# Object-Parallel Programming with pC++

*Peter Beckman  Dennis Gannon*
*Jacob Gotwals  Neelakantan Sundaresan*
*Shelby Yang*

**CRPC-TR95608**
**November 1995**

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

# Object-Parallel Programming with pC++ *

Shelby X. Yang, Dennis Gannon, Peter Beckman,
Jacob Gotwals, Neelakantan Sundaresan

November 17, 1995

**Abstract**

pC++ is a data parallel extension to C++ that is based on the concept of *collections* and *concurrent aggregates*. It is similar in many ways to newer languages like ICC++, Amelia and C** in that it is based on the application of functions to sets of objects. However, it also allows functions to be invoked on each processor to support SPMD-style libraries and it is designed to link with HPF programs. pC++ currently runs on almost all commercial massively-parallel computers, and is being used by the NSF Computational Grand Challenge Cosmology Consortium to support simulations of the evolution of the universe. In this chapter we describe the language and its performance on a variety of problems.

*This research is supported in part by ARPA DABT63-94-C-0029 and Rome Labs Contract AF 30602-92-C-0135. Copyright MIT Press and Indiana University. This material is to appear in *Parallel Programming Using C++*, MIT Press, 1996

## 1 Introduction

The goal of the pC++ project was to design a simple extension to C++ for parallel programming that provides:

- a platform for parallel object-oriented software capable of running without modification on all commercial MIMD systems;
- an interface to Single Program Multiple Data (SPMD) libraries such as ScaLapack++ [6], A++ [18] and POOMA;
- an interface to High Performance Fortran (HPF) [17];
- an interface to control-parallel C++-based languages such as CC++ [5];
- a way to exploit parallel I/O systems and persistent object databases; and
- a complete programming environment including all the tools that users of conventional C++ systems expect, as well as tools for parallel performance analysis and debugging.

We do not think that pC++, or other object-oriented parallel programming languages, should be viewed as replacements for Fortran-90 or HPF. Rather, object-oriented parallelism should be to express those types of parallelism that cannot easily be expressed in these languages. To accomplish this, pC++ exploits the two definining characteristics of object-oriented design: encapsulation and inheritance.

pC++ is based on a *concurrent aggregate* model of data parallelism. This means that a pC++ program consists of a single main thread of control from which parallel operations are applied to *collections* of objects. Each object in a collection is an instance of an *element* class. pC++ has two basic extensions to the C++ language: a mechanism to describe how operations can be invoked over a set of objects in parallel, and a mechanism to refer to individual objects and subsets of objects in a collection.

pC++ has been implemented on a wide range of commercially available parallel systems; we describe its performance on such platforms later in this chapter. Other examples of pC++ programs and performance can be found in [10, 11, 8, 3, 12]. Our primary experience with testing the pC++ ideas on large scale problems has come from our involvement with the NSF Grand Challenge Cosmology Consortium $GC^3$. This chapter describes some of these applications. We also discuss two libraries that support parallel I/O and persistent objects in pC++ programs.

One of the most interesting byproducts of the pC++ project has been a language preprocessor toolkit called Sage++ [2]. This toolkit has been extended in a variety of ways and is used for a large number of applications, including the TAU programming environment.

## 2 History

In 1984 the parallel programming research group at Indiana University, working with the Center for Supercomputing Research and Development (CSRD) at the University of Illinois, developed an extension to the C programming language called Vector Parallel C (VPC) [9]. VPC used parallel loops for spawning new threads of control and a vector notation similar to Fortran-90 for data parallel operations, and assumed a shared memory model of execution.

By 1986 we had become interested in distributed memory multicomputers, and decided to build a new system based on object-oriented design ideas. Our goal was to implement parallel control mechanisms by applying member functions to sets of objects. The first problem to be solved was how to describe a generic set of objects in C++. At the time, the C++ template mechanism was not yet a complete proposal to the C++ standards committee, although early public documents such as [19] guided our thinking.

Even had they existed, templates would not have solved all of our problems. To see why, consider the following definition of a set of objects of type `T` derived from a templatized container class `Set`:

```
Set<T> S;
```

Suppose that the set element type `T` takes the form:

```
class T {
 public:
  void foo();
};
```

Our desire was to be able to apply the member `foo()` to the entire set `S` in parallel with the expression `S.foo()`. Unfortunately, this could not be done using the standard overloading and inheritance mechanisms of C++. Furthermore, because there were no implementations of templates in C++ at that time, we decided to add an extension to pC++ to represent a type of class called a *collection*. Each collection had one built-in "template" parameter called `ElementType`. To simplify the compiler, we put the mechanisms for managing a distributed set of elements into a library called the `SuperKernel` collection. The way in which these collection classes are used is described in detail in the next section.

About the time that our first implementation of pC++ for shared-memory multiprocessors was complete, the HPF Forum was being established. Because HPF was also a data parallel programming language, we were convinced that we needed to base the allocation and data distribution mechanisms for collections on distributed memory systems on the HPF model. Such a design would help make it possible to share distributed data structures with HPF implementations (although this idea has never been tested). In retrospect, we have realized that a standard interface to single-node Fortran-90 is more important support for the HPF model.

In 1992 ARPA provided the support for a complete redesign of pC++ and a public release. The final version of pC++ (version 2.0) will be released in early 1996. This chapter describes this new version of the language.

## 3 Overview of pC++

pC++ was designed to work on both multiprocessors and multicomputers. We use the HPF model to describe the way in which a array-like data structure can be distributed over the memory hierarchy of a parallel computer.

To build a collection of objects from some class type `T`, which is called an *element* class[1] in pC++, one needs a distribution and an alignment object. The distribution object defines

[1]In its current implementation, elements of a collection must be of the same type.

a grid and a mapping from the grid to the physical processors on a parallel machine. The alignment object specifies the shape, size, and the mapping of the element objects to the grid points. In addition, a processor object of type `Processors` is needed to represent the set of processors available to use. For example:

```
Processors P;
Distribution D(100, &P, BLOCK);
Align A(20, "[ALIGN(X[i], D[i+10])]");
```

creates a one-dimensional grid of a size of 100 which is mapped to the processors of the machine by blocks. If there are 20 processors, grid positions 0 through 4 are mapped to processor 0, positions 5 through 9 are mapped to processor 1, etc. The alignment object aligns the logical vector `X[0:19]` with the grid positions `D[10:29]`.

Given a distribution, an alignment and the class type of the element objects, it is easy to build a collection. The starting point is the `SuperKernel` collection provided by the pC++ collection library. This collection is the base type for all other collections. It builds arrays of element objects and provides a global name space for the element objects. Thus, the declaration:

```
SuperKernel<T> MyCollection(&D, &A);
```

creates a collection called `MyCollection`, consisting of a set of 100 objects of type `T` distributed in the manner described above.

The most important feature of a collection is the ability to apply a function in parallel across all the element objects. For example, if `T` is defined as:

```
class T{
  ...
 public:
  void foo();
  int x, y, z;
  float bar(T &);
  ...
};
```

a parallel application of `foo()` to all elements of `MyCollection` would take the form:

```
MyCollection.foo();
```

In the case above, `foo()` has a void result, so the expression `MyCollection.foo()` has a void result as well. However, pC++ extends the type system so that, for example, `MyCollection.x` is an object of type `SuperKernel<Int>`, where `Int` is a library class with one integer value. The expression:

```
MyCollection.x = 2*MyCollection.y + MyCollection.z;
```

is therefore a parallel computation involving pointwise multiplication, addition and assignment on the members each element of the collection.

Similarly, if `t` is of type `T` the expression `MyCollection.bar(t)` applies `bar(t)` to each element of the collection. The result is of type `SuperKernel<Float>`. Also, if `C` is another collection whose size is the same as `MyCollection` and whose element type is `T`, the expression

```
MyCollection.bar(C)
```

will apply `bar()` to the $i^{\text{th}}$ element of `MyCollection` using the $i^{\text{th}}$ element of `C` as an argument.

It is often the case that an operation must be applied to a subset of the elements of a collection. pC++ extends the Fortran 90 vector notation so that descriptors of the form `base:end:stride` can be used to select elements from a collection. For example:

```
MyCollection[0:50:2].foo()
```

will apply `foo()` to the first 25 even numbered elements of the collection.

To access an individual member of a collection, one can use the overloaded operator `()` which returns a *global* pointer to an element, i.e. a pointer that can span the entire address space of a distributed-memory machine. For example:

```
MyCollection(i)
```

returns a global pointer to the $i^{\text{th}}$ element in the collection. In this way, any object can have a global address. The function call:

```
MyCollection(i)->foo();
```

is a remote invocation. It sends a message to the processor that contains the $i^{\text{th}}$ element of `MyCollection`, and a thread on that processor executes the function[2].

