RC 20265 (04/27/95)
Computer Science/Mathematics

IBM Research Report

Current Proposals for Parallel C++

Howard L. Operowsky

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Current Proposals for Parallel C++

Howard L. Operowsky
IBM T. J. Watson Research Center*†

Abstract

As powerful microprocessors become increasingly cheaper, there is a growing trend towards distributed processing on groups of these processors, both in the forms of networks of workstations, as well as distributed-memory computers where each processing node has its own local memory, and there is no global memory. Parallel applications are constructed by starting a program on each of the nodes in the network or distributed-memory computer, and having the individual programs communicate by sending messages to each other. Common message-passing libraries, such as MPI, PVM, and P4, are available to enhance portability across different platforms.

There is an obvious analogy between sending messages between the nodes involved in these parallel applications and sending messages between objects in an object-oriented application. But programming with message passing libraries is cumbersome. Programs are easier to write when the programmer can use a familiar language.

With this in mind, there has been much work on extending object-oriented languages to support parallelism, especially C++. However, there are many differences between the various proposals. Some use language extensions and require compiler support, while others use class libraries. Some support control parallelism, while others support data parallelism. In this presentation, we describe the features of some competing proposals for adding parallelism to C++. We also code a simple example in each of the proposals.

*On assignment from IBM RISC System/6000 Division

†Also to be published by CASCON'95

1 Introduction

With the decreasing cost of hardware, and the growing movement towards "open" software, computer users are moving their applications to distributed client/server environments, workstation clusters, and massively parallel computers. In addition, more manufacturers are producing symmetric multi-processors(SMPs) as another means of providing increased performance.

Unfortunately, it is not a simple matter to write applications that can fully exploit multiprocessor environments. In addition to the typical errors found in serial code, parallel code is plagued by errors due to race conditions that are frequently not repeatable and thus very difficult to correct.

These difficulties have been known for some time, and researchers have long been trying to simplify the process. Since most of the performance demands have been pushed by numerically intensive applications, most of the previous work has focussed on FORTRAN, and much work has been done to improve the compiler's ability to recognize and extract program segments that can operate concurrently. With the emergence of distributed memory multi-computers, FORTRAN users have again pushed FORTRAN forward in the development of High-Performance FORTRAN to exploit various architectures in a single manner, via data parallelism.

With the growing popularity of C and C++, attention has been turning to providing parallelism in those languages as well. Furthermore, the message-passing paradigm used for communication on distributed-memory multiprocessors is quite similar to the message-passing between objects in object-oriented applications. Thus, many efforts are under way to parallelize various object-oriented languages including Smalltalk and Eiffel, as well as C++.

In this paper, we will review five parallel C++ efforts. In section 2, we review some general ways in which the important issues of parallelism, communication, and synchronization are being addressed. Sections 4 through 8 discuss how these issues are addressed by the ABC++, CC++, Charm++, Mentat, and pC++ proposals. They were chosen because they all support distributed-memory computing, they illustrate a wide range of both language design, and implementation options, and because we were able to verify the results of our sample programs. We compiled and executed each of the sample programs presented here, except for the Mentat program, which was verified by the Mentat group. Note, however, that our goal in writing these code samples was not to write the shortest possible program, but to illustrate features indicative of the given language. The basis for the sample is outlined in Section 3. Section 9 provides a summary and some additional areas of investigation.

2 Issues

Any language designed to support the development of parallel applications must answer certain basic questions:

1. How are concurrent units of execution identified?
2. How do these concurrent units communicate with each other?
3. How do these concurrent units synchronize their activities?

In this section, we summarize some of the general approaches that are being taken to answer the above questions.

2.1 Parallelism Model

There are primarily two models of parallelism being promoted today: *control* parallelism and *data* parallelism. Historically, parallelism has meant *control* parallelism: parallelism achieved through the concurrent (or apparently concurrent) execution of multiple independent agents commonly called *tasks* or *processes*. Control parallelism can be further classified by the amount of overhead required to establish the concurrent unit of execution.

In object-oriented programs, the most common approach has been to provide special objects that are capable of executing in parallel with other such special objects, either on the same or different physical processors. This approach nicely parallels the model provided by distributed memory clusters, where independent entities execute asynchronously in separate address spaces, and communicate via messages along well-established public channels. These objects are typically identified through use of a new keyword, or by virtue of being derived from a special base class. Generally, these objects have their own threads of control, and are therefore commonly referred to as *active* objects. They are created dynamically through use of analogues of the **new** operator.

