User Guide for a Portable Parallel C++ Programming System, pC++ *

Dennis Gannon, Shelby X. Yang, Peter Beckman
Department of Computer Science & CICA
Indiana University, Bloomington, Indiana, U.S.A.

September 9, 1994

Abstract

This document describes version 1.0 of the pC++ system for building portable parallel programs. It is intended for users that are already C++ programmers and would like to use *data parallel* style operations in C++ using *arrays of objects* in which object member functions can be applied in parallel. To accomplish this, pC++ provides a very simple mechanism to build "collections of objects" from some base *element* class. Member functions from this element class can be applied to the entire collection (or a subset) in parallel. In addition there is a mechanism for encapsulating SPMD style computation in a thread based computing model that is both efficient and completely portable.

*This research is supported by DARPA under contract AF 30602-92-C-0135 from Rome Labs, National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616

pC++ User Guide

Contents

1 Introduction

In its current form pC++ is a very simple preprocessor that generates C++ code from the pC++ extensions to C++. Just as the original AT&T C++ system used a preprocessor called Cfront to translate C++ to C, pC++ can be considered a C++front, i.e. a translator from an extension of C++ down to C++. Consequently, to run pC++ programs you need a current implementation of C++ (AT&T version 3.0 or better, or g++ version 2.4.5 or better) running on, or cross compiling for, your target machine.

pC++ currently runs in uniprocessor mode on workstations from Sun, SGI, HP, NeXT, DEC and IBM. This allows one to partially debug parallel programs as uniprocessor tasks. Unfortunately, you get no special performance win from running parallel programs on these machines and there are parallel program bugs that may not be discovered during serial execution. On the other hand, if you have a Thinking Machines CM-5, Intel Paragon, Sequent Symmetry, BBN TC2000, or KSR-1, you can run pC++ programs that exploit parallelism. (The IBM SP1 port is nearly completed, and Cray T3D will be the next target.) More important, a pC++ program, if it has been designed with care, will run on all these machines without changing a line of code. In addition to sequential execution on workstations, there are two parallel simulators that pC++ programs will run on. These simulators allow one to emulate true parallel execution on a sequential machine. However, our experience indicates that once a program works with the sequential pC++ system on a workstation, it is an easy job to port it to the parallel platforms. And once it is running on one parallel machine, it simply requires recompilation to run on any of the other parallel platforms.

This brief document will outline some of the main points of the language and how to use it. We will also try to outline some of the problems you will encounter. This is an "alpha" version of free software and there are bound to be many many bugs. Version 1.0 will be followed very quickly by Version 2.0 which will have many more features. However, there are certain things to know in advance about the limitations of pC++. These include

- pC++ DOES NOT provide a way to get parallelism out of a network of workstations. However we are currently working on a PVM port, and should have a prototype running before the end of 1993. This is an exciting area and others are working on it. Check out CC++ from Caltech.
- pC++ DOES NOT do large grain data flow. Look at Mentat from the University of Virginia.
- pC++ IS NOT a good foundation for implementing a concurrent operating system.

In the future we hope to be able to mix modules of pC++ with code designed for these other systems, but that will not work right now.

In addition to the limitations to pC++ as a programming language and model, there are limitations of Version 1.0 that WILL BE FIXED in Version 2.0. These limitations are listed below.

- Version 1.0 of pC++ does not yet have a HPF or node level Fortran interface. (to be fixed in version 2.0)

- Version 1.0 of pC++ does not support some of the standard C++ functionality. Included in this are things like the overloaded comma operator. More important, support for C++ templates is weak. A more complete list of known preprocessor bugs is included in the appendix.
- Version 1.0 of pC++ does not support separate compilation when pC++ extensions are spread across multiple files. The rules for separate compilation will be described later.
- Version 1.0 of pC++ does not support the CM-5 vector units directly, but you can link CMF node Fortran subroutines into pC++ class functions. This is described later in this document,
- Version 1.0 of pC++ has none of the good optimizations that you might expect. Versions 2.0 and 3.0 will focus on these optimizations.
- Version 1.0 of pC++ has very limited I/O. In particular, the C++ stream library is not integrated because it will need a rewrite for the parallel environment. I/O issues are considered in detail in a later section.

In the following section we cover both the current alpha release, pC++ version 1.0 and the next release, pC++ version 2.0. We have included both in this document so users of version 1.0 can know what direction pC++ is going. We have indicated with each section header which version number that is required. The symbol *Ver. 1.0+* means that this applied to the current version as well as future versions. The symbol *Ver. 2.0* means that this is not implemented in version 1.0.

2 Installing the System. Ver. 1.0

The following comes from the file INSTALL in the standard software distribution. It is best to read that file instead of this section to make sure you have the most recent information.

In order to save disk space, we have created three different pC++/Sage++ distributions. You may select the distribution that best fits your planned use. All distributions are Unix "tar"ed and GNU gzipped.

- `sage++-*.tar.gz` The Sage++ compiler tools and the c, c++, and fortran parsers. The pC++ run-time system and code generator are not included.
- `pc++-*.tar.gz` Just enough of Sage++ to compile pC++. Includes run-time systems for several parallel machines.
- `pc++sage++-*.tar.gz` Everything! The Sage++ tools and the pC++ run-time system.

.

The tools required to make pC++ are:

- An ANSI C compiler (The compiler MUST be ANSI compatible)
- A C++ compiler (either g++ AT&T's C++, DEC's cxx, or IBM's xlC)
- About 25-80 Meg of disk space for compilation.

A word about compilers. Unfortunately, the vision of portable Unix programs has yet to be fulfilled. Since every vendor's C compiler and header files vary slightly, pC++/Sage++ uses gcc/g++ by default. This choice was also influenced by companies like SGI, SUN, and HP "unbundling" their ANSI C compilers, and hoping people pay extra for their compilers.