Programmers often need to create specialized collections with properties appropriate for their particularl applications. The task of building a new derived collection is almost the same as building a derived class in C++. The definition of a collection derived from `SuperKernel` takes the form:

```
Collection MyCollectionType: SuperKernel {
 public:
  // Public data members duplicated on each processor.
  // Public member functions executed in parallel on all processors.
 MethodOfElement:
  // Data members and member functions here are added to
  // the element class.
};
```

There are two types of data and member functions in a collection definition. Data and functions labeled as `MethodOfElement` represent new data members and functions that are to be added to each element class. Such member functions are invoked just as if they were member functions of the element class. Data members not labelled as `MethodOfElement` are defined once on each processor; functions not labeled `MethodOfElement` are invoked in SPMD mode.

[2]Remote invocation of this kind is part of pC++ 2.0, and is not part supported by the current pC++ 1.0 distribution.

## 3.1 pC++ Run-Time System

pC++ is extremely portable. It currently runs on the Cray T3D, IBM SP2, Intel Paragon, Meiko CS-2, SGI Power Challenge, TMC CM5, Convex Exemplar, and networks of workstations. The key to this portability is the simple execution model and layered run-time system. The first run-time layer is machine independent, and defined by the pC++ compiler (source to source translator). The compiler generates calls to the C++ class library, `kernel.h`. There are two versions of this class library layer, one for SPMD execution and one for fork/join thread-based execution.

For distributed memory machines, SPMD execution is used, and the pC++ compiler converts parallel invocations such as

```
MyCollection.foo();
```

into loops over the local collection elements (owner computes).

```
for (i= FirstLocal(); i >= 0; i = NextLocal(i))
  MyCollection_T(i)->foo();
pcxx_Barrier();
```

The data type `MyCollection_T` is generated by the compiler for the declaration

```
SuperKernel<T> MyCollection();
```

The generated loop uses the overloaded `()` operator, provided by the pC++ class library, to find the $i^{\text{th}}$ collection element. After each processor has applied `foo()` to its local elements, a barrier synchronization between processors in initiated.

Shared memory machines can use the SPMD model shown above, or the pC++ compiler can generate a special thread-based run-time interface. A fork/join class library thread interface replaces the explicit `for(;;)` loop with a call to simply execute `thread_Fork(...)`. `Thread_Fork()` executes `foo()` on a set of threads. The abstraction of work is sufficiently general to permit many different thread packages. An implementation could create a new thread for each iteration or use a set of persistent threads.

## 3.2 Tulip

The next portion of the run-time system is the low-level machine-dependent layer, called "tulip". Tulip describes an abstract machine, and defines standardized interfaces for basic machine services such as clocks, timers, remote service requests, and data movement. Tulip has a C interface, and has no knowledge of pC++ or the class library, which are built atop Tulip. Therefore, wherever Tulip can be ported, pC++ can run.

Tulip has several basic abstractions:

*Context*: An address space. A Unix process on a symmetric multiprocessor would be a single context. Lightweight threads share a context. A machine such as the SP2 can support several contexts per node.

*LocalPointer*: A simple, untyped, memory address. A LocalPointer is valid only within the Context it was created.

*GlobalPointer*: The tuple (Context, LocalPointer). A GlobalPointer uniquely identifies any memory address in the computational hardware.

Those abstractions are used in the following basic functions:

```
tulip_Put(tulip_GlobalPointer_t destination, char *source,
          int length, tulip_ACK_t *handle);

tulip_Get(char *destination, tulip_GlobalPointer_t source,
          int length, tulip_ACK_t *handle);

tulip_RemoteServiceRequest(int context, char *buffer,
                           int length, tulip_ACK_t *handle);
```

`Put()` and `Get()` simply move data between contexts. They are very similar to `memcpy()`, except destination and source are global pointers respectively. Furthermore, an acknowledge handle is provided so the status of the data transfer can be monitored. If the handle is `NULL` when the function is called, no acknowledgment is done. The functions are non-blocking, so that they can be easily integrated with user-level thread packages.

The remove service request mechanism provides asynchronous communication between contexts. It is particularly useful for bootstrapping, building remote procedure execution for pC++ (see section Section 3), and transmitting short control messages to other contexts.

The basic abstractions and functions are supported on three architecture models: shared memory, message passing, and network DMA.

The SGI Challenge and Convex Exemplar are examples of a shared memory machines. The hardware maintains cache and memory consistency, and communication is done by simply sharing pointers. In this case, `Put()` and `Get()` need not be used, because those functions move data between contexts. On a shared memory machine, there is usually only one context. However, if `Put()` or `Get()` were used, they would simply be a call to `memcpy()` followed by setting the acknowledge handle to `TRUE`.

Two examples of message passing machines are the Intel Paragon and IBM SP2. Since `Put()` and `Get()` are one-sided communication primitives, and do not require synchronization, either active messages or polling loops must be used to detect when a data movement request arrives. For `Get()`, a `recv()` is posted for the anticipated data, then a data request message is sent to the remote context (node). When the sender detects the data request message during a message poll, the data is sent to the awaiting `recv` without a buffer copy. `Put()` uses a similar mechanism, but requires an extra round trip to avoid any buffer copies. If the message is sent to the remote context "eagerly", the extra round trip latency is not incurred, but the messaging system must copy and buffer the data.

The Meiko CS-2 and Cray T3D are network DMA machines. They are not truly shared memory, since transfers to "remote" memory must be done through special system calls. On the other hand, there is no synchronization with the remote node when data is moved with either a load or a store. `Get()` and `Put()` can be written as calls to the underlying vendor-supplied transport functions.

For all machines, a polling loop or interrupt must be used to detect a remote service request. Currently, Tulip uses a polling loop to detect requests. However, as active message layers for machines such as the SP2 become available, Tulip will be rewritten to take advantage of fast handlers and eliminate the need for polling.

## 3.3 I/O

pC++/streams is a library which supports a simple set of high level I/O primitives on pC++ collections. To illustrate its capabilities, we describe how pC++/streams can be used to checkpoint a collection having variable-sized elements.

Assume our application simulates the behavior of particles in three-dimensional space. We can model the particles with a one-dimensional distributed array of variable-length particle lists, each of which keeps track of the particles in a slab of space:

```
class Position {
  double x, y, z;
};

class ParticleList {
  int numberOfParticles;
  double * mass;          // variable sized
  Position * position;    // arrays
 };

Collection DistributedArray {
  updateParticles();
};

Processors P;
Align a(12,"[ALIGN(collection[i], template[i])]");
Distribution d(12, &P, CYCLIC);

DistributedArray<ParticleList> particleArray(&d,&a);
```

The programmer can write a function to checkpoint the `particleArray` collection as follows:

```
#include "pc++streams.h"
void saveParticleArray() {
  oStream stream(&d, &a, "myFileOne");
  stream << particleArray;
  stream.write();
}
```

The first line of `saveParticleArray()` defines an output pC++/stream called `stream`, connected to the file `myFileOne`. The second line inserts the entire `particleArray` collection into the buffers of the stream. The third line causes those buffers to be written to the file, using parallel I/O. The file associated with the stream is closed automatically when the program block in which the stream was declared is exited.

The programmer would write a function to restore the checkpointed `particleArray` as follows:

```
void loadParticleArray() {
  iStream stream(&d, &a, "myFileOne");
  stream.read();
  stream >> particleArray;
}
```

pC++/streams also allows selective I/O on individual fields of collection elements:

```
stream << particleArray.numberOfParticles;
```

pC++/streams supports I/O on collections with complex elements (e.g. variable-sized elements, tree-structured elements, etc) by giving the programmer a straightforward mechanism for defining how these data structures are to be read and written: insertion and extraction functions. In our example the programmer would define an insertion function for `ParticleLists` as follows:

```
declareStreamInserter(ParticleList &p) {
  eltBuf << p.numberOfParticles;
  eltBuf << array(p.mass, p.numberOfParticles);
  eltBuf << array(p.position, p.numberOfParticles);
}
```

The `array()` macro tells pC++/streams that `mass` and `position` are dynamically-allocated arrays of size `numberOfParticles`. Extraction functions are defined similarly. pC++/streams is described in more detail in [14].

## 3.4 Persistence

pC++/persistence is an I/O library supporting persistence for pC++ collections. This library is currently implemented using the SHORE persistent object system from the University of Wisconsin, Madison [4].