The second model of parallelism employed is *data* parallelism. In data parallelism, data objects are gathered into composite objects called *aggregates* or *collections*, which are distributed among the physical processors. There is a single logical thread of control. The code typically looks like serial code, except for the directives describing the data distribution. The compiler is responsible for splitting the activity across the available processors, transferring data to the appropriate concurrent threads, and returning results as appropriate. The programmer is able to reduce communication overhead by describing how various data objects should be distributed among the processors. Function invocation is a synchronous event, and control doesn't continue to the next source instruction until all work on the collection has completed.

In addition, methods may be applied concurrently to some or all of the component objects, or elements, of an aggregate. This is sometimes called *element parallelism.* Individual elements are referenced through subscript notation, and element subgroups have been specified with FORTRAN style *section notation* as in pC++, or through *index vectors* as in [13].

2.2 Communication Facilities

The primary communication paradigm in distributed computing is *message-passing*, and many independent library packages, such as MPI and PVM, have been developed to support application portability across platforms. The benefit of supporting message-passing within the language is that it can be made more familiar to the C++ programmer. In this case, `message` classes are added to the language, and tasks communicate by sending message objects to "mailbox" entry points. Unlike method invocations, message-passing is generally one-way, and to obtain a return value, the sender must specify a return mailbox.

A more natural approach for the C++ programmer is to pass arguments through method invocation. The compiler then becomes responsible for collecting or *marshalling* the arguments, transferring them to the remote object using appropriate underlying mechanisms, and recreating the argument list for the called method in the remote object.

When the model is control parallelism, active objects are created dynamically, and methods are identified through use of the pointer returned by the allocator. In the data parallelism model, the compiler generally follows an "owner-computes" rule and initiates activity on the processor that owns the data that will be modified.

Regardless of the kind of parallelism, pointers within passed arguments pose a problem. In some cases, they are simply forbidden. In others, however, facilities are provided to allow the programmer to compress data before it is sent, and expand it upon receipt but before it is used by the remote task. UC++ [12], for example, allows the programmer to process arguments by overloading the *iostream* library operators $<<$ and $>>$.

A third communication mechanism is the use of shared memory. Although quite simple to support on SMPs, where each processing node has equal access to the shared global memory, it is problematic in distributed memory environments, and C++ facilities are generally restricted. For example, use of **static** variables is not supported when the class objects are on different processors, but may behave as expected when the objects are on the same processor. Global and file scope variables are restricted and frequently may be modified only by the main task in a parallel application, when they can be shared across tasks at all.

Another approach is to provide additional facilities to access data across processor boundaries. One method is to provide *global* pointers, which transcend processor boundaries. An object can grant another object access to its local data by passing to it a unique system-wide address, commonly composed as a {processor node, local address} pair. Such addresses are considered opaque objects, and normal pointer arithmetic is not supported. Of course, when global pointers are supported, the programmer assumes responsibility for synchronizing accesses to the data in question.

In an attempt to overcome such synchronization problems, some languages provide simulated shared objects, where the run-time system assumes responsibility for providing consistency. This is achieved by adding a monitor-like facility to the language. The multiple-readers, single-writer approach is typically followed. A task (or thread) is granted read/write access to a simulated shared object only when no other task or thread has access. When access is granted, the task operates on a local copy of the object. When the object is released, the run-time system automatically broadcasts its new value to all other processors participating in the computation. To enhance performance of applications that can tolerate "stale" data, the local copy is generally still accessible by the task. However, unless it goes through the monitor, the task cannot be sure that it has the latest value for that object, and none of the changes that it makes will be acknowledged by the system. On the contrary, local changes may be overwritten if other tasks modify the simulated shared object through the monitor.

2.3 Synchronization Facilities

In data parallel programming languages, synchronization is implicitly handled by the run-time system. In message-passing environments, data is generally not shared, and program activity is synchronized by receipt of messages. Additional facilities are provided when control parallelism is employed.

One approach is through the use of *future* variables, which combine synchronization with data access. When a process attempts to read the value of a future variable before it has been set, that process is suspended until the future variable is given a value. Future variables provide a very elegant means of supporting asynchronous execution of concurrent objects. Methods that are invoked asynchronously indicate completion by returning a result into a future variable, allowing the caller to continue while the callee performs the requested operation. When the callee completes, the run-time system places the returned value into the future variable so that it is available when the caller requests it. If however, the caller needs its value before it has been set, the caller is automatically suspended until the future variable value is available. Some languages distinguish among different kinds of references. For example, some languages allow uninitialized future variables to be passed as arguments to other concurrent tasks without suspending the caller, because it is recognized that the caller has no need of the variable's value. The called method, in turn, would only be suspended if it attempted to use the value of the future variable argument before it was available.

