasteful branch and bound nodes.
- Primes : This is a program to compute the number of primes between 2 and a large number N (of the order of a billion). The parallel algorithm is based on the Eratosthenes' sieve method. The implementation uses normal chares to distribute work among processors. The speedups are with reference to a serial algorithm which does not create any

chares, hence does not include any overhead of parallelism.

PEs	Jacobi (32x32)	TSP (40 cities)	Primes (10^9)
1	0.88 (1.0)	242.7 (1.0)	4717.7 (1.0)
16	1.5 (9.1)	20.5 (11.9)	579.0 (8.2)
64	1.6 (35.4)	11.2 (21.7)	150.8 (31.3)
256	1.7 (130.2)	11.2 (21.8)	32.3 (145.9)

Table 1: **Times (in seconds) and Speedups for 3 applications on the nCUBE/2.**

PEs	Jacobi (32x32)	TSP (40 cities)	Primes (10^8)
1	10.5 (1.0)	889.6 (1.0)	734.8 (1.0)
4	11.4 (3.7)	224.0 (4.0)	184.2 (4.0)
9	12.5 (7.5)	101.8 (8.7)	82.4 (8.9)
16	13.9 (12.1)	58.8 (15.1)	46.6 (15.8)

Table 2: **Times (in seconds) and Speedups for 3 applications on the Sequent Symmetry.**

PEs	Jacobi (64x64)	TSP (40 cities)	Primes (10^8)
64	18.7	19.6	25.3

Table 3: **Preliminary Times (in seconds) for 3 applications on the CM-5.**

7 Conclusions and Future work

We have presented a portable object oriented parallel programming system. We have discussed the design issues for Charm++, described its important constructs, and presented preliminary performance results for both shared as well as non-shared memory machines.

Charm++ provides a rich set of features that make it suitable for a broad range of applications. Some of the unique features in Charm++ are: its comprehensive support for *both* regular as well as irregular computations; its message driven execution model which leads to better efficiency; its support for specific, widely useful information sharing abstractions; and its user-selectable strategies for managing parallelism. Charm++ makes the programmer and the runtime system each do what they do best. The programmer has to specify parallel computations, which leads to better parallel algorithm design. The runtime system decides when and where to execute work by scheduling and dynamic load balancing. We feel that for efficient utilization of computing power (sustained speedups as opposed to theoretical peak speedups) good parallel algorithm design is essential. This is only possible if the programmer is given flexibility in making design decisions along with abstractions that hide low level details of *how* things are done. Charm++ makes a significant step in this direction.

Charm++ is the latest component of the broader family of Charm parallel programming tools. Since Charm++ shares the runtime system with Charm, it can be used with Projections[22] and future performance feedback tools developed for Charm. We also expect Charm++ modules to co-exist with Charm as well as DP[23] (an HPF based data-parallel language being developed on top of Charm) modules, in a single application. Dagger[16] is a notation (and a visual editor) for expressing synchronization constraints (dependences between messages and computations) within a chare. It will be extended to provide the same facilities in Charm++.

Our future agenda for Charm++ consists of work in implementation, language design, and applications. We will fine tune Charm++ for better performance and enhance the front-end for better error checking and recovery. The syntax of Charm++ constructs may be refined depending on initial experiences and capabilities of the translator. Many of the features in Charm++ can be generalized. We may consider allowing multiple value parameters as entry point arguments, with marshaling at the sender's end. Alternatively, message objects can be generalized to arbitrary objects, including parallel objects. Finally, we will continue development of applications in Charm++ that benefit from object oriented parallelism.

Acknowledgement

This work would not have been possible without the research on the Charm runtime system over the past several years by the current and past members of the Parallel Programming Laboratory, including Wennie Shu, Kevin Nomura, Wayne Fenton, Balkrishna Ramkumar, Vikram Saletore, Amitabh Sinha and Attila Gursoy.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] P. America. Issues in the design of a parallel object oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[3] W. Athas and N. Boden. Cantor : An actor programming system for scientific computing. In *Proceedings of the ACM SIGPLAN Workshop on Object Based Concurrent Programming, ACM SIGPLAN Notices*, pages 66–68, April 1989.

[4] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object oriented parallel programming. *Software: Practice and Experience*, 18(8), August 1988.

[5] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language, 1992.