Normally, elements of pC++ collections are transitory, i.e., their data disappears when the program terminates. In order to preserve transitory data, the programmer must output that data to a file before the program terminates, using either an I/O mechanism supported by the operating system or a higher-level library such as pC++/streams.

pC++/persistence allows programmers to define persistent collections, whose elements can contain persistent data in addition to ordinary transitory data. The persistent section of each element is automatically preserved across program executions; no application I/O code is required to save or load this data. A transaction mechanism is supported, allowing programmers to checkpoint persistent data with a single line of code that commits a transaction. In addition, the persistent part of a collection is concurrently accessible by multiple pC++ programs, with no explicit code for communication required. Concurrent access to persistent data can allow simplified programming of concurrent computation and visualization, computational steering, and modular multi-disciplinary simulations, since no application code needs to be devoted to I/O or communication of the persistent data.

As an example, we first we define the per-element persistent data using SDL (SHORE Data Language). For simplicity, our persistent data will consist of just a single long integer per element, called `myPersistentLong`:

```
module MyElement {
  interface PersistentElementData {
    public:
      attribute long myPersistentLong;
  };
}
```

This SDL specification is processed by the SHORE SDL type compiler, informing SHORE of the structure of the persistent part of our elements.

We next define an element class `MyElement` in ordinary pC++. We derive it from the class `PersistentElement`, and define an ordinary transient data member (`myTransientLong`) in the usual way:

```
#include "PersistentElement.h"

class MyElement : public PersistentElement{
 public:
  long myTransientLong;
  void P_initialize();
  void hello();
};
```

The class `PersistentElement` contains a member `P` through which the persistent part of each element is accessed:

```
void MyElement::hello() {
  printf(" Hello world: %ld %ld",
         myTransientLong, P->myPersistentLong);
}
```

The function `P_initialize()`, defined within `MyElement`, gives the application programmer a mechanism for initializing the persistent part of each element. `P_initialize()` is called immediately after the persistent part of each element is first created.

```
void MyElement::P_initialize() {
  P.update()->myPersistentLong = 1234;
}
```

The call to `P.update()` above informs pC++/persistence that the persistent part of the element is to be modified, rather than just accessed.

A persistent collection is defined just like an ordinary collection, except that it is derived from `PersistentCollection`:

```
#include "PersistentCollection.h"

Collection MyCollection: public PersistentCollection {
 public:
  MyCollection(Distribution *T, Align *A,
               char *persistentCollectionName);
 MethodOfElement:
  virtual void hello();
};

MyCollection::MyCollection(Distribution *T, Align *A,
                           char *persistentCollectionName)
 : PersistentCollection(T, A, persistentCollectionName) {}
```

When the programmer instantiates the collection `X` below, the string `myPersistentCollectionName` is passed into the collection constructor, and then to `PersistentCollection`. This string identifies a particular database of persistent elements to be associated with the collection.

```
void Processor_Main(int argc, char **argv){
  Processors P;
  Distribution T(SIZE, &P, BLOCK);
  Align A(SIZE,"[ALIGN(V[i], T[i])]");
  MyCollection<MyElement> X(&T, &A, "/myPersistentCollectionName");

  beginTransaction();
  X.hello();
  commitTransaction();
}
```

Changes to the persistent part of a collection must be made within a transaction, initiated by `beginTransaction()`. These changes do not become permanent and are not visible to other applications until the transaction is committed with a call to `commitTransaction()`. So to checkpoint the persistent part of a collection, all that is required is a call to `commitTransaction()`.

pC++/persistence is still under development at the time of the writing of this text; some details may change and some functionality may be added before the implementation is complete.

## 4 An Example: Parallel Sorting

To see how pC++ is used, consider the problem of sorting a large vector of data using a parallel bitonic sort algorithm. A bitonic sequence consists of two monotonic sequences that have been concatenated together where a wrap-around of one sequence is allowed. That is, it is a sequence:

$$a_0, a_1, a_2, \ldots, a_m$$

where $m = 2^n - 1$ for some $n$, and for index positions $i$ and $j$, with $i < j$, $a_i, a_{i+1}, \ldots, a_j$ is monotonic and the remaining sequence starting at $a_{(j+1) \bmod n}$, where $a_0$ follows $a_n$, is monotonic in the reverse direction.

Merging a bitonic sequence of length $k$ involves a sequence of data exchanges between elements that are $k/2$ apart, followed by data exchanges between elements that are $k/4$ apart, etc. The full sort is nothing more than a sequence of bitonic merges. We start by observing that a set of two items is always bitonic. Hence for each even $i$, the subsequence $a_i$ and $a_{i+1}$ is always bitonic. If we merge these length two bitonic sequences into sorted sequences of length two and if we alternate the sort direction, we then have bitonic sequences of length

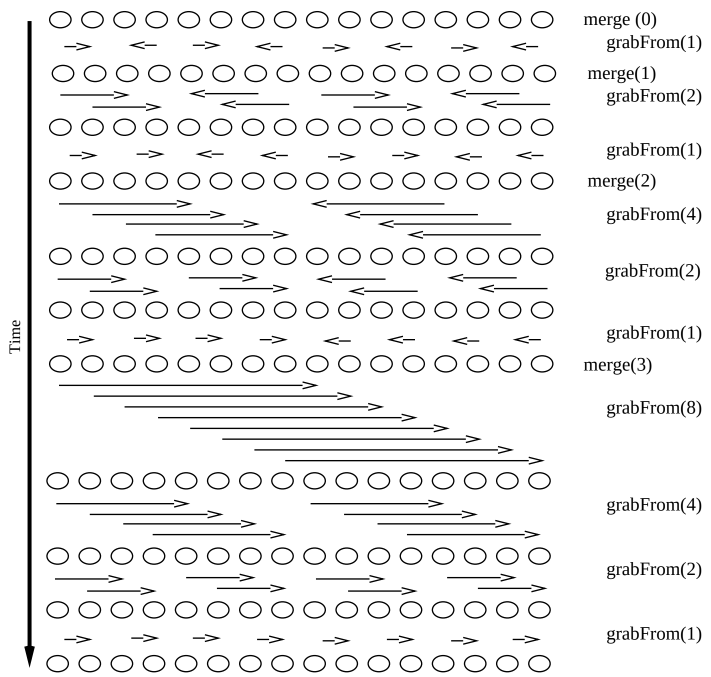

Figure 1: Data exchanges in the Bitonic Sort Algorithm

four. Merging two of these bitonic sequences (of alternating direction) of length 4 we have sorted sequences of length 8. The sequence of data exchanges is illustrated in Figure 1.

In pC++, a pure data-parallel version of this algorithm can be built from a collection `List` of objects of type `Item` as shown below. Each item contains an object of type `E` which is assumed to be the base type of the list we want to sort:

```
struct E {
 public:
  int key;
};
class Item {
 public:
  E a;
};
```

The `List` collection contains one public function `sort()` and a number of fields and members that are defined in the `MethodOfElement` section. Because the parallel algorithms require parallel data exchanges, we must have a temporary `tmp` to hold a copy of the data to be exchanged for each element. In addition, there are two flags, `exchangeDirection` and `sortDirection` which are used to store the current exchange direction and the current sort order respectively. As can be seen in the figure above, the value of these flags depends on the location of the element in the list as well as the point in time when an exchange is made.

```
Collection List: SuperKernel {
 public:
  void sort();
  int N;  // number of elements
 MethodOfElement:
  E tmp;
  virtual E a;
  int sortDirection, exchangeDirection;
  void set_sort_direction (int k) { sortDirection  = (index1/k)%2; }
  void set_exchange_direction(int k) { exchangeDirection = (index1/k)%2; }
  void merge(){
    if (((sortDirection == exchangeDirection) && (this->a.key > tmp.key)) ||
        ((sortDirection != exchangeDirection) && (this->a.key <= tmp.key))){
      this->a = tmp;
    }
  }
  void grabFrom(j){
    if(exchangeDirection == 1) tmp = (*thisCollection)(index1+j)->a;
    else        tmp = (*thisCollection)(index1-j)->a;
  }
};
```

In general `MethodOfElement` functions are those element-wise operations in an algorithm that depend on the relation of one element to the whole collection or to other elements in the collection. For example, the function `grabFrom(int j)` is a method that when applied to one element will access the data in another element `j` positions to the right or left of the current position. The `SuperKernel` class provides two additional members, `thisCollection` and `index1` which provide a pointer to the containing collection and the position of the element in the collection, respectively. The function `merge()` uses the current state variables `sortDirection` and `exchangeDirection` to determine which element of the data to keep after the exchange step.