When active objects are single-threaded, instance variables can only be accessed by the single thread. However, when active objects are multi-threaded, instance variables are shared and threads have to synchronize their access to them. One way of accomplishing this is through the use of *atomic* methods. When an atomic method is invoked, no other thread may invoke any other atomic function for that object. By changing shared variables only within atomic functions, a thread is guaranteed that only one thread may change a shared variable at any time, and that the change will be performed atomically from the perspective of all other threads.

Figure 1 illustrates the use of atomic methods in CC++. In that example, only one thread may add to `data` in any `syncdata` object. However, because **getdata** is not atomic, other threads may read `data` while it is being updated.

Another means of restricting access to instance methods is by providing mechanisms by which the programmer can control when methods may be invoked. This is called *Selective Method Acceptance.*

```
class syncdata {
  double data;
public:
  atomic double add(double d) {
    // Synchronize write access to "data"
    data += d;
    return data;
  }
  double getdata(void) {return data;}
};
```

Figure 1: Atomic Methods in CC++

```
// Producer/Consumer Example
mselect{
             : maccept count();   break;
  [!empty]   : maccept consume(); break;
  [!full]    : maccept produce(); break;
}
```

Figure 2: Selective Method Acceptance in Mentat

Figure 2 illustrates the Ada-like select clause provided by Mentat. This example illustrates one way that producer/consumer example may be coded. Each method in the Mentat **mselect** clause may optionally be guarded by a boolean expression. Unguarded methods may always be invoked; guarded methods may be invoked only when the guard expression is non-zero. If one attempts to invoke a method while it is ineligible, Mentat's run-time system suspends the thread until the method becomes eligible.

In Figure 2, requests to produce or consume items are guarded by the `empty` and `full` variables which describe the state of the item buffer. The **count** method is unguarded, and so will always accept requests.

3 Example

In order to illustrate the languages described, a simple example is coded in each language. The example (which has also been used to survey parallel FORTRAN dialects [6]) is to compute the value of π using the equation

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

The serial program is presented in Appendix A.1. The source of parallelism is the for-loop on line 11, which approximates the above integral by computing

$$\frac{1}{N}\sum_{i=1}^{N}\frac{4}{1+{x_i}^2}$$

over N equal intervals, where x_i is the midpoint of the *ith* interval. In the parallel versions, we distribute the computation among three processors to illustrate basic task creation, communication, and synchronization.

4 ABC++

Unlike the other languages discussed here, ABC++[1, 15] is implemented as a class library, rather than a compiler, and there is heavy use of C++ templates to provide its function. ABC++ supports coarse-grained control parallelism via objects derived from the system-provided `Pabc` base class. Active objects of type `task-type` are allocated through the **Pabc_create()** function , which returns values of type `Pabc_pointer<task-type>`. The user may specify which processor the active object should execute on in the **Pabc_create()** call. Once allocated, ABC++ active objects never migrate to other processors.

Methods within active objects may be invoked synchronously or asynchronously by calling one of four library functions, and passing as its arguments an `Pabc_pointer` pointing to the desired active object, the method's name, and arguments to the method. If the method returns a value of type `return-type`, that value must be assigned into a local future variable of type `Pfuture<return-type>`. Active objects are single-threaded; concurrent requests to the same object are queued and processed sequentially and non-preemptively. The **Paccept()** and **Paccept_any()** functions provide selective method acceptance.

Referring to the ABC++ example on page 10, the active class, `calcpi`, is declared on line 4, and three `calcpi` class objects are allocated in the for-loop on line 26 and are invoked asynchronously in the for-loop on line 30. Each instance computes the partial sum for one-third of the intervals. The partial sums are stored into future variables upon return, and referenced in the for-loop on line 39.

In addition to communication via argument passing, ABC++ supports simulated shared memory through *parametric shared regions*(PSRs). One active object creates the PSR and provides access to it via *handles*. Handles also provide access rights to the shared object. Users synchronize access to a shared object by obtaining a *locked* handle for it, or creating a locked handle from an unlocked handle that is already owned.

5 CC++

CC++ (Compositional C++)[8, 3] enhances C++ for parallelism by adding five new constructs to the language. CC++ supports both fine-grained and coarse-grained parallelism, although the basic unit of parallelism is the thread. Coarse-grained parallelism is provided by first creating *logical processor objects* on remote processors, and then creating local threads to execute public methods of those processor objects.

CC++ is the only language discussed here that supports multi-threaded objects, and provides three constructs for explicitly defining local parallelism: the **par** block, which is comparable to the *cobegin* ... *coend* construct and causes each statement in the block to be executed concurrently by a separate thread of control; the **parfor** statement, which is similar to a for-loop except that each iteration is executed concurrently by a separate thread; and the **spawn** statement, which allows the caller to invoke a function asynchronously. Unlike the **par** block and **parfor** statement, which both provide barrier synchronization before control passes to the next instruction, the **spawn** statement itself provides no indication that the called program has completed.