However, relying on gcc/g++ is a problem for parallel machines that use use custom CPUs for which gcc is not defined, such as the KSR1. Therefore we also support AT&T's C++ compiler, which can be ported to almost any machine with a good C compiler.

2.1 Installation

Get the most up-to-date version from the FTP archive. Use "finger sage@cica.indiana.edu" to learn about the current version and where to find it.

1. Move the archive to the directory where tar will create the distribution directory. A good choice is `/usr/local/src` (if you are root) or your home directory (if you are a user).
2. execute `gzip -d -c <distribution>-*.tar.gz | tar xvf -`
3. From within the distribution directory, run the `./configure` script. This will automatically modify the Makefiles to include the correct libraries and/or compile flags for your architecture. You can override the defaults with `./configure -cfront`, or `./configure -gnu`. You may also choose compilers with `./configure -cc=cc -c++=CC`. You may specify the architecture as the last parameter if `./configure` cannot automatically determine it.

Please inform us if there are other modifications required for your system. Running "`./configure unknown`" will return files to their original form.

4. `make install` At this point, simply running "make" will NOT work; "`make install`" is required. This will invoke a recursive make for each of the source subdirectories.

   If you are using a supported parallel machine, the machine specific runtime module will also be compiled. `./TestSuite` will also be compiled for uniprocessor simulation. See `./TestSuite/README`. If all went well, you should be ready to test the uniprocessor version with "`cd TestSuite; make test`".

5. Reprint this user manuals in `./doc` (See the `./doc/README`) because it may have changed.

6. You may "`make clean`" after you have installed the system to remove all the .o files, and really save on disk space. The libraries (*.a) and binaries will be preserved until you "`make cleaninstall`", which gets rid of everything but the source code.

3 The pC++ Programming Language.

3.1 Threads, Processors and Runtime Models Ver. 1.0+.

We define a *thread of execution* to mean an abstract "program counter", i.e. a mechanism to evaluate expressions and follow the control flow of a program, together with a name space of variables and code segments in an application. Without getting too technical, a thread represents the activity of one processor (or process) in a typical MP system. For example, in a conventional uniprocessor environment each program executes with only one *thread* i.e. the one that starts with the invocation of the `main()` function and sequentially steps through the code. That thread can see and modify the global variables declared outside of `main()` and the variables on the procedure call stack as it executes.

There are many different ways in which multiple threads can be used to speed the execution of a program. In a *master/worker* "multi-threaded" execution model, we use one control thread, called the main, master, or scheduler, which starts with `main()` and a set of "worker" threads which help out on the parallel part of the program. It is the job of the master to signal the workers where to begin execution and where to find the data they should work on. This is accomplished by identifying loops in the program that can be executed in parallel and then distributing iterations to the worker threads as tasks. The rules for how names spaces of variables and data interact between different threads vary from model to model. In some systems, there is a shared name space of variables and all threads can see and modify these variables and, of course, suffer the consequences when two threads try to modify the same variable without proper synchronization.

Another model is the *Single Program Multiple Data*, or *SPMD* mode of execution. In this case, a set of threads each start executing the program from the beginning and follow it all the way through. The only thing different in the execution from one thread to another is that different threads operate on different data sets and different subsets of the parallel loop iterations space. In the SPMD model it is assumed that the name spaces of variables are all distinct. That is, if `X` is a name in a program then each thread sees a different `X`. Consequently, sequential work is duplicated but parallel work is shared and some sort of communication protocol is used share information.

In a pC++ program, there is a distinguished *sequential main thread*, invoked from a function called *Processor_Main()* (as it is called in Version 1.0. It will be known as *main()* in future versions). This thread is the main *control thread* and it is the scheduler of parallel operations. To define additional computational resources beyond the main thread the pC++ programmer creates processor objects, which define a set of threads.

Parallel tasks are defined in terms of threads that are associated with a *processor object* which defines a set of threads and their bindings to processors, i.e. physical computing resources.

One declares an array of processor objects with a statement of the form

```
Processors P(m,n)
```

This defines P to be a 2 dimensional array of "virtual processor objects", i.e. execution threads, of size m by n. In general, processor objects can be of any dimension of any size. However, there is a very serious and disappointing limit on what is possible for Version 1.0 of pC++. As mentioned above, current compiler, operating system and hardware technology only allows one virtual processor object per physical object and this set of threads is declared as

```
Processors P;
```

The reason for this limitation is that on current distributed memory machines, there is no support for communication mechanisms that operate on anything other than a physical process level. In general, software multitasking or special hardware can allow more than one thread of execution per physical processor and future versions of pC++ will support this extension.

To make a thread operate in a C/C++ world, one also needs a way to define program variables that are visible to the threads as well as variable that may be modified by that thread. Note that we make a distinction between variables that are visible and those that can be modified. Just as a `const` variable in C++ may be initialized and not modified, a global (file static) variable in a pC++ program can be seen and modified by the main thread, but only be seen by the processor objects. In other words, if a variable is a global static or external variable for the main thread, it is a `const` variable for the processor object threads. The reasons for this limitation will be described later.

In addition to rules for variable visibility for virtual processor objects, we need a way for the main thread to identify and invoke sections of code for the worker threads to execute. This is accomplished through Thread Environment Classes.

3.2 Thread Environment Classes. Ver. 2.0

Thread Environment Classes are not directly supported at the user level in Version 1.0, Because they will be supported in 2.0 and because they are an important part of the runtime model, we discuss them here.

Once one defines a set of processor threads, it is necessary to define an execution environment for each thread. In addition, one needs a mechanism to assign work to threads. This is accomplished with a *Thread Environment Class* which is a basic extension that pC++ makes to C++.

In its most basic form a *Thread Environment Class* defines a set of local variables for a processor object and a mechanism in which the main thread can start each thread executing a well defined unit of work.