The `sort()` function is then a sequence of merge steps, each of which contains a sequence of exchanges as shown below. The main program allocates a list of items and then calls the sort function.

```
List::sort(){
  int k = 1;
  for (int i = 1; i < log2(N); i++){  // merge(i) step
    k = 2*k;
    this->set_sort_direction(k);
    for (int j = k/2; j > 0; j = j/2){ // exchange(j) step
      this->set_exchange_direction(j);
      this->grabFrom(j);
      this->merge();
    }
  }
}

Processor_main(){
  Processors P;
  int N = read_problem_size();
  Distribution D(N,&P,BLOCK);
  Align A(N,"[ALIGN(X[i],D[i])]");
  List< Item >  L(&D, &A);
  ...
  L.sort();
}
```

This version of the program works, but has a serious flaw. If the size of the list to be sorted is $N$ and there are only $P << N$ processors in the system, the bitonic sort has parallel complexity $O(\frac{N}{P}\log^2 N$, which is far from optimal. To improve the efficiency, we can build a hybrid algorithm as follows. Let us break the list of $N$ into $P$ segments of length $K = \frac{N}{P}$. We begin the sort by applying a quicksort to each segment, but sorting them in alternating directions. Now each pair of adjacent sorted segments forms a bitonic sequence and we can apply the bitonic merge as before. However, at the end of each merge step, the list in each segment is only a bitonic sequence, not a sorted sequence. We must then apply a local bitonic merge to sort it. If we rewrite the algorithm above with a `Segment` class replacing the `Item` class and expanding the `tmp` variable to an array we only need to make a few modifications to the program. These changes, shown below, consist of inserting the calls to the local quicksort and local bitonic merge in the sort function. The `grabFrom` and `merge` functions also need to be replaced by ones that can accommodate an array.

```
// P is the number of elements (processors)
// N is the total number of elements to sort.
// K = N/P is the size of each segment.

class Segment{
 public:
  E a[K]
  quickSort(); // O(K log(K))
  localBitonicMerge(int direction); // O(K)
};

List::sort(){
  int k = 1;
  this->quickSort();
  for (int i = 1; i < log2(P); i++){  // merge(i) step
    k = 2*k;
    this->set_sort_direction(k);
    for (int j = k/2; j > 0; j = j/2){ // exchange(j) step
      this->set_exchange_direction(j);
      this->grabFrom(j);
      this->merge();
    }
    this->localBitonicMerge(d);
  }
}
```

```
void SortedList::grabFrom(int dist){
  E *T;
  int offset = (d2)? -dist: dist;
  T = &((*ThisCollection)(index1+offset)->a[0]);
  for(int i = 0; i < K; i++) tmp[i] = T[i];
}
void SortedList::merge(){
  for (i = 0; i < K; i++)
    if(((d == d2) && (a[i].key > tmp[i].key)) ||
       ((d != d2) && (a[i].key <=tmp[i].key))){
      a[i] = tmp[i];
    }
  }
};
```

Assume that the quicksort computation runs with an average execution time of $DK \log K$ for some constant $D$, that we can ignore the cost of the barrier synchronization, that there are $P = 2^n$ processors available, and that the size of the list to be sorted is $N$. The time to sort is then roughly:

$$T(N) = \frac{N}{P} C \log^2 P + D \frac{N}{P} \log \frac{N}{P} + \log P$$

where $C$ is a constant that depends upon communication speed. Given a sequential complexity of $DN \log N$ we see that the parallel speed-up is of the form:

$$\mathit{Speedup}(N, P) = \frac{P}{1 + \frac{C \log^2 P}{D \log N}}$$

which, for large $N$, approaches $P$.

This algorithm has been tested on a variety of machines and it is both portable and fast. Sorting one million items takes 3.56 seconds on a 64 node Paragon and 1.68 seconds on an 8 node SGI Challenge. However, comparing this to the standard system routine `qsort` reveals that the speedup is not great. On the same data set with one node of the SGI Challenge, `qsort` requires 10.21 seconds. Hence the speed-up of our algorithm is 6.08 on 8 processors. This value matches the formula above when $C = D$.

## 5 The Polygon Overlay Program

The following algorithm is used to implement the polygon overlay code in pC++. Given two maps $A$ and $B$ as input, map $A$ is divided into smaller maps $A_s$. These smaller maps are then distributed over the elements of a pC++ collection. If there are $N$ polygons in map $A$ to be divided and $P$ collection elements, then each element gets $N/P$ polygons, except element zero, which gets $N/P + N \bmod P$ polygons (Figure 2).

Map B is duplicated in each element. During a parallel computation, each element finds the overlay of map $A_s$ and map $B$. In the output stage, the resulting overlay map in each element is combined with the maps in the other elements to form the final overlay map.

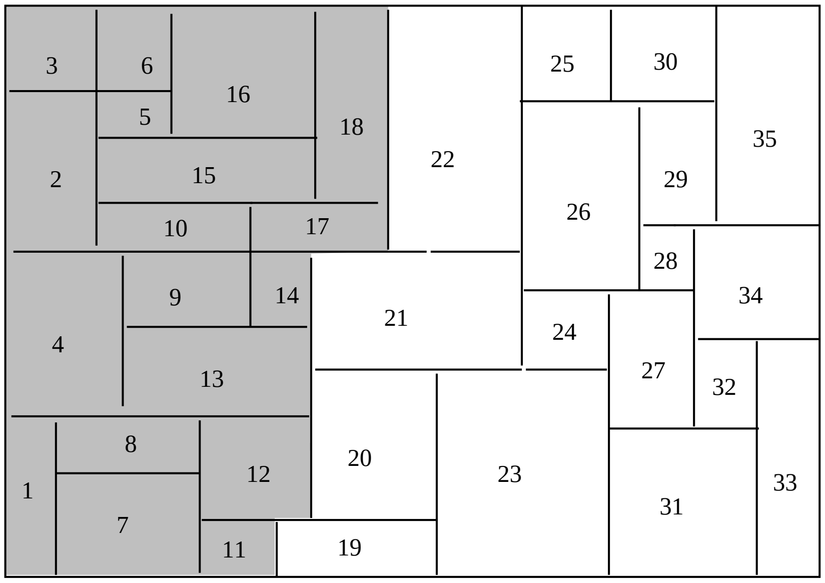

Figure 2: Polygon map distribution scheme. This shows the distribution of a map consisting of $N = 35$ polygons over $P = 2$ elements. The polygons are numbered (sorted) according to the order used in the sequential ANSI C implementation. With $N/P = 17$ and $N \bmod P = 1$, element 0 gets the shaded polygons and element 1 gets the unshaded polygons. In our tests reported in this paper, the load imbalance was generally insignificant.

No inter-element communication is required during the parallel computation and thus the computation is carried out in the "embarrassingly parallel" fashion. In this algorithm, map $B$ is not divided and distributed; if it were, it would be difficult for an $A_s$ map to know whether it overlaps with a $B_s$ map which is in another element. A more elaborate parallel algorithm would be required, and inter-element communication would be unavoidable. We discuss this further later in this section.

The pC++ element class is defined as follows:

```
class Patch {
 public:
  polyVec_p leftVec, rightVec, outVec;
  Patch() {}
};
```

where `leftVec`, `rightVec`, and `outVec` are, respectively, pointers to map $A_s$, map $B$, and their resulting overlay. The pC++ collection is defined as follows:

```
Collection Overlay : public SuperKernel {
 public:
  Overlay(Distribution *D, Align *A);
 MethodOfElement:
  virtual polyVec_p leftVec, rightVec, outVec;
  void readMap();
  void writeMap();
  void distributeMap();
  void findOverlay();
};
```

In this definition, readMap inputs the two polygon maps. Since it is a MethodOfElement, the actual reading is carried out by element zero. After the two maps are read, element zero calculates the number of polygons all other elements should have and broadcasts the information. In distributeMap, all other processors then fetch their piece of the first polygon map and the entire second polygon map from element zero. findOverlay finds the overlay of $A_s$ and $B$ maps. In its pC++ implementation, findOverlay simply calls the original ANSI C polygon overlay functions based on user-selected options. No modification of the ANSI C code is needed, except in the case of a modified list-deletion algorithm described in later in this section.