[6] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, Department of Computer Science, California Institute of Technology, 1992.

[7] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system : Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles, in ACM SIGOPS Operating Systems Review*, December 1989.

[8] A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[9] A. Chien and W. J. Dally. Concurrent aggregates. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–196, March 1990.

[10] T. W. Christopher. Early experience with object-oriented message driven computing. In *Proceedings of the 3rd Symposium on Frontiers of Massively Parallel Computing*, October 1990.

[11] W. Dally and A. Chien. Object oriented concurrent programming in CST. In *Proceedings of the Third Conference on Hypercube Computers*, pages 434–439. SIAM, 1988.

[12] J. Feldman, C-C. Lim, and T. Rauber. The shared-memory language pSather on a distributed-memory multiprocessor. In *Proceedings of the Second Workshop on Languages, Compilers and Runtime Environments for Distributed Memory Multiprocessors*, October 1992.

[13] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, August 1991.

[14] D. Gannon and J. K. Lee. Object oriented parallelism: pC++ ideas and experiments. In *Proceedings of 1991 Japan Society for Parallel Processing*, pages 13–23, 1993.

[15] A. S. Grimshaw. Easy-to-use object oriented parallel programming with Mentat. Techni-

cal Report CS-92-32, Department of Computer Science, University of Virginia, Charlottesville, 1992.

[16] A. Gursoy and L. V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. Technical Report 93-3, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, March 1993.

[17] C. Hewitt, P. Bishop, and R. Steiger. A universal ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 235–245. SIAM, 1973.

[18] C. Houck and G. Agha. Hal: A high level actor language and its distributed implementation. In *Proceedings of the International Conference on Parallel Processing*, August 1992.

[19] C-H. Jo, K. M. George, and K. A. Teague. Parallelizing translator for an object-oriented parallel programming language. In *Proceedings of Tenth Annual Phoenix Conference on Computers and Communications*. IEEE Computer Society Press, March 1991.

[20] L. V. Kale. A tutorial introduction to CHARM, December 1992.

[21] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

[22] L.V. Kale and A. B. Sinha. Projections: A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Symposium*, April 1993.

[23] E. Kornkven and L. V. Kale. Dynamic adaptive scheduling in an implementation of a data parallel language. Technical Report 92-10, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, October 1992.

[24] J. Larus, B. Richards, and G. Viswanathan. C** : A large-grain, object-oriented, data-parallel programming language. Technical Report 1126, Computer Sciences Department, University of Wisconsin-Madison, 1992.

[25] W. Lau and V. Singh. An object-oriented class library for scalable parallel heuristic search. In *Proceedings of the European Conference on Object Oriented Programming*, July 1992.

[26] A. B. Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *Proceedings of the International Parallel Processing Symposium*, April 1993.

[27] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[28] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, June 1993.

[29] A. Yonezawa. *ABCL: An Object Oriented Concurrent System*. MIT Press, 1990.

Appendix A

The Charm++ program shown in figures 5, 6 and 7 computes the number of primes between 1 and a large number (of the order of a billion). The algorithm first divides the range of numbers among dynamically created chares using a divide-and-conquer tree. Each chare at the leaf of the tree calls a sequential function which computes the number of primes in its range. The count of primes computed by each chare is summed using an accumulator object.

Figure 5 presents the message and accumulator declarations for the Primes program. `seqPrimes` is a function defined in a separate file, and so is accessed via an extern declaration as usual. The `RangeMsg` is used to spawn the divide-and-conquer tree. The type of the data held by the accumulator class `AccCount` is "pointer to `MsgAccCount`". The only data any accumulator object is allowed to hold is a pointer to a communication object (i.e. a message), because the system may implement accumulators by having multiple copies on different processors, thus requiring the accumulator data to be sent across processors. The `AccInitMsg` is used to send initial data to the accumulator. Although it appears redundant in this example, general usage of accumulators requires such a message. For example, if one were to define an accumulator to hold a histogram, the initialization message may

```
module Primes {
// This is the sequential function which
// returns the number of primes in a range.
extern int seqPrimes(int low, int high);

// LENGTH is the size of the range which is
// processed sequentially by seqPrimes()
const int LENGTH = 10000;

message MsgAccCount { int data; };

message AccInitMsg { int data; };

// This message tells a chare what range of
// numbers it should process
message RangeMsg {
  int Low, High;
};

// This is the accumulator type used for
// summing primes found by each chare.
class AccCount :  public Accumulator {
  MsgAccCount *msg;

public:
  AccCount(AccInitMsg *initmsg)
  {
    // "Constructor" used for initializing
    msg = (MsgAccCount *)new_message(MsgAccCount);
    msg->data = initmsg->data;
  }

  void Accumulate (int x)
  { // This accumulates a count
    msg->data += x;
  }

  void Combine (MsgAccCount *y)
  { // Called only by the system,
    // to combine counts from two processors
    msg->data += y->data;
  }
};

// This is the accumulator shared object,
// which can be accessed by all processors
AccCount *total;
```