Threads execute within the context of logical processor objects, which are self-contained objects that may reside on physical processors.

Logical processor objects are specified using the **global** keyword, and are implemented as separate executables by the compiler. Logical processor objects are created using the **new** operator, which returns a **global** pointer to the new object. The **new** operator is also the means by which the programmer specifies which physical processor should support the new logical processor object. Logical processor objects are typed just like all other C++ objects, and the type is checked by the run-time system at the time of creation. Logical processor objects may have both public data and methods that are accessible via the `global` pointer returned from **new**. Remote method invocation is synchronous. To obtain asynchronous execution, the user must first create a thread to invoke the method.

In our CC++ example beginning on page 12, we define the `calcpi` logical processor class on line 3, and create three instances on lines 28–34. **Compute()** is a public method that will compute the partial sum for that logical processor. The **parfor** loop on line 39 generates three threads, each of which invokes **compute** on a different logical processor. Control does not continue to line 43 until all three threads complete.

In addition to the barrier synchronization provided by the **par** and **parfor** constructs, and **atomic** methods, which were described earlier, CC++ provides write-once future variables, called **sync** variables.

6 Charm++

Charm++ [9, 5] is an extension to C++ based on the Charm run-time system, which supports a "latency-tolerant" message-driven style of programming. Parallel computations are defined in terms of medium-grained units called *chares*, which contain local variables, private functions, and external interfaces called *entry points*; public data members are not supported. Chares are invoked by calling an entry point with at most one parameter, called a `message`. All calls are non-blocking, and one-way. A chare that needs to return a value does so by sending a message to an entry point in the original calling chare. Charm++ allows the programmer to provide *pack* and *unpack* for use by the run-time system to marshal and demarshal message data. The Charm philosophy is that the run-time system is in the best position to determine the best use of computing resources, and generally determines which processor should execute a particular chare. In addition, Charm++ provides for various user-selectable dynamic load-balancing strategies, and prioritized messages.

There is a second variant of the chare, called the *branched chare*, which is replicated on all processors participating in the computation, and which supports data-parallelism. Unlike the standard chares described above, when a branched chare is called, control passes by default to the local copy (although it is possible to explicitly pass control to any copy). Because branched chares reside on the same processor as their callers, branched chares may have public data members.

In place of the global `chare` variables, Charm++ provides several abstract information-sharing mechanisms. These mechanisms are:

- read-only objects that are set once by the **main** chare after execution begins and may thereafter be read by all other chares at the cost of a local access.
- write-once objects that are similar to read-only objects, except that they may be set *once* by any chare.
- accumulator objects that may be updated many times by any chare running on any processor, but may only be read *once*. Thus, accumulator objects are implemented for high-speed update.
- monotonic objects that are similar to accumulator objects except that they must take on monotonically increasing values. However, monotonic objects may be referenced multiple times. These objects are provided primarily for branch and bound calculations. Any value referenced may not be the "best" value currently, but the system propagates the best value as soon as possible.

- Distributed tables that are keyed records distributed across the processors participating in the calculation. These objects support insert, find, and delete operations.

In the example that begins on page 14, the `calcpi` chare class is declared on line 34, and three copies are created on line 80. The preceding code in the loop allocates and initializes the message, which is passed to **calcpi::Compute()**. Control continues to line 84, which tells the Charm++ run-time system to invoke **cleanup()** on line 59 after all messages in the pool have been processed. That will occur when all partial sums have been computed. When **Compute** is invoked, it reads its interval range from the message, but gets the interval width from the `readonly` variable defined on line 29 and initialized by the main chare on line 69. **Compute** stores its partial sum into the `Sum` accumulator variable, at the cost of a local reference, and exits.

When all three chares have computed their partial sums, **main::cleanup** is invoked, which sends a message to the **CollectValue** entry point to send the total of the partial sums to **FinalResult()** on line 52.

7 Mentat

Mentat [10, 14] provides a model of coarse-grained control parallelism, but takes a much more aggressive position in managing parallel activity than the other languages discussed here. It shares the Charm++ philosophy that once the programmer defines the units of parallelism within the program, the compiler and run-time system are best equipped to manage the execution; but uses a *macro data flow* model [4] to manage scheduling, concurrency, and synchronization. As a result, there are fewer language extensions.

The programmer identifies coarse-grained active objects through use of the **mentat** keyword on the class declaration. There are three varieties of **mentat** classes: **regular**, **persistent**, and **sequential**. Regular mentat classes do not preserve state between method invocations, and are created or destroyed dynamically at the discretion of the Mentat run-time system based on the requirements of the application. In the sample code on page 16, the `calcpi` class is defined to be a **regular mentat** class on line 2, and as a result, there are no explicit calls for active object creation in the main routine. Mentat objects are referenced via mentat *names* much as `global` pointers are used in CC++, or `Pabc_pointers` in ABC++. Line 26 defines `my_pi` to be an array of three `mentat` names.