In writeMap, element zero gathers overlay maps from all the elements. It calls a sorting routine to sort the polygons in a special order and writes the entire overlay map out. The sorting can actually be done in parallel (Section 7.1), but since our focus was the parallelization of the polygon overlay algorithm itself, we did not parallelize the sorting routine. The actual implementation of function findOverlay is given in the following piece of code.

```
void Overlay::findOverlay()
{
  double time;
  pcxx_UserTimerClear(index1);
  pcxx_UserTimerStart(index1);
  if (useLnArea && useOrder){
    /* sorted-ordered list-deletion overlay */
    outVec = overlayAreaLinkedOrder(leftVec, rightVec);
  } else ···{
    ···as in sequential code···
  }
  pcxx_UserTimerStop(index1);
  time = pcxx_UserTimerElapsed(index1);
  printf("Time for element %d : %lf", index1, time);
}
```

where pcxx_UserTimer functions clear, start, and stop a timer numbered by the element's index. pcxx_UserTimerElapsed reports the elapsed time. The main program is:

```
void Processor_Main() {
  int elem_count = pcxx_TotalNodes();
  Processors P;
  Distribution D(elem_count,&P,BLOCK);
  Align A(elem_count,"[ALIGN(X[i],D[i])]");
  Overlay<Patch> X(&D,&A);
  X.readMap();
  X.distributeMap();
  X.findOverlay();
  X.writeMap();
}
```

| Platform | Number of Processors | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| Cray T3D | 2135.7 | 1143.2 | 590.2 | 299.8 | 151.1 | 75.8 | |
| | 19.9 | 990.5 | 735.7 | 429.6 | 229.6 | 118.2 | |
| | 19.5 | 10.6 | 5.7 | 3.2 | 2.1 | 1.5 | |
| Intel Paragon | 3782.4 | 1942.4 | 983.4 | 494.9 | 248.1 | 124.2 | |
| | 28.5 | 1414.8 | 1048.7 | 612.7 | 327.6 | 168.8 | 85.6 |
| | 29.1 | 14.5 | 7.6 | 4.3 | 2.9 | 2.0 | 1.66 |
| IBM SP-2 | 1587.7 | 812.1 | 410.4 | 205.7 | 103.3 | | |
| | 10.2 | 554.4 | 430.1 | 238.8 | 127.6 | | |
| | 10.6 | 5.4 | 2.9 | 1.7 | 1.0 | | |
| Power Challenge | 1409.8 | 724.8 | 367.6 | 185.1 | | | |
| | 11.7 | 547.1 | 405.4 | 236.2 | | | |
| | 11.7 | 6.6 | 4.0 | 2.5 | | | |
| SPARC 10 | 1562.3 | | | | | | |
| | 14.0 | | | | | | |
| | 13.4 | | | | | | |

Table 1: Time in seconds spent in the `findOverlay` function. Two maps each containing about 60,000 polygons (file K100.00 and K100.01) were used as input. For each platform, results are shown for the naive overlay algorithm first, then for the list-deletion overlay algorithm, then for the modified list-deletion overlay algorithm.

where `pcxx_TotalNodes` returns the number of processors used for the computation.

The pC++ code was tested on a variety of platforms including a Cray T3D, an IBM SP-2, a SGI Power Challenge, an Intel Paragon and a Sun Sparc 10. Two maps each containing about 60,000 polygons were used as input. Three sets of tests were conducted using the naive overlay algorithm, the list-deletion overlay algorithm, and a modified list-deletion overlay algorithm.

The reason that the modified list-deletion overlay algorithm was used was that after testing the list-deletion algorithm on parallel machines we found it very inefficient for parallel execution. The list-deletion overlay algorithm is best suited when two maps cover roughly the same area. Otherwise, extra work is needed for comparing polygons in map $A$ with polygons in map $B$ in the area where no overlay occurs (see Figure 3). It turned out the extra work was so significant that the performance of the algorithm virtually degraded below that of the naive algorithm. Because the polygons were already sorted in all the maps we used for our tests, we modified the list-deletion overlay algorithm so that it could remember where in map $B$ the overlay occurred. Comparison involving a subsequent polygon in map $A$ would start from this location instead of from the beginning of $B$ (see Figure 3). The benchmark results of the three sets of experiments are shown in Table 1 and Figure 4.

As can be seen in Table 1 and Figure 4, on all the machines we were able to obtain nearly linear speedups for the naive and the modified list-deletion algorithm. The speedup curves decreased slightly for the modified list-deletion algorithm as the number of processors

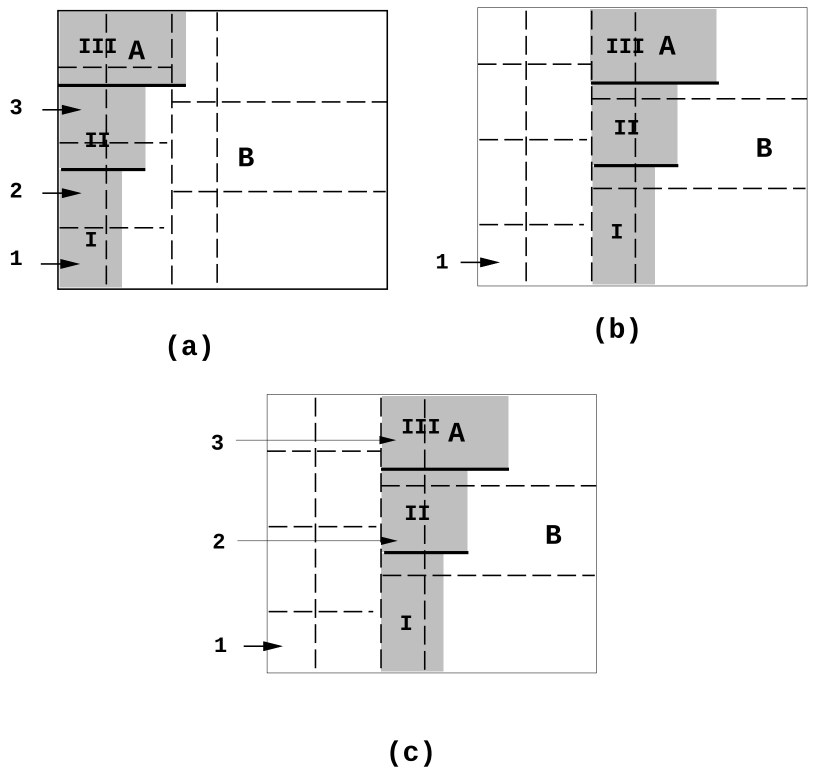

Figure 3: Comparing the list-deletion with the modified list-deletion algorithm. We are given two maps $A$ and $B$. $A$ is indicated by shaded area. Polygons in map $A$ are separated by solid lines. Polygons in $B$ are separated dashed lines. Assume the polygons are sorted according to the $x$ coordinates of their upper right corner, the ordering scheme used in the ANSI C code, so that comparison of the two maps would begin with the polygons in the lower left corners of both maps. (a) In both algorithms, when the lower left corners of maps $A$ and $B$ coincide, the loop which compares polygons in $B$ with polygon $I$ in $A$ begins with the polygon pointed by arrow 1. The subsequent comparisons of polygons in $B$ with polygon $II$ in $A$ begin with the polygon pointed arrow 2, because the polygon pointed by arrow 1 has been eliminated in earlier comparisons. Similarly, subsequent comparisons of polygons in $B$ with polygon $III$ in $A$ begin with the polygon pointed arrow 3. (b) In list-deletion algorithm, when the lower left corners of the maps do not coincide, all comparisons of polygons in $B$ with polygons $I$, $II$, $III$ in $A$ begin with the polygon pointed by arrow 1. This is because polygons to the left of map $A$ are never eliminated in the comparison process. (c) In the modified list-deletion algorithm, when the lower left corners of the maps do not coincide, only comparisons involving polygon $I$ begin with the polygon pointed by arrow 1. Subsequent comparisons involving $II$ and $III$ begin with the polygons pointed by arrow 2 and 3 respectively.