A thread environment class looks like an ordinary C++ class. The syntax is identical to a standard C++ class, but with a different keyword.

```
TEClass class_name: super_classes {
     member_data_definitions;
```

```
    member_function_definitions;
};
```

A regular C++ class defines a data type that consists of a structure of data field members and member/method functions that have special privilege to read and modify these data fields. A TEClass is identical except that it defines a structure of data fields that are private to each thread represented by a given `Processor` object. The invocation of member functions of a `TEClass` object by the main control thread transfers control to the individual threads for the duration of the execution of that function.

A `TEClass` is declared the same as any other class with the exception that

- When defined within the global name space of a program or allocated within the main control thread, the constructor of a `TEClass` object must contain a processor object as the last parameter. Upon invocation of this constructor, one copy of the member field object is allocated to each processor object thread defined by the argument. The lifetime of these objects is determined by their life time in the control thread.
- If a `TEClass` object constructor is invoked by a thread of a processor object, it is invoked as a regular class constructor for that thread. In this case, one does not supply the Processor parameter.
- A call to a `TEClass` member function by the main program control thread represents a transfer of control to a parallel action on each of the threads associated with the object. The main control thread is suspended until all the processor threads complete the execution of the member function. If the member function returns a value to the main control thread, it must return the same value from each PO thread or the result is undefined.
- If a `TEClass` member function is invoked by one of the processor object threads, it is a sequential action by that thread. (Hence, there is no way to generate nested parallelism with this mechanism.)
- Each thread may see the same scope of variables that an ordinary class member function can see. However, there is a major difference between a TEClass object and a regular class object. An ordinary class member function can read and modify any global variables visible from the scope of its declaration. However, in the case of `TEClass` objects, member functions cannot modify the "global" variables in the program defined outside the scope the `TEClass` declaration.

The invocation of a `TEClass` member function by the main control thread is the mechanism for transferring control from the main thread to the worker threads defined by a `Processors` class used to create the `TEClass` object. While a worker thread is in execution of a member function, it can only see the data members of that `TEClass` object instance and the program global variables that are visible from the scope of the `TEClass` definition. Figure 1 gives an illustration of the multithreaded runtime model that the user should have in mind when programming with `TEClass` objects.

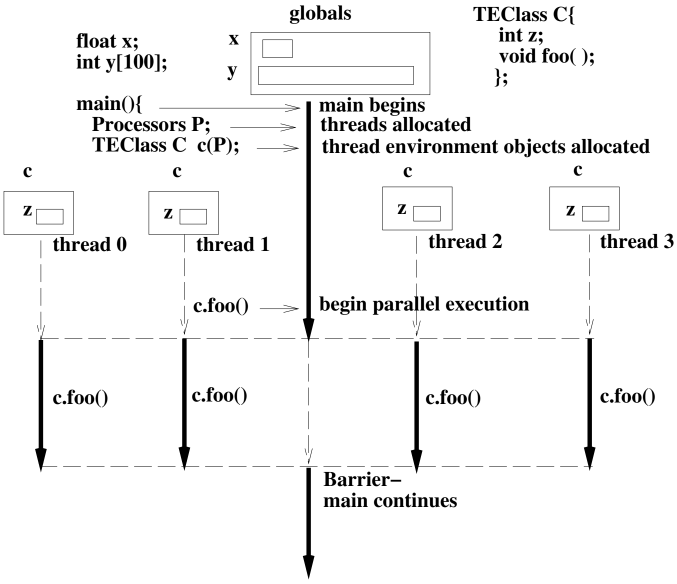

Figure 1: Multithreads and TEClass

To understand this control flow and these visibility rules, it is best to consider a simple example. The code below defines a simple thread environment class and the member functions that access it.

```
 int my_global;
 float a_global_array[3];

 TEClass C{
     int x, y[100];
   public:
      int i;
      void f(int j){ i = j; }
  };

main(){
  Processors P;
  C myThreadEnv(P);

  myThreadEnv.f(2);
}
```

In this case the main thread defines a set of processor object threads called `P`. The thread environment object, `myThreadEnv` is next allocated. This object is actually a set of instances of objects defined by the class you would get if you replaced the key word `TEClass` by `class`. The constructor calls this with the processor object `P`, so there is one instance of an object which contains one copy of an integer variable x and one copy of an array y[100] for each processor object associated with P.

The constructor for this set of objects is executed by each processor object bound to it. While this is taking place, the main thread is suspended. In our example, there is only a default constructor, so control returns directly to the main thread.

When the main thread continues, it invokes `myThreadEnv.f(2)`. Again, this is a signal for the processor-object/worker-thread to take over. Each thread executes `myThreadEnv.f(2)` applied local instance of the TEClass object that was allocate to it. Once the worker threads each start evaluating a member function of a TEClass object, they are free to call other member functions, or normal C functions. There are three important restrictions that need further investigation.

1. A processor object, while executing a TEClass member function may read the variables declared in the global scope of the same file, in this case `a_global` and `my_global_array`. However, processor thread may NOT MODIFY these global values.
2. There are some restrictions on I/O to and from the TEClass member functions.
3. When a TEClass member function returns a value to the main thread it is up to the programmer to make sure that this is well defined.

We will discuss each of these restrictions and illustrate them with examples.

The first restriction is very important. At first sight, it may seem strange that a processor object thread cannot modify global variables. However, keep in mind that allowing them to do this can result in complex race conditions. That is, without some form of global synchronization mechanisms, there is no way to guarantee that one processor thread will modify a variable before another processor reads the variable. Consequently, one cannot easily predict the behavior of programs that ignore this rule.