Persistent and sequential mentat classes do preserve state between invocations, and once allocated remain on the same processor. Persistent class objects are created and bound to mentat names using the **create()** function. The user may either specify which processor the object should be created on, or allow Mentat to decide. Mentat will choose the processor that has the most processing power currently available. Once created, active objects do not migrate. A sequential mentat class is a persistent class with the additional function that the user can control the order to member function invocation. Ordinarily, `mentat` class methods are invoked by the run-time system as soon as all their arguments are available.

Because Mentat follows a data-flow model, method invocation is asynchronous, and variables that receive function results are implicitly treated as futures. A method multi-casts its results to all methods that are dependent on the data, which may or may not include the method's original caller, via the **rtf()** function. (The current implementation requires that a return statement be coded as well, due to the requirements of underlying C++ compilers.) This is accomplished by having the run-time system do data flow analysis based on data produced by the compiler.

All mentat objects are disjoint, and only methods can be public. Mentat objects communicate by argument-passing only.

8 pC++

pC++ [11, 2] supports the data parallelism model rather than control parallelism. The programmer defines **collection** objects, which are derived from the system-provided `SuperKernel` base class, and describes how they are to be distributed among processors in two steps, using **Distribution** and **Align** objects. Distribution objects define a logical coordinate system on **Processor** objects (the set of processors available for a computation), and Align objects map collection elements onto that coordinate system. Two elements mapped to the same coordinate will physically reside on the same processor. This process is modelled after that employed in High-Performance FORTRAN, and the alignment mapping uses the same syntax. In the pC++ example in Appendix A.6 on page 18, we obtain parallelism on lines 39–42 by creating a collection object, `proc`, with three `calcpi` objects as elements, and mapping each element to a different processor. The **compute()** method is invoked on line 47, and causes each `calcpi` element to compute a different partial sum. Note that, as is typical in data parallel languages, one cannot tell from the call site whether the computation will be performed using parallelism or not. However, pC++ does not support nested parallelism; i.e., threads cannot generate additional threads. Each method invocation is sequential and non-preemptive.

In addition to the methods belonging to the collection object, pC++ supports element parallelism. pC++ defines a new **MethodOfElement** section through which element methods can be invoked on collection objects. The **virtual** qualifier on a method in the **MethodOfElement** section indicates that the method is an element method. In our example, we define these routines in the `calcpi` class on lines 10–11, reference them in the `C` **Collection** class on lines 20–21, and use them to access local data on lines 33 and 50. Functions in the **MethodOfElement** section also have access to information about the collection's topology, such as the number of processors running the application. In our example, the **compute()** routine is placed in the MethodOfElement section to allow it to reference the `dim1size` variable and obtain the number of elements in the collection. `ThisCollection` is inherited from the `SuperKernel` base class and points to the current collection object, just as `this` points to the current object in C++.

One deficiency with collection methods, such as **compute** in our example, is that when called from the main routine, all threads must return the same value, or the result is undefined. Reduction functions are promised for the next release to alleviate this problem. In our example, we used the **get_partial()** method to obtain the partial results from each element.

As already observed, caller and callee communicate via normal argument passing. If nonlocal data is required in a computation, the run-time system transfers the data automatically. One constraint is that when a remote element is referenced through subscript notation, a single location is used for the local copy. Thus, when two elements are explicitly referenced, the second will overwrite the first.

pC++ also supports global variables in that file scope variables may be set by the main routine and read by all the threads, but threads are not allowed to update such global variables. This is illustrated in our example by virtue that we only pass two arguments to **compute()**. In addition, there are strong indications that pC++ will incorporate CC++-style `global` pointers.

9 Summary

In this overview, we have focussed on those issues that must be considered by any language designed to support parallel programming: how parallelism is expressed, how concurrent objects communicate with each other, and how concurrent objects synchronize their activities. As even this brief survey indicates, there are many current proposals, and more practical experience is required before a consensus will be reached on the most effective set of constructs. For example, there is a growing sense that languages should support both control parallelism and data parallelism. Several conclusions can be drawn however. One is that objects do provide a viable means of expressing parallelism. Second, standard procedural call interfaces have obvious advantages in readability and ease-of-use over CHARM++'s one-way message passing. In particular, the message overhead associated with the accumulator variable used in our test case complicated the program greatly. Although the accumulator variable was appropriate to our application, a simpler program could have been written had we chosen not to use it. Third, some form of memory-sharing between objects will be provided. Even though these languages were chosen because they supported distributed-memory computers, three already support memory-sharing, and a fourth has indicated its intention to provide global pointers.