Figure 5. Primes module : declarations

```
chare class main {

entry:
  main()
  {
    int Limit;

    CPrintf("Enter upper limit of range :  ");
    CScanf("%d", &Limit);

    // Create and initialize the accumulator
    AccInitMsg *acc_msg = (AccInitMsg *)
                          new_message(AccInitMsg);
    acc_msg->data = 0;
    total = new AccCount(acc_msg);

    // Create the first chare at the root
    // of the divide-and-conquer tree
    RangeMsg *msg =
             (RangeMsg *) new_message(RangeMsg);
    msg->Low = 1;
    msg->High = Limit;
    new_chare(PrimesChare, Goal, msg);
  }

  Quiescence() {
    // This is executed when all chares finish
    main handle *myid;
    myid = MyChareHandle();

    // Ask the accumulator to send its total
    // to the PrintResult entry point
    total->CollectAccValue(PrintResult, myid);
  }

  PrintResult(MsgAccCount * result)
  {
    CPrintf("The # of primes is:%d.",
                                result->data);
    CharmExit();
  }
};
```

Figure 6 : `main` chare of the Primes module

contain the number of slots in the histogram. Although we include the definition of this accumulator in the Primes module for illustration, such commonly used accumulator subclasses are available and can be reused from the Charm++ system library.

Figure 6 shows the `main` chare definition. The `main` entry of this chare reads the input from the user and creates an instance of the accumulator (`AccCount`) object. The handle to this accumulator instance is stored in `total`, which can be accessed uniformly on all processors. Similarly, a message `msg` is used to create an instance of the `PrimesChare`. Note that all parallel objects are created by providing an initial message for them to process. Thus, for example, each chare instance processes this initial creation message before it processes any other messages directed to it. Charm++ supports *quiescence detection*. If the `Quiescence` entry point is defined in the `main` chare, the quiescence detection algorithm is activated. When there remain no messages to process (i.e. all produced messages have been processed), and all processors are idle, the system calls this entry point. In the Primes program, this serves the purpose of detecting that the tree of chares generated by the `PrimesChare` is exhausted. At this point, the `main` chare requests the accumulator object `total` to return its final value to the `PrintResult` entry point, which simply prints this count and terminates the overall program execution.

Figure 7 shows the `PrimesChare` definition, which happens to have no local variables, and only one entry point. The code at this entry point checks if the range given to it is small enough. If so, it calls `seqPrimes` and adds the count of primes in the range to `total` via the `Accumulate` call. Otherwise, it simply divides the range into two, and creates chares for each sub-range. In either case it calls `ChareExit` to relinquish the resources occupied by the chare instance.

```
chare class PrimesChare
{
public:
  PrimesChare() {}

entry:
  Goal(RangeMsg * msg1)
  {
    int L = msg1->Low;
    int H = msg1->High;

    if ((H-L+1) > LENGTH)
    { // Distribute the halves of this range to
      // two new chares
      int Mid = L + (H-L+1)/2;
      RangeMsg *msg2 = (RangeMsg *)
                         new_message( RangeMsg);
      msg2->Low = Mid;
      msg2->High = H;
      msg1->High = Mid-1;
      // Reuse msg1 ; msg1->Low == L already
      new_chare(PrimesChare, Goal, msg1);
      new_chare(PrimesChare, Goal, msg2);
    }
    else {
      int count = seqPrimes(L,H);
      delete_message(msg1);

      // Accumulate local count in the
      // global total accumulator
      total->Accumulate(count);
    }
    ChareExit();
  }
};

}; // End of module Primes
```

Figure 7 : the PrimesChare