A second, more pragmatic reason for this restriction is that pC++ is designed to be portable across distributed memory as well as shared memory systems. On distributed memory machines the global data is duplicated in each processors address space and the compiler generates SPMD style parallelism for these machines. If a processor threads were allowed to make arbitrary changes to global variables, there is no way for the compiler to make sure they maintain a consistent state across address spaces. (This is because it is not possible to completely solve the problems associated with aliases caused by pointers in C.) To get around this problem, the main thread is the only one that is allowed to modify these variables. Because, in an SPMD execution environment, the main thread is duplicated across process, it is a much easier task to make sure that the main thread is identical on each processor.

To see this restriction in more detail consider these cases

```
int a, b;

int foo(){a++; }
```

```
int bar(int *p){ *p = 0; }

TEClass C{
  public:
        int x;
        void f(int *z){
           foo();  //<<< error: modifies global state
           bar(z); // may be o.k.
           }
};

main(){
    Processors P;
    C T(P);
    foo();     // o.k.
    T.f(&a);   //<<< error: because bar will modify a
    T.f(&T.x); // o.k.
}
```

In this case the function `foo()` can be called from the main program, but not from a TEClass member function because it modifies a global variable. However, the call to `T.f()` is problematic. In one case, it is an error because it calls `bar()` with the address of a global variable and in the other case it operates on a member field of the TEClass instance hence the call `T.f(&T.x)` is valid.

Perhaps the most perplexing rule associated with TEClass objects has to do with values returned to the main control thread. When a TEClass method is invoked by a processor thread, it behaves exactly like any other class member function. However, when a TEClass member function is invoked by the main thread the action is different. Each processor object thread executes the function and each will return a value. Unfortunately, the calling thread is only expecting a single value to be returned. To avoid non-determinism in the program the programmer is required to make sure these values are all identical.

To illustrate this consider the following simple example. We will use two special pC++ intrinsics. `MyProc()` is a function that may be called by a processor object thread to determine its "thread identifier" which is a value between `0` and `p-1` where `p` is the number of processor threads that are allocated in a processors object. (In version 1.0 of pC++, this corresponds to the processor number that the thread is assigned to.)

The second function `pCxx_max()` is a pC++ intrinsic function that selects the maximum of a set of values from each of the processor threads. (This function is a member of a larger family of special TEClass reduction functions that will be described in greater detail below.)

The example illustrates both the correct and incorrect use of the TEClass member function return mechanism.

```
TEClass C{
```

```
   public:
     int f(){ return MyProc(); }
     int g(){
        int y = f();  //<<< call to f o.k.
        return pCxx_max(y);
        }
};

main(){
  Processors P;
  C MyThreads(P);
  int x = MyThreads.f();  //<<< error.  multiple return values for f.
  int max_thread_id = MyThreads.g();  //<<< o.k.
}
```

In this example, the function `f()` returns a different value for each processor object thread. Consequently, it does not make sense to call it from the main thread. However, its meaning as a scalar valued function is well defined within the individual thread environments: it simply returns the identifier of that thread. On the other hand, the function `g()` uses the `pCxx_max()` reduction to select the largest value returned by each of the calls to `f()` as a return value, so each thread object returns the same value.

Note that there are several alternative designs that pC++ could have used that would avoid this "single value return value" restriction. First, one could stipulate that the main thread would "pick" a value at random from the set of values generated on each thread. However, experience has shown us that users rarely want this feature. In addition, this scheme is non-trivial to implement. Alternatively, one could devise a mechanism that would allow sets of values to be returned and assigned to scalar variables just like a scalar return value. Unfortunately, this would violate the C++ type system. However, pC++ does provide a way to create and manipulate aggregate values and this topic is discussed in the chapter on Collections.

3.2.1 Thread Reduction Functions. Ver. 2.0

The most common use of returning a single value to the main control thread is to reduce, or summarize, the information computed by each thread of a processor object. Each of these functions are primitive, "built in", member functions of TEClass objects. In addition to computing and returning a uniform value to each processor object thread, they provide a way to synchronize the computations of each thread. The reduction functions available in pC++ are summarized in the table below. They are described in terms of a data type `T` which is assumed to have operators `>`, `==`, `>=`, `+`, `*` which obey an associativity law. `T` may be a standard base type like `int` or `double`, or it may be a user defined type subject to some simple restrictions which we will describe later.

- `T pCxx_max(T x)` returns the maximum of a set of thread values.

- `T pCxx_min(T x)` returns the minimum of a set of thread values.
- `T pCxx_sum(T x)` returns the sum of a set of thread values.
- `T pCxx_product(T x)` returns the product of a values distributed over a processor set.

It is important to understand that these functions require the involvement of every thread in a processor set. For example, the following code illustrates two attempts to sum a set of values over the odd numbered threads of a TEClass object.

```
TEClass C{
   float x;
   float oddSum(){
      if(MyProc()/2 != 0)    //<<< error: only odd threads
         return pCxx_sum(x); //<<< are executing this line
      else return 0;
      }
   float tryAgain(){
      float y;
      if(MyProc()/2 != 0) y = x; else y = 0;
      return pCxx_sum(y);
      }
 };
```

In the first attempt, `oddSum()`, only the odd numbered threads execute the sum reduction. Consequently, they will "hang" waiting for the even numbered threads to contribute their part. In the second case, `tryAgain()`, all threads execute the reduction and there is no problem.

3.2.2 Communication Between Threads. Ver. 2.0

Because one of the reasons for introducing TEClass structures into pC++ was to allow SPMD style computation to co-exist with the data parallelism, we must also consider the issues related to the portability of SPMD code. Until fairly recently, there have been no standard ways to make message passing code portable. However, there are now several ways in which one can write portable message passing software systems. PVM is one approach and others include Express, P4 and MPI, the new message passing standard. We view MPI as being still a bit limited in that there is no provision for communication between threads on the same processor, but we also expect it to evolve.