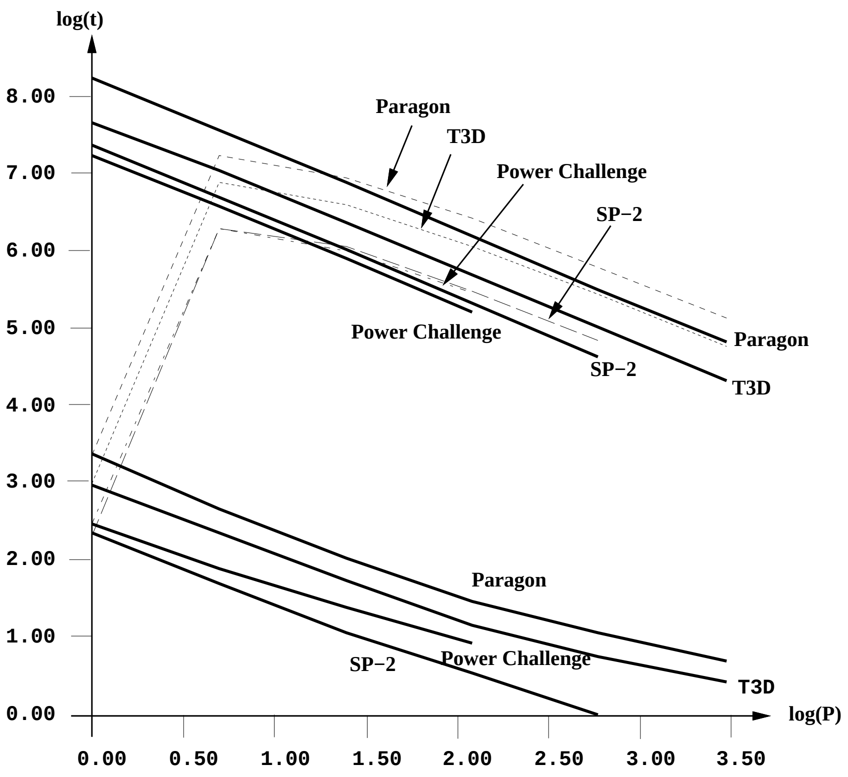

Figure 4: log($P$) vs. log($t$) plot. $P$ is the number of processors and $t$ is the execution time in seconds. The upper four solid lines are log($P$) - log($t$) curves for the naive overlay algorithm; the lower four solid lines are log($P$) - log($t$) curves for the modified list-deletion algorithm; the four dashed lines are the log($P$) - log($t$) curves for the list-deletion algorithm.

increased. This was due to the fact that as workload on each processor decreased, the overhead became more prominent. The results show that the parallel algorithm we adopted worked very well for the naive and the modified list-deletion algorithm. The original list-deletion algorithm is not well-suited for parallel execution, causing the parallelized code to perform poorly.

One way to parallelize the list-deletion algorithm without modifying the sequential list-deletion algorithm is to divide $B$ into $B_s$ and distribute $B_s$ as we did with $A_s$. Assuming after the division and distribution $A_s$ and $B_s$ roughly cover the same area, finding the overlay of them will be straight forward. Once the overlay of $A_s$ and $B_s$ is found, the collection elements exchange the part of $B_s$ where no overly is found, and a second phase of parallel operation can be carried out. This parallel algorithm requires $N$ phases of parallel operation where $N$ is the number of collection elements (usually chosen to be equal to the number of processors). The algorithm also requires that the input polygons be sorted.

On the other hand, if we can assume the input polygons are sorted, we can modify the list-deletion deletion algorithm slightly to completely eliminate the cost of communication and achieve a much better result. This was the reason that the modified list-deletion algorithm was used in our tests. However, it should be noted that the result of the algorithm is a distributed list of polygons which are locally sorted but not globally sorted. But globally sorting the polygons is a very simple task. The sorting algorithm described in the previous section has been applied to a data set of this size and the time to sort it was 0.35 seconds on an 8 processor SGI power challenge. Hence the execution times in the table above should have about one third of a second added to account for the final sort.

A large fraction of the code in many parallel applications is devoted to I/O. For example, in an early version of the polygon overlay program using ordinary UNIX file I/O, 200 lines of code (approximately 10% of the total), was devoted to I/O. In addition to programming time overhead for file I/O, there is run time overhead as well; I/O is increasingly being identified as a bottleneck in parallel applications.

pC++/streams (Section 3.3) can reduce the programming time and run time overheads associated with file I/O in pC++ applications. Rewriting the original UNIX I/O in the polygon overlay program using pf-streams reduced I/O code from 200 lines to 70 lines.

## 6 The Self-Consistent Field Code

Here and in Section 7 we describe our work with with the Grand Challenge Cosmology Consortium (GC$^3$). This work is abstracted from two longer papers [20] and [13].

One of the N-body codes developed by the GC$^3$ researchers is the Self-Consistent Field (SCF) code, which is used to simulate the evolution of galaxies. It solves the coupled Vlasov and Poisson equation for collisionless stellar systems using the $N$-body approximation approach. To solve Poisson's equation for gravitational potential:

$$\nabla^2\Phi(\vec{r}) = 4\pi\rho(\vec{r}),$$

the density $\rho$ and potential $\Phi$ are expanded in a set of basis functions. The basis set is constructed so that the lowest order members well-approximate a galaxy obeying the de

Vaucouleurs $R^{1/4}$ projected density profile law. The algorithm used is described in detail in [15].

The original SCF code was written in Fortran-77 by Lars Hernquist in 1991. In 1993, the code was converted to Thinking Machines CM Fortran by Greg Bryan. Expirements conducted by Bryan on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) indicate that with 10 million particles the CMF code can achieve 14.4 Gflops on 512 nodes of the CM-5 [15].

The expansions of the density and potential take the following forms:

$$\rho(\vec{r}) = \sum_{nlm} A_{nlm}\rho_{nlm}(\vec{r})$$

$$\Phi(\vec{r}) = \sum_{nlm} A_{nlm}\Phi_{nlm}(\vec{r})$$

where $n$ is the radial quantum number and $l$ and $m$ are quantum numbers for the angular variables. Generally, the two sums will involve different expansion coefficients. But the assumption of bi-orthogonality ensures a one-to-one relationship between terms in the expansions for the density and potential. The basis sets $\rho_{nlm}$ and $\Phi_{nlm}$ also satisfy Poisson's equation:

$$\nabla^2\Phi_{nlm}(\vec{r}) = 4\pi\rho_{nlm}(\vec{r})$$

and are given by:

$$\rho_{nlm}(\vec{r}) = \frac{K_{nl}}{2\pi}\frac{r^l}{r(1+r)^{2l+3}}C_n^{2l+3/2}(\xi)\sqrt{4\pi}Y_{lm}(\theta,\phi)$$

$$\Phi_{nlm}(\vec{r}) = -\frac{r^l}{(1+r)^{2l+1}}C_n^{2l+3/2}(\xi)\sqrt{4\pi}Y_{lm}(\theta,\phi)$$

$$\xi = \frac{r-1}{r+1}$$

where $K_{nl}$ is a number related only to $n$ and $l$, and $C_n^{2l+3/2}(\xi)$ and $Y_{lm}(\theta,\phi)$ are ultraspherical polynomials and spherical harmonics, respectively. After some algebra, the expansion coefficients become

$$A_{nlm} = \frac{1}{I_{nl}}\sum_k m_k[\Phi_{nlm}(r_k,\theta_k,\phi_k)]^*$$

where $I_{nl}$ is a number and $m_k$ is the mass of the $k$th particle. Once the gravitational potential is found, the gravitational force per unit mass can be obtained by taking the gradient of the potential and the particles can be accelerated accordingly.

## 6.1 The pC++ Version of the SCF Code

We design a C++ class called `Segment` to represent a subgroup of the N particles used in the simulation. As we have discussed earlier, the major procedure in the SCF code is to compute the sums for the expansion coefficients $A_{nlm}$. Our approach is to first compute local sums within each `Segment` object. After this, global sums are formed by a global reduction. The global sums are then broadcast back to each `Segment` object where the particles are accelerated by the gravitational force. Now Fortran subroutines in the original Fortran code can be used as member functions of the `Segment` class, although subroutines involving inter-element communication and I/O need to be modified. The Fortran subroutines are called by pC++ through a specially designed Fortran interface [20]. The `Segment` class is declared (with many unimportant variables and member functions omitted) as follows:

```
class Segment {
 public:
  FArrayDouble x, y, z, vx, vy, vz, ax, ay, az, mass,
               plm, clm, dlm, elm, flm, dplm;
  double sinsum[lmax+1][lmax+1][nmax+1],
         cossum[lmax+1][lmax+1][nmax+1];
  Segment();
  void compute_polynomial();
  void compute_acceleration();
  void update_position();
  void update_velocity();
};
```

The data type `FArrayDouble` is defined in the Fortran library; it serves as an interface to Fortran double precision arrays. The `FArrayDobule` variables defined above are one-dimensional arrays that contain the positions, the velocities, the accelerations, and the masses of particles belonging to a `Segment` object, and the expansion coefficients and values of the polynomial. `sinsum` and `cossum` contain the local sums and eventually the global sums of the expansion coefficients. The class member functions call Fortran subroutines: `compute_polynomial` computes the polynomials and local sums, `compute_acceleration` computes the acceleration for each particle, and `update_position` and `update_velocity` update the positions and velocities of particles.