In addition, there are other questions that we have not addressed here and which require further investigation. Proper support for parallel I/O has not been adequately addressed by most languages. There is widespread support for marshalling and demarshalling simple argument types, but support for complex data structures is also required. The *inheritance anomaly* [7] affects our ability to reuse components in parallel applications. Finally, we have not addressed the question of performance. This omission, which resulted from our inability to access a single hardware/software platform supported by all of the various languages discussed here, will hopefully soon be corrected.

A Sample computations of π

A.1 Serial computation of π

```
#include <iostream.h>
#include <math.h>

inline double f(double x) { return 4.0/(1.0+x*x); }

void main() {
  double width, error, pi, EstPi=0, sum;
  int i, intervals;

  cout << "Enter number of intervals: "; cin >> intervals;
  width = 1.0/intervals;        // Compute width of each interval

  // Calculate value of pi, using mid-point of each interval
  // Later examples will execute this loop in parallel
  for(i=1; i<=intervals; i++) {
    sum = sum + f((i-0.5)*width);
  }
  EstPi = sum * width;

  pi = 4.0 * atan(1.0);      // Get "real" value of pi
  error = EstPi-pi;          // Compare computed value to "real value"
  cout << "PI=" << pi << "  ESTPI=" << EstPi
                      << "  ERROR=" << error << endl;
}
```

A.2 ABC++ Calculation of π

```
#include <ABC++.h> // Get ABC++ definitions
#include <math.h>

// calcpi will be an active object
class calcpi : public Pabc {
  inline double f(double x) { return 4.0/(1.0+x*x); }
 public:
  double compute(const double, const int, const int);
};
```

```
double
calcpi::compute(const double w, const int low, const int high) {
  double sum=0;
  int i;
  for(i=low; i<=high; i++) {
    sum = sum + f((i-0.5)*w);
  }
  return sum;
}

void main() {
  Pinit();
  double width,error,pi,EstPi=0;
  int i,intervals;
  Pfuture<double> partial[3]; // Will hold partial sums
  // Will define 3 active objects to compute pi
  Pabc_pointer<calcpi> my_pi[3];

  // Create the 3 active objects, round robin
  for(i=0; i<3; i++)
    Pabc_create(Pproc_set.pick(), my_pi[i]);

  cout << "Enter number of intervals: "; cin >> intervals;
  width = 1.0/intervals; // Compute width of interval

  for(i=0; i<3; i++) {
    // Initiate 3 asynchronous invocations of calcpi::compute
    partial[i] = Ppar_value(my_pi[i],   // ABCpointer to active object
                     calcpi::compute,   // Method within object
                    //Method Arguments
                       width, 1+(i*intervals)/3, (i+1)*intervals/3
                   );
  }
  pi = 4.0 * atan(1.0);  // Get "real" value of pi while computing sums

  for(i=0; i<3; i++) // Pick up partial sums; may wait
    EstPi = EstPi + partial[i];
```

```
  EstPi = EstPi * width;        // Compute approximation
  error = EstPi-pi; // Compare computed value to "real" value
  cout << "PI=" << pi << "  ESTPI=" << EstPi
                    << "  ERROR=" << error << endl;
  Pexit(0); // Clean up and exit
}
```

A.3 CC++ Calculation of π

```
// calcpi.h

#include <iostream.h>

global class calcpi {
  inline double f(double x) { return 4.0/(1.0+x*x); }
 public:
  double compute(const double, const int, const int);
};

// calcpi.cc++

#include "calcpi.h"

double
calcpi::compute(const double w, const int low, const int high) {
  double sum = 0;
  int i;
  for(i=low; i<=high; i++) {
    sum = sum + f((i-0.5)*w);
  }
  return sum;
}

// main program

#include <iostream.h>
#include <math.h>
#include "calcpi.h"

int main() {
```

```
  double width,error,pi,EstPi=0;
  int i,intervals;
  double partial[3];
  calcpi *global my_pi[3]; // Active objects used to calculate pi
  proc_t proc[3];
  proc[0]=proc_t("cc++-calcpi.out", "fc10.watson.ibm.com");
  proc[1]=proc_t("cc++-calcpi.out", "fc11.watson.ibm.com");
  proc[2]=proc_t("cc++-calcpi.out", "fc12.watson.ibm.com");

  for (i=0; i<3; i++) {
    my_pi[i] = new (proc[i]) calcpi;
  }

  cout << "Enter number of intervals: "; cin >> intervals;
  width = 1.0/intervals; // Compute width of interval

  // Initiate 3 asynchronous invocations of calcpi::compute
  //  (Note i is local to parfor loop)
  parfor(int i=0; i<3; i++) {
    partial[i] =
      my_pi[i] -> compute(width, 1+(i*intervals)/3, (i+1)*intervals/3);
  }

  pi = 4.0 * atan(1.0); // Get "real" value of pi

  for(i=0; i<3; i++) // Pick up partial sums
    EstPi = EstPi + partial[i];

  EstPi = EstPi * width;        // Compute approximation
  error = EstPi-pi; // Compare computed value to "real" value
  cout << "PI=" << pi << "  ESTPI=" << EstPi
                    << "  ERROR=" << error << endl;

  delete my_pi[0], my_pi[1], my_pi[2];

  return 0; // cc++ currently must return an int
}
```