In the short term, we are providing a simple mechanism to support a limited class of communication operations in pC++ between threads of a processor object. In the longer term we will integrate MPI and PVM support when we are sure we can find implementations on all target systems.

For now, there are three simple communication routines. The first is a way to broadcast data to each thread TEClass object member function from any other. The function, `pCxx_BroadcastBytes()` takes three arguments.

```
void pCxx_BroadcastBytes(int source, int length, void * buffer)
```

The argument `buffer` a a pointer to a buffer that exists in the private address space of each TEClass representative thread. The argument `source` is a flag which, if set to zero, determines the source of the buffer that should be copied to each of the other buffers. Only one thread may execute the function with the argument set to zero and all others must execute the function with a non-zero value. All representative threads in a Processors object must execute this function if one of them executes it. For example,

```
TEClass C{
   float x;
   void sendPi(){
       if(MyProc() == 0) x = 3.1415;

       // now send the value of x to each of the other threads
       pCxx_BroadcastBytes(MyProc(), sizeof(float), &(this->x));
      }
```

The idea behind `pCxx_BroadcastBytes()` is to have a mechanism to share a value among all thread that may have been computed by only one thread. More direct communications between threads is achieved by a blocking send called `pCxx_send()` and a blocking receive `pCxx_receive()`. They are defined as follows

1. `void pCxx_send(int dest, int length, void *buffer)` which send a message to thread `dest` of specified `length` in bytes that contained in the specified `buffer`. The functions returns as soon as the buffer is free to be used again.
2. `int pCxx_receive(int *from, void *buffer)` when called will wait until a message has been received by the executing thread. The value returned from the function is the length, in bytes, of the message stored in `buffer`. The index of the sending thread is stored in `from`.

As we do not yet recommend the use of these function, we will not provide examples. They will be part of the next release, but we expect that PVM or MPI will be the preferred mode of communication.

3.2.3 Constructors for TEClass Objects. Ver. 2.0

The examples from the previous sections, used only one type of constructor for TEClass Objects: one that that had a single Processors object parameter. This constructor is automatically generated by the compiler. User defined constructors can be of any standard C++ form. THE COMPILER WILL MODIFY THESE SO THAT THE PROCESSORS PARAMETER IS ADDED AS THE LAST ARGUMENT. For example,

```
TEClass A{
  int n;
```

```
  float data[100];
  A(int size){
     n = size;
     for(int i = 0; i < size; i++) data[i] = 0.0;
     }
};

main(){
 Processors P;
 A vectors(64,P);
 A *x = new A(32, P);  // another valid constructor.
 A *local = new A(32);  // generates a local object. (not supported in v.1.0)
}
```

In this case the system binds the value 64 to the size argument in the users constructor and uses the value `P` to map the objects to processor threads.

If a TEClass object is built without the processor parameter, as in the third declaration above, a single representative object is created that is identical to a single instance underlying class. (Not supported in version 1.0).

3.2.4 Encapsulating SPMD Libraries. Ver. 2.0

One of the reasons for including Thread Environment Classes in pC++ is to provide a mechanism to encapsulate code that is designed for execution in a message-passing SPMD environment. This includes many of the libraries that have been designed at the national laboratories such as Lapack++, AMR++, and many more.

To understand how this works, consider an example of a matrix class `Matrix` defined as follows

```
TEClass Matrix{
  double **data;
 public:
  int rows, cols;
  Matrix(n,m,p);
  void matMul(Matrix &A, Matrix &B);
  double &operator ()(int,int);
 };
```

In an SPMD style execution, the matrix object would be created on each processor participating in the computation and the constructor, given global dimensions m by n would automatically partition the data over the p processors. The interesting part of these libraries is the way processor communication is managed. In a typical application every processor must participate in each matrix operation done in parallel. All communication is hidden within the class operators and the resulting "user code" looks exactly like sequential code. (The version 1.0 pC++ compiler for distributed memory machines works exactly in this manner.) Take for example the way the library designer would implement the operator `matMul()`. Let us assume that the library is designed so that rows are partitioned over the processors. That is, rows *(0, n/p -1)* are on processor *0*, rows *(n/p, 2n/p-1)* on processors *1*, etc. The SPMD code for `matMul()` would look something like the following. Each processor has part of three matrices, A, B and `*this`. The code below first broadcasts a column of B to each thread which then assembles the pieces of the column and computes the appropriate dot product of that column with its share of the rows of A.

```
void Matrix::matMul(Matrix &A, Matrix &B){
    int i,j,k,s, n, m, r, p, from;
    p = NumProc(); //NumProc() gives the number of processor threads
    k = A.rows;  m = cols, n = rows/p; r = k/p;
    double *rowbuf = new double[k];
    double *buffer = new double[r];
    for(i = 0; i < m; i++){
       // boadcast a column of B to each processor
       for(j = 0; j < p; j++){
         for(s = 0; s < r; s++) buffer[s] = B.data[s][i];
         pCxx_send(j, r, buffer);
         }
       // assemble column blocks into a B row
       for(j = 0; j < p; j++){
         pCxx_receive(&from, buffer);
         for(s = 0; s < r; s++) rowbuf[from*r+s] = buffer[s];
         }
       for(j = 0; j < n; j++)
           for(s = 0; s < k; s++)
                data[i][j] += A.data[i][s]*rowbuf[s];
     }
}
```

This version of the program is not optimal (a blocked version should be used) but it is easy to understand and it is typical of the style of SPMD libraries.