The collection that distributes the elements, allocates memory, and manages inter-element communication is declared as below. Again, many less important member functions are omitted for brevity:

```
Collection SelfConsistField : public Fortran {
 public:
  SelfConsistField(Distribution *D, Align *A);
  void InParticles();
  void InParameters();
  void OutParticles(int nsnap);
 MethodOfElement:
  virtual void compute_polynomial();
  virtual void compute_acceleration();
  virtual void update_position();
  virtual void update_velocity();
  void read_segment();
  void write_segment();
};
```

The functions declared here are pC++ functions. Their main purpose is to handle I/O. `InParticles`, `InParameters`, and `OutParticles` read input files and write to output files, while `read_segment` and `write_segment` are called by `InParticles` and `OutParticles` to perform parallel I/O. Functions that are already defined in element class `Segment` are declared as virtual functions in this collection declaration. The inherited `Fortran` collection is a parent collection which handles inter-element communication. `Fortran` itself is derived from the `SuperKernel` collection.

The main program is:

```
void Processor_Main() {
  elem_count = pcxx_TotalNodes();
  Processors P;
  Distribution D(elem_count, &P, BLOCK);
  Align A(elem_count, "[ALIGN(X[i], D[i])]");
  SelfConsistField<Segment> X(&D, &A);
  // read initial model
  X.InParameters();
  X.InParticles();
  X.compute_polynomial();
  X.ReduceDoubleAdd(offset,variable_count);
  X.compute_acceleration();
  // main loop
  for (n = 1; n <= nsteps; n++) {
    X.update_position();
    X.compute_polynomial();
    X.ReduceDoubleAdd(offset, variable_count);
    X.compute_acceleration();
    X.update_velocity();
    X.OutParticles(n);
  }
}
```

where `ReduceDoubleAdd` is a reduction function inherited from `SuperKernel`. `offset` is measured from the beginning of the class `Segment` to the beginning of the field `sinsum`, and `variable_count` is the total number of array elements in `sinsum` and `cossum`. A leapfrog integration scheme is used to advance particles.

## 6.2 Benchmark Results

Our experiments with the pC++ SCF code were conducted on a Thinking Machines CM-5, an Intel Paragon, an SGI Power Challenge, an IBM SP-2, and a Cray T3D. For comparison, we also ran the CM Fortran SCF code on the CM-5. 51,200 particles were used for the simulation. The system was allowed to evolve for 100 time steps. The results of these experiments are listed in Table 2

As can be seen, the SCF code scales up very well on the parallel machines. On the CM-5 the pC++ version is about 1.1 times faster than the CM Fortran code. This is mainly because the pC++ code used a faster vector reduction routine, while the CM Fortran code used a scalar reduction routine. The code achieved highest speed—approximately 50 MFLOPS per processor—on the SGI Power Challenge.

| | Number of Processors | | | |
|---|---|---|---|---|
| Platform | 8 | 16 | 32 | 64 |
| Cray T3D | | 223.0 | 115.3 | |
| Intel Paragon | | 667.3 | 332.5 | 168.5 |
| IBM SP-2 | | 186.9 | 103.5 | |
| Power Challenge | 116.9 | 58.6 | | |
| CM-5 (pC++) | | | 45.8 | |
| CM-5 (CM Fortran) | | | 50.3 | |

Table 2: Execution time in seconds for evolving a 51,200 particle stellar system for 100 time steps. The expansions were truncated at $nmax = 6$ and $lmax = 4$.

## 7 The Particle Mesh Code

Another N-body code in the dossier of the GC$^3$ group is the Particle Mesh (PM) code [7]. Originally implemented in Fortran-77 and CM Fortran, the particle-mesh method used in the PM code computes long-range gravitational forces in a galaxy or galaxy cluster system by solving the gravitational potential on a mesh. The three-dimensional space is discretize by a three-dimensional grid. An average density for each grid point is then computed using a Nearest Grid Point scheme, in which the density value at a grid point is the sum of all masses of the particles nearest to that grid point. Once the density values at the grid points are known, Fourier transforms are performed to compute the potential values at those points. The potential values at the grid points are finally interpolated back to the particles, and the particle positions and velocities are updated. The natural data structures for this are a one-dimensional particle list and a three-dimensional mesh.

### 7.1 The Particle List Collection

The particles in the simulation are first sorted according to their affinity to mesh points; particles closest to a given mesh point are neighbors in the sorted list. The sorted list is then divided into segments and each segment forms an element of a particle list collection.

There are two approaches that we can follow when dividing the sorted list. There is a tradeoff between data locality and load balance associated with the two approaches. In the first approach, the sorted list is evenly divided so that the segments have the same length. In the second, particles belonging to the same mesh points are grouped into the same segment and segments will have different lengths.

In the first approach, load balancing is ensured because each processor has the same number of particles. However, this approach may cause particles belonging to the same mesh point to be distributed among different elements, thus requiring more inter-element communication and remote updates. The second approach allows a greater exploitation of data locality, but there is a potential load balancing problem. As the system evolves, particles (stars or galaxies) tend to group together into clumps. Consequently, some mesh points may have 1000 times more particles than other mesh points, and segments that have these mesh points will have much longer lengths. Since we usually distribute the collection elements (in this case segments) evenly across the processors in a parallel machine, the processors that have those long segments will do more work. We therefore decided to follow the first approach.

The `Segment` class is defined as

```
class Segment {
 public:
  int particle_count;
  FArrayDouble x, mass, g, v;
  Segment();
};
```

where `x`, `mass`, `g`, and `v` represent the position, the mass, the acceleration induced by gravity, and the velocity of a particle, respectively.

The `ParticleList` collection is defined as

```
Collection ParticleList : public Fortran {
 public:
  ParticleList(Distribution *D, Align *A);
  void SortParticles();
 MethodOfElement:
  void pushParticles(Mesh<MeshElement> &G);
  void updateGridMass(Mesh<MeshElement> &G);
  ...
};
```

The function `SortParticles()` sorts particles in lexicographic order according to their positions. The particles within each segment are first sorted using the standard C library quicksort function `qsort()`. A global parallel sort is then performed using the bitonic sort of Section 4.

`pushParticles()` uses the gravitational force to update the positions and velocities of the particles. The argument passed to `pushParticles()` is a collection designed for the mesh data structure (see next subection). The mesh collection is passed to `pushParticles()` so that potential values at the grid points can be accessed by particles in the `Segment` element and used to update particles' velocities and positions. The function `updateGridMass()` is used to add the mass of a particle to the total mass of the mesh point to which it is closest. This function first loops through the particles local to a segment and accumulates a local total mass for each mesh point. It then adds the local total mass to the mesh point's total mass by a remote update operation on the appropriate mesh point. Because remote updates are expensive, the particles are sorted to minimize the number of remote updates.

## 7.2 The Mesh Collection

The mesh is logically a three dimensional array of mesh points, each containing values for density and position. Because an FFT is used to solve the gravitational potential equation, the data structure is designed as a one-dimensional collection, each element of which contains a slice of the three-dimensional mesh:

```
class MeshElement {
 public:
  double density[x_dim_size][y_dim_size];
  MeshElement();
  void add_density(double density, int x_zone, int y_zone);
};
```

add_density is remotely invoked by Segment elements to deposit mass on grid points.
The collection Mesh is defined as:

```
Collection Mesh : Fortran {
 public:
  Mesh(Distribution *T, Align *A);
  void computePotential();
 MethodOfElement:
  void xyFFT_forward();
  void zFFT_forward();
  void zFFT_backward();
  void xyFFT_backward();
  void transpose_xy_to_xz();
  void transpose_xz_to_xy();
};
```

The function computePotential() computes the gravitational potential using the total mass at each mesh point. It calls the FFT routines listed under MethodOfElement. The density distribution is first transformed into the wavenumber domain by FFT along the $x$, $y$, and $z$ directions. After solving the Poisson's equation for the gravitational potential in the wavenumber domain, the potential (or force components) is transformed back into the spatial domain.

The FFT transform in the $x$, $y$, and $z$ directions is performed by the Mesh collection. The FFT in the $x$ and $y$ directions is straightforward, since each MeshElement contains an entire array of mesh points. To perform an FFT in the $z$-direction, data are transformed using transpose_xy_to_xz and transpose_xy_to_xy.