A.4 Charm++ Calculation of π

```
extern "C" {
#include <math.h>
#include "quiescence.h"
}

message computeMsg {  /* parameters to compute routine */
  int low;
  int high;
};

// Message used by accumulator constructor to initialize data member
message InitAccumMsg { double value; };

// Data member of accumulator object
message AccumDataMsg { double value; };

// Accumulator object used by calcpi chares to report partial sums
accumulator class AccumSum {
  AccumDataMsg *DataMsg;

public:
  // Update local copy of accumulator object
  void Accumulate(double partial)   {DataMsg->value += partial;}

  // Combine 2 accumulator objects from different processors
  void Combine(AccumDataMsg *OtherAccum){
    DataMsg->value += OtherAccum->value;
  }

  AccumSum(InitAccumMsg *init) {
    DataMsg = (AccumDataMsg*) new_message(AccumDataMsg);
    DataMsg -> value = init -> value;
  }
};

// Readonly variables only allowed at file scope
readonly double width;   /* Rectangle width  */
```

```
// Handle for the accumulator for the partial sums.
// It is a readonly variable to provide global access to all processors
readonly AccumSum handle PartialSums;

// Chare used to calculate partial sum
chare class calcpi {
private:
  double f(double x) {return 4.0/(1.0+x*x);}

entry:
  void compute(computeMsg * compmsg) {
    double sum=0;
    int i;
    for(i=compmsg->low; i<=compmsg->high; i++) {
      sum = sum + f((i-0.5)*width);
    }
    PartialSums->Accumulate(sum);
  }
};

// Main chare starts program
chare class main {
  double error,pi,EstPi;

entry:
  // Compute and print estimate
  void FinalResult(AccumDataMsg *SumPartialsMsg) {
    EstPi = SumPartialsMsg->value * width;
    error = EstPi-pi;
    CPrintf("PI=%f    ESTPI=%f   ERROR=%f\n", pi, EstPi, error);
    CharmExit();
  }

  // Invoked when all other work is done
  void cleanup(QUIESCENCE_MSG *m) {
    // Request sum accumulator to send its value to FinalResult
    PartialSums -> CollectValue(&(main::FinalResult), mainhandle);
  }
```

```
public:
  main() {
    int i,intervals;
    computeMsg *cm[3];
    InitAccumMsg *acc_msg;

    CPrintf("Enter number of intervals: "); CScanf("%d", &intervals);
    width = 1.0/intervals;

    // Create the accumulator and initialize it
    acc_msg = (InitAccumMsg *) new_message(InitAccumMsg);
    acc_msg -> value = 0;
    PartialSums = new_accumulator(AccumSum, acc_msg);

    // Create 3 chares to compute partial sums
    for (i=0; i<3; i++) {
      cm[i] = (computeMsg*) new_message(computeMsg);
      cm[i]->low = 1+(i*intervals)/3;
      cm[i]->high = (i+1)*intervals/3;
      // Compute partial sum and accumulate in Sum accumulator object
      new_chare(calcpi, &(calcpi::compute), cm[i]);
    }

    pi = 4.0 * atan(1.0); // Get "real" value of pi

    // Go to cleanup when above chares complete
    CStartQuiescence(&(main::cleanup), mainhandle) ;
  }
};
```