This function can now be called with a pC++ main program as follows

```
Processor_Main(){
   Processors P;
   Matrix C(n,m), A(n,k), B(k, m);
   ....
   C.matMul(A, B);
};
```

A more interesting problem is that of the element reference operator. The job of the `( int, int )` operator is to make sure that any read or update from the main thread is propagated to the correct position in this distributed array. For example, if the main thread invokes

```
x  = M(i,j);
```

then the thread that contains the $(i,j)^{th}$ element must return the correct value to the main thread. On the other hand, if we call

```
M(i,j) = x;
```

then it is the job of the `(...)` operator to make sure the $(i,j)^{th}$ on the correct processor is updated. This problem is complicated because we are not sure which thread may be invoking this operator. If it is a thread for which the requested data reference lies in the same address space, there is no problem. However, if they are different, such as when the main thread invokes this operation on each worker, we have a problem. To see this difficulty consider the following possible implementation.

```
double dummy_buffer;
double &Matrix:: operator( int i, int j){

   double *z;
   int not_local = 1;
   int p = NumProc();
   if (MyProc()*(rows/p) <= i) && (i < MyProc()*(rows/p)){
        // I have the desired row!
        z = &(data[ i % p][j]);
        not_local = 0;
}
   else z = &dummy_buffer;
   pCxx_BroadcastBytes(non_local, sizeof(double), z);
   return *z;
}
```

If each of the worker threads associated with the TEClass executes this operation, then the reference evaluation will be correct when called by the main thread only if the main thread shares its address space with one of the worker threads. (This is the case in the current ver. 1.0 pC++), However, in the future versions, this may not hold. There are two solutions to this problem. One solution is to introduce the CC++ `global` data type qualifier, so that special pointers and references can be created that can be passed between address spaces. We are strongly considering this for ver. 2.0. The other solution is to introduce more explicit member functions for read and write operations.

```
TEClass Matrix{
  double **data;
  double &operator ()(int,int);
 public:
  int rows, cols;
  Matrix(n,m,p);
  void matMul(Matrix &A, Matrix &B);
  double read(int i, int j){ return (*this)(i,j); }
  void write(int i, int j, double value){ (*this)(i,j) = value; }
 };
```

This solution protects the (...) operator to be only used in the TEClass thread environments and it allows only data values (instead of references or pointers) to be passed between address spaces.

4 Collections. Ver 1.0+

Thread environment classes provide a simple, portable mechanism for a limited form of parallelism. However, a more important abstraction is based on a structure called a concurrent aggregate, or collection.

The motivation is to describe a structure that defines a set of objects of some given type that can be distributed over the memory modules of a parallel system.

The way in which elements of a collection are distributed over processors is determined by a two-step mechanism similar to HPFF Fortran. In the first step, collection elements are mapped to a logical coordinate system. This first mapping is called an alignment and it is defined by an `Align` class object. A `Distribution` class object defines the logical coordinate system and the way in which it is mapped to Processors object threads.[1]

The declaration of a collection is specified by a collection type name followed by a type name of the element objects enclosed in angle brackets. The arguments for the three constructors define these collections in terms of alignment and distribution objects.

```
collection-name <  element-name >  object( constructor arguments )
```

For example, suppose we want to create a matrix, A, and two vectors, X and Y, of complex numbers and want to distribute them over processors of a parallel machine. Given a C++ class for complex numbers, *Complex*, we can build distributed matrix and vector collections by using the pC++ library collection classes *DistributedMatrix* and *DistributedVector* as follows:

```
DistributedMatrix<Complex>  A(...constructor arguments ...);
DistributedVector<Complex>  X(...constructor arguments ...);
DistributedVector<Complex>  Y(...constructor arguments ...);
```

4.1 Distributions, Alignments, and Processors Ver 1.0+

Distributions can be viewed as abstract coordinate systems that allow us to align different collections with respect to each other. If two elements from two different collections are mapped to the same Distribution point, they will be allocated in the same processor memory. Consequently, if there is a data communication between two collections, it is best to align them so that costly interprocessor communication is minimized.

Unlike Templates in HPFF Fortran, Distributions in pC++ are first class objects.[2] A Distribution is characterized by its number of dimensions, the size in each dimension and the function by which the Distribution is mapped to processors.

[1]Note the pC++ concept of Distribution is, by design, completely identical to the HPF Template. Unfortunately, the word template already has a meaning in C++, so we changed the name to avoid confusion.

[2]Because Distributions are first class objects they can be created at run-time or passed as a parameter to a function. This is very convenient for creating collections at run time, i.e., when creating a new collection the Distribution and align objects can be taken from another collection with which the new collection is to be aligned.

Current distribution functions allowed in pC++ include BLOCK, CYCLIC, and WHOLE. To understand these consider the mapping of a one dimensional Distribution to a set of processor objects. The constructor for a one dimensional distribution takes the form

```
Distribution distribution-object-name(size, &(a processors object),
                                      map-function);
```

For example to make a one dimensional Distribution `Q` of size `n` over a processors object `P` with a `BLOCK` mapping one would write

```
    Distribution Q(n, &P, BLOCK);
```

The mapping function `BLOCK` means that the `n` positions in the distribution are mapped to processor objects by associating contiguous blocks of positions with each processor object so that the load is even. For example, if `P` has 3 threads, then positions 0 through 3 will go to thread 0, positions 4 through 7 to thread 1, and position 8 through 11 to thread 2. In the case that the number of threads does not divide the size of the distribution we use the rules given by the HPF Forum.

Collection are sets of objects that are distributed over processors object threads by a two stage mapping process. One stage is defined by a distribution. The first stage is defined by specifying an alignment object. An alignment object defines the size and shape of a collection and the way in which it is mapped to a distribution.