## 7.3 The Main Simulation Loop

Given these collections, the main body of the simulation can be implemented as follows:

```
main(){
 int num_of_segments = pcxx_TotalNodes();
 int mesh_dim_z = 64;
 Processors P;

 Distribution Dist_PartList(num_of_segments, &P, BLOCK);
 Align Align_PartList(num_of_segments, "[ALIGN(G[i], T[i])]");
 ParticleList<Segment> part(&Dist_PartList, &Align_PartList);

 Distribution Dist_Mesh(mesh_dim_z, &P, BLOCK);
```

| Platform | Number of Processors | | | |
|---|---|---|---|---|
| | 8 | 16 | 32 | 64 |
| Cray T3D | | 33.4 | 23.1 | |
| IBM SP-2 | | | 81.0 | |
| Power Challenge | 30.4 | 16.1 | | |
| CM-5 (pC++) | | | 134.6 | |
| CM-5 (CM Fortran) | | | 20.4 | |

Table 3: Execution time in seconds for evolving a 32,768 particle stellar system for 10 time steps. A $64 \times 64 \times 64$ grid was used.

```
 Align Align_Mesh(mesh_dim_z, "[ALIGN(G[i], T[i])]");
 Mesh<MeshPlane> mesh(&Dist_Mesh, &Align_Mesh);

 // initialize particle list
 ...
 // main loop
 for (int i = 0; i < number_of_steps; i++){
   mesh.computePotential();
   particlelist.pushParticles(mesh);
   particlelist.sortParticles();
   particlelist.updateGridMass(mesh);
 }
 ...
}
```

The main loop involves computation on both the `Mesh` and `ParticleList` collections. First the potential is computed in parallel on the grid. Second, the particle velocities and positions are updated. If particles have moved to new grid points, the appropriate data structures are then updated. The particles are then sorted, after which the particle masses are accumulated in their corresponding points for the next iteration step.

## 7.4 Benchmark Results

Our experiments with the pC++ PM code were conducted on a Thinking Machines CM-5, an Intel Paragon, an SGI Power Challenge, an IBM SP-2, and a Cray T3D. For comparison, we also ran the CM Fortran PM code on the CM-5. 32,768 particles were used for the simulation. The system was allowed to evolve for 10 time steps. The results of these experiments are listed in Table 3.

As can be seen in the table, the code scales up relatively well on the T3D and Power Challenge. On the CM-5, the pC++ code is considerably slower than the CM Fortran code. This is because the CM Fortran code can make use of transpose routines embedded in an FFT developed by Thinking Machines' engineers. The pC++ code has complicated data structures and cannot use those transpose routines. Again, the best performance was obtained on the Power Challenge, although this architecture is limited to a small number of processors.

## 8 Conclusion and Project Evaluation

pC++ offers a very simple data-parallel programming model which makes use of the object-oriented feature of C++. Very few parallel computations have not proven to be well-suited to this model of computation. However, we have discovered a number of serious limitations in our system. Some of these can easily be overcome, but others have led our research in new directions.

**The current pC++ compiler is not well suited to support nested data parallelism.**

Computations for which the available concurrency is nested or "multi-level" are among the most interesting. Many of the important problems that confront the $GC^3$ effort involve dynamic, adaptive data structures. More specifically, multi-level, adaptive grid techniques, which are becoming standard in the simulation world, are not easy to express without support for dynamic, nested parallelism.

As a simple example, consider the problem of supporting collections of collections in the runtime environment. Because the current pC++ preprocessor translates the single threaded data parallel style into direct SPMD emulation of data parallelism, it is very difficult to allow nested parallel operations. Our thinking here has been greatly influenced by the NESL project at Carnegie-Mellon University [1]. This research has demonstrated that a wide variety of nested parallel computations can be "flattened" by the compiler and runtime system to produce very efficient code. However, most of the examples where this works are relatively static in structure; it is not clear how well this technique works for very dynamic, adaptive computation.

**C++ has been a moving target.**

The majority of parallel C++ efforts were constructed without thinking about the impact of the template system and the Standard Template Library (STL). Templates and the STL have introduced concepts into the C++ programming methodology that are different from the standard object-oriented concepts that most users understand. However, the template mechanism in C++ and the STL have taught us to think about libraries of "generic functions" that work in harmony with object-oriented design ideas. Our future work on parallel C++ will embrace these concepts.

**C++ users are not willing to accept radical extensions to their programming language.**

While experimenting with parallel programming language ideas is exciting, extensions to C++ will have little impact unless there is consensus on a very small set of changes to the language. Users require programming environments that are stable and that are supported by all vendors. For the future it may be important to design a meta-level control extension facility for C++ (similar to the Japanese RWPC C++ system [16]) so that language extensions are needed. With this approach, new parallel constructs could be added to the language by providing a library that would work with any C++ system.

**It is a mistake to design a system that limits the parallel programming paradigms which can be used.**

One of the problems with pC++ is that it supports only one paradigm for writing parallel algorithms. While a data parallel object-oriented style is good for some applications, it certainly does not cover all applications. For example, it is not very easy to simulate the concurrency in an operating system with a data parallel language. ICC++ and COOL are interesting because they mix general parallel control constructs with some special new object-oriented features. In general, it seems best to provide very simple primitives on which users can implement a variety of different programming paradigms. We feel that CC++ does very well in this respect.

**Basic C++ optimization is still a major obstacle to performance on most systems.**

One of the most frustrating problems with high performance computing in any language is the low quality of code optimization compared to Fortran. This is one reason a clean interface to parallel Fortran is very important. The OOPACK benchmark tests compilers to see how well the code generators work on programs that are designed using standard C- and Fortran-like iteration, C++ style type and operator extension and C++ STL-style iterators. Very few of the high performance systems are able to optimize the more advanced programming constructs. As we learn to rely on template based class libraries, good optimization techniques will become more complex and more important.

**The evolution of C++ is driven by standards and ideas from many sources.**

In the distributed computing community the Object Management Group (OMG) has established a standard for distributed object systems called the Common Object Request Broker Architecture (CORBA). The Object Data Base Group will soon describe its standards. All of these represent technologies that must be considered when we think about parallel computation in object-oriented terms.

## References

[1] G.E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, April 1993.

[2] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An Object Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In M. Chapman and A. Vermeulen, editors, *Proceedings of the Second Annual Object-Oriented Numerics Conference*, Corvallis, OR, 1994. Rogue Wave Software.

[3] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[4] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and Y. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, May 1994.

[5] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press, 1993. ISBN 0-272-01139-5.

[6] J. Choi, J.J. Dongarra, and D.W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In H.J. Siegel, editor, *Proc. Eighth International Parallel Processing Symposium.* IEEE Computer Society Press, April 1994.

[7] R. Ferrell and E. Bertschinger. Particle-Mesh Methods on the Connection Machine. *International Journal of Modern Physics C*, 1993.

[8] D. Gannon. Libraries and Tools for Object Parallel Programming. In *Advances in Parallel Computing: CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing, Saint Hilaire du Touvet*, volume 6, pages 231–246. Elsevier Science Publisher, 1993.

[9] D. Gannon, V. Guarna, and J.-K. Lee. Static Analysis and Runtime Support for Parallel Execution of C. In *Languages and Compilers for Parallel Computing*, pages 254–274. MIT Press, 1993.

[10] D. Gannon and J.-K. Lee. Object Oriented Parallelism: pC++ Ideas and Experiments. In *Japan Society for Parallel Processing*, pages 13–23, 1991.

[11] D. Gannon and J.-K. Lee. On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions. In Bourge and Cosnard, editors, *Proceedings of CONPAR 92–VAPP V*, pages 769–774. Springer Verlag, 1992.

[12] D. Gannon, N. Sundaresan, and P. Beckman. pC++ Meets Multithreaded Computation. In J.J. Dongarra and B. Tourancheau, editors, *Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 76–85. SIAM Press, 1994.

[13] D. Gannon, S. Yang, P. Bode, and V. Menkov. Object Oriented Methods for Parallel Execution of Astrophysics Simulations. In *Mardigras Teraflops Grand Challenge Conference.* Lousiana State University, 1994.

[14] J. Gotwals, S. Srinivas, and D. Gannon. pC++/streams: a Library for I/O on Complex Distributed Data Structures. In *Symposium on the Principles and Practice of Parallel Programming.* ACM, 1995.

[15] L. Hernquist and J.P. Ostriker. A Self-Consistent Field Method for Galactic Dynamics. *The Astrophysical Journal*, 386:375–397, 1992.

[16] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and T. Tomokiyo. RWC Massively Parallel Software Environment and An Overview of MPC++. In *Proceedings of the Workshop on Parallel Symbolic Languages and Systems*, 1995.

[17] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[18] M. Lemke and D. Quinlan. P++, a Parallel C++ Array Class Library for Architecture-Independent Development of Numerical Software. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 268–269, 1993.

[19] B. Stroustrup. Parameterized Types for C++. In *USENIX C++ Conference, Denver*, October 1988.

[20] S. Yang, D. Gannon, S. Bodin, P. Bode, and S. Srinivas. High Performance Fortran Interface to the Parallel C++. In *Scalable High Performance Computing Conference.* IEEE, 1994.