A.5 Mentat Calculation of π

```
// calcpi.h

regular mentat class calcpi { // calcpi will be an active object
  inline double f(double x) { return 4.0/(1.0+x*x); }
 public:
  double compute(double, int, int);
};
```

```
// calcpi.C

#include "calcpi.h"

double
calcpi::compute(double w, int low, int high) {
  double sum=0;
  int i;
  for(i=low; i<=high; i++) {
    sum = sum + f((i-0.5)*w);
  }
  rtf(sum); // rtf() required for all mentat functions
  return sum; // "return" required by underlying compiler
}

// main program

#include "calcpi.h"
#include <math.h>

void main() {

  double width,error,pi,EstPi=0;
  int i,intervals;
  double partial[3]; // Will hold partial sums
  calcpi my_pi[3]; // Mentat names for 3 active objects

  cout << "Enter number of intervals: "; cin >> intervals;
  width = 1.0/intervals; // Compute width of interval

  // Initiate 3 asynchronous invocations of calcpi::compute
  // (Regular mentat objects do not need to created by the programmer)
  for(i=0; i<3; i++)
    partial[i] =
      my_pi[i].compute(width, 1+(i*intervals)/3, (i+1)*intervals/3);

  pi = 4.0 * atan(1.0); // Get "real" value of pi simultaneously

  for(i=0; i<3; i++) // Pick up partial sums; may wait
```

```
    EstPi = EstPi + partial[i];

  EstPi = EstPi * width;        // Compute approximation
  error = EstPi-pi; / Compare computed value to "real" value
  cout << "PI=" << pi << "  ESTPI=" << EstPi
                      << "  ERROR=" << error << endl;
}
```

A.6 pC++ Calculation of π

```
#include "pcxx.h"
#include <iostream.h>

// Global variables that are set by main and
//   read by Collection elements
int    intervals;
double width;

class calcpi {
  double sum=0;
public:
  double get_partial(void) {return sum;}
  void set_partial(double p) {sum=p;}
};

#include "kernel.h"

Collection C : SuperKernel {
public:
  C(Distribution *, Align *);
MethodOfElement:
  double f(double x) { return 4.0/(1.0+x*x); }
  void compute(void);
  virtual double get_partial(void);
  virtual void set_partial(double p);
};

C::C(Distribution *D, Align *A) : SuperKernel(D,A) {}

void C::compute(void) {
```

```
  int i;
  double sum=0;

  // compute() must be in MethodOfCollection to reference dim1size
  int parts = ThisCollection->dim1size;

  int low = 1+Ident*intervals/parts;
  int high = (Ident+1)*intervals/parts;

  for(i=low; i<=high; i++)
    sum = sum + f((i-0.5)*width);

  set_partial(sum);
  return;
}

Processor_Main() {
  double error, pi, EstPi=0;
  int i;

  Processors P;
  Distribution D(3, &P, BLOCK);
  Align A(3, "[ALIGN(V[i],D[i])]");
  C<calcpi> proc(&D,&A);

  pi = 4.0 * atan(1.0);
  cout << "Enter number of intervals: "; cin >> intervals;
  width = 1.0/intervals;

  // Invoke compute for all elements in collection
  proc.compute();

  // Collect values from each element (1 per processor)
  for(i=0; i<proc.dim1size; i++)
    EstPi += (*proc(i)).get_partial();

  EstPi *= width;
  error = EstPi-pi;
  cout << "PI=" << pi << "  ESTPI=" << EstPi
```

```
                          << "  ERROR=" << error << endl;
  return 0;
}
```

References

[1] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, and G. R. Gao. ABC++: Concurrency by inheritance in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[2] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3):11–20, Fall 1993.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical report, California Institute of Technology, March 1993.

[4] Andrew S. Grimshaw. The Mentat computation model data-driven support for object-oriented parallel processing. Technical Report CS–93–30, University of Virginia, 1993.

[5] L.V. Kale and Sanjeev Krishnann. Charm++ : A portable concurrent object oriented system based on C++. *ACM Sigplan Notices*, 28(10):91–108, Fall 1993. Proceedings of OOPSLA-93, September 26-October 1, 1993, Washington, D.C.

[6] Alan H. Karp and Robert G. Babb II. A comparison of 12 parallel Fortran dialects. *IEEE Software*, pages 52–67, December 1988.

[7] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[8] CC++ Home Page. *http://www.compbio.caltech.edu/ccpp/.*

[9] Charm++ Home Page. *http://charm.cs.uiuc.edu/.*

[10] Mentat Home Page. *http://uvacs.cs.virginia.edu/~mentat/.*

[11] pC++ Home Page. *http://www.extreme.indiana.edu/sage/index.html.*

[12] T. O'Brien, G. Roberts, M. Wei, R. Winder, D. Titcombe, M. Plaxton, A. McEwan, and J. Poole. *UC++ V1.3 Language and Compiler Documentation.* London Parallel Applications Centre, January 1995.

[13] Thosmas J. Sheffler and Siddhartha Chatterjee. An object-oriented approach to nested data parallelism. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 203–210, 1995.

[14] University of Virginia The Mentat Research Group. *The Mentat 2.8 Programming Language Reference Manual*, 1995.

[15] Gregory V. Wilson and William G. O'Farrell. *An Introduction to ABC++.* IBM Canada Ltd., 1995.