The syntax for an `Align` constructor is given by

```
  Align  object-name( dimensions-of-collection, mapping-string);
```

For example, a one dimensional collection of size 100 mapped to a one dimensional Distribution of the same size would be defined by the verb

```
  Align OneD( 100, "[ALIGN( dummy[i], myDistribution[i])]");
```

The mapping string is designed to be similar to the HPF ALIGN directive. it is merely a notation to describe a linear affine imbedding of one lattice into another. In other works, we use array notations to say how one array is to align with another. In the example above it is the identity mapping. The mapping

```
  ALIGN( dummy[i], myDistribution[2*i])
```

would map the source array to the even positions of the target. The mapping

```
  ALIGN( dummy[i], myDistribution[3][2*i+1])
```

would map the source array to the odd position of the third row of the target. However the mapping

```
  ALIGN( dummy[2*i], myDistribution[i]) //<<< error
```

is in error because it does not define the odd members of the source. But the mapping

```
ALIGN( dummy[i], myDistribution[i/2])
```

defines a two to one mapping of the source to the target and it is o.k.

In version 1.0 of pC++, one can only define array shaped collections and `Align` is always dimensioned like an array. In future versions, more general `Align` constructors will be made available.

The mapping function is described in terms of a text string which corresponds to the HPFF Fortran alignment directive. It defines a mapping from the domain to a Distribution using dummy domain and dummy index names.

To map a two-dimensional matrix A to a set of processors one can define a two-dimensional Distribution and align the matrix with the Distribution and then map the Distribution to the processors. Suppose we have a 7 by 7 Distribution and the matrix is of a size of 5 by 5, and suppose the Distribution will be mapped over the processor object threads so that an entire row of the Distribution is mapped to an individual processor and the i^{th} row is mapped to processor $i \bmod$ *P.numProcs()*. This mapping scheme corresponds to a CYCLIC map in the Distribution row dimension and a WHOLE map in the Distribution column dimension. The Distribution and distribution can be defined as follows.

```
Processors P;
Distribution myDistribution(7, 7, &P, CYCLIC, WHOLE);
```

The alignment of the matrix to the Distribution is constructed by the declaration

```
Align myAlign(5, 5, "[ALIGN( dummy[i][j], myDistribution[i][j])]" );
DistributedMatrix<Complex>  A(&myDistribution, &myAlign);
```

Notice that the alignment object *myAlign* defines a two-dimensional domain of a size 5×5, and a mapping function.

We may now align the vectors to the same Distribution. The choice of the alignment is best determined by the way the collections are used. For example, suppose we wish to invoke the library function for matrix vector multiply as follows.

```
Y = A*X;
```

While the meaning and computational behavior of this expression is independent of alignment and distribution, we would achieve best performance if we aligned X along with the first row of the matrix A and Y with the first column. (This is because the matrix vector algorithm, which is described in more detail later, broadcasts the operand vector along the columns of the array and then performs a reduction along rows.). The declarations take the form

```
Align XAlign(5, "[ALIGN( X[i], myDistribution[0][i])]");
Align YAlign(5, "[ALIGN( Y[i], myDistribution[i][0])]");
DistributedVector<Complex>  X(&myDistribution, &XAlign);
DistributedVector<Complex>  Y(&myDistribution, &YAlign);
```

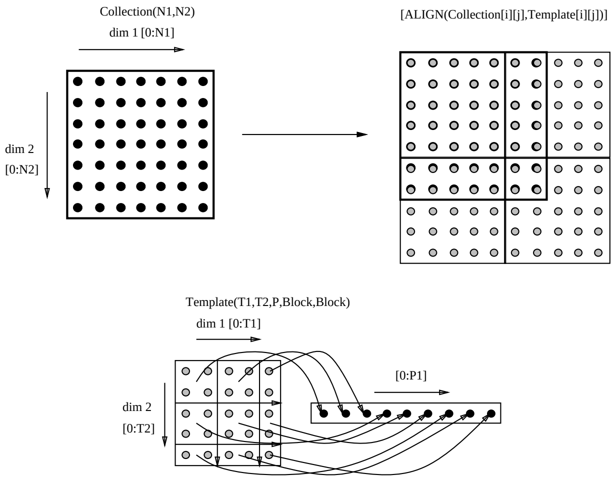

Figure 2: Alignment and Distribution

The alignment and the Distribution form a two stages mapping, as illustrated in Figure 2. The array is mapped into the Distribution by the alignment object and the Distribution definition defines the mapping to the processor set.

Because all of the standard matrix-vector operators are overloaded with their mathematical meanings, pC++ permits expressions like

```
Y = A*X;
Y = Y + X;
```

even though X and Y are not aligned together. We emphasize that the meaning of the computation is independent of the alignment and distribution; the correct data movements will be generated so that the result will be the same.

4.2 Collection as a Template TEClass. Ver 2.0

Template TEClass objects are not supported in pC++ version 1.0. This section is a preview for version 2.0.

The most direct approach to building a collection is to define it as a *template TEClass*, where we supply the type of the collection element as the first template parameter. The template structure can be derived from a parent template TECLASS which will provide the data structures to manage the set of element on each processor thread according to the alignment and distribution specifications. We call this data structure manager collection the SuperKernel. The basic Template TEClass takes the following form.

```
template<Class ElementType> TEClass C: SuperKernel<ElementType>
{
     C(Distribution *D, Align *A, n, m): SuperKernel(D, A) { ... }
     ...
```

```
    int alpha();
};
```

Given an element class E defined by

```
class E{
 public:
   float x, y;
   void f( int i);
   double g();
  };
```

and a distribution D and alignment A, one can create a collection X of type C with element type E with a declaration of the form:

```
   C<E> X(&D, &A);
```

There are a number of different parallel operators that can be applied to the collection X. The simplest is the evaluation of TEClass member functions, such as `alpha()` in the collection C above, on each thread associated with the distribution. Because the collection is, at heart, a TEClass this is nothing new and the expression

```
  X.alpha();
```

behaves exactly as described in the previous chapter.

However, with a collection there is another form of parallelism. Given a member function of element class, `f()`, one can apply the function to every element of the collection with the expression

```
X.f(1);
```

Alternatively, one can index a subset of elements in a collection with the expression

```
   X[i:j:k].f()
```

which will invoke the function `f()` on elements with index in the range from `i` to `j` with step size `k`. (For multi-dimensional collections, the array notation extends in the obvious manner, but this is not reliable in Version 1.0.)

It is also possible to cause an element member function to be invoked on a single element of a collection by using the SuperKernel overloaded operator `(...)`.

4.3 Element Types. Ver 1.0+

There are several restrictions on the classes that are allowed as element types of a collection. We list here some of these restrictions and try to explain why they exist and which restrictions will be lifted.

- An *ElementType* in a collection must not contain pointers to heap based data. The reason for this is that the basic communication mechanism in a pC++ program is to make a copy of an element. Though pC++ maintains a global name space for elements in a collection, there is no global name space other than program global data. In particular, the heap and stack of each processor object thread is for the private and exclusive use by that thread. This is because on distributed memory machines without shared address spaces, pointers cannot be moved across the machine and still have meaning as pointers.

  Future versions of pC++ will have the concept of a global pointer that will help ease the pain of this restriction.

- An *ElementType* in a template TEClass collection can be a basic type like `int`, `float`, etc. It can even be a global pointer to an element in version 2.0. However, the type of an element must be a full C++ class in form of the collection described below. In other words, in version 1.0, if you want a collection of element of type `int` you must declare a class of the form

  ```
  class Int{
    int data;
     // constructor (see next item) and
     // any overloaded integer operators you need
     // would be here.
  };
  ```

  In most cases this restriction is not serious. Elements that are fine grained are discouraged because the overhead of member function calls will far outweigh the advantage of parallel arithmetic.

- In version 1.0 an *ElementType* must have a constructor with no arguments.

  ```
  class Int{
      int data;
     public:
       Int(){ data = 0; }
       ...
  };
  ```

  This constructor is used by the *SuperKernel* collection to allocate the elements. In the future this rule will be relaxed to allow more general constructors so that one may build collections from elements with arbitrary initializations.

4.4 The Structure of a Collection. Ver 1.0+

While the C++ template based collections described in the preceding sections provides a very easy to use form of the collection concept, it does not allow for all the types of operations on aggregates that one frequently needs. For example, there is no way in which a member function of a element class to refer to the structure of the collection as a whole. This missing feature makes it difficult to write programs where one element object can refer to other elements in terms of the topology defined by the collection. To solve this problem there is a more elaborate (and more specialized) form of the collection class that has a special syntax.

```
Collection NameOfCollectionType: ParentCollection{
    private:
      < private data fields and method functions>
    protected:
      < protected data fields and method functions>
    public:
      < other public data fields and method functions>
    MethodOfElement:
      < data and functions that are added to each element>
    };
```

The key points to understand are listed below.

- Collections are derived from a special machine dependent root class called *SuperKernel*. Consequently, every user defined collection is required to be a subclass of SuperKernel or, at least, have the SuperKernel in its ancestry. The `SuperKernel class` uses the distribution and alignment objects of a `collection` to initialize the `collection` and allocate the appropriate element objects on each memory module. The kernel also provides a global name space based on an integer enumeration of the elements of each `collection`. In other words the kernel numbers each element of the `collection` and this number can be used by any element to identify and access any other element. The important element access functions provided by the kernel will be described in greater detail later.
- A `collection` has private, protected and public data and member function fields exactly as any other `class`. However, unlike a standard `class`, when a `collection` object is allocated a copy of the basic `collection` structure is allocated in the local memory of each processor. Consequently, the data fields are not shared in the same way that element object indices are shared. The method functions in a `collection` are invoked in SPMD mode, i.e. they execute in an independent thread on each processor. This is explained in more detail later.
- Because the type of the `collection` element is not specified when the `collection` is defined, the special keyword *ElementType* is used to whenever it is necessary to refer to the type of the element.
- A `collection` also has a set of data fields and member functions that are copied into each element as public fields. These are labeled in the `collection` as *MethodOfElement:* fields. The purpose of `MethodOfElement` fields is to allow each element to "inherit " properties of the `collection` and to define operators that act on the elements with

knowledge of the global structure of `collection`. Fields or method functions within the `MethodOfElement` section that are defined as `virtual` are assumed to be overridden by a definition within the element class.

To illustrate how the MethodOfELement field works, we will show how the pC++ compiler translates a collection into the correct template class. We will start with a simple "hello world" example. There are four parts to every pC++ program and they must be in the following order. First are the element class definitions. We will use a simple class `E` with one member function. Next, one must include the definition of the kernel with an `include` directive as shown below. Third are the definitions of the collections in the program. Last is the main program, which in pC++ version 1.0 is called `Processor_Main()`.

```
#include "pcxx.h"
class E{  // the element
  int e_data;
  void sayHi(){ printf("hello"); }
};

#include "kernel.h"

Collection Set: SuperKernel{
   int i;
   Set(Distribution *T, Align *A): SuperKernel(T,A){};
  MethodOfElement:
    int j;
    virtual void sayHi();
    void hello(){ printf("\nfrom element %d", Ident); }
};

Processor_Main(){
     Processors P;
     Distribution T(4,&P,BLOCK);
     Align A(4,"[ALIGN(V[i],T[i])]");
     Set<E> x(&T, &A);
     x.sayHi();
     x.hello();
}
```

The variable `Ident` is the unique element identifier that is assigned to each element by the SuperKernel. This program will print, up to a permutation of the last 4 lines,

```
hellohellohellohello
from element 0
from element 3
```

```
from element 1
from element 2
```

The collection Set in this example defines two new features that are added to each element. One is the integer field j and the other is the member function hello. Consequently, the user should view the action of the compiler in this case as defining a new class, which we shall call Element which is derived from the users original class E.

```
class Element: E{
  public:
   int Ident; // from SuperKernel MethodOfElement
   Superkernel<Element> *ThisCollection;
   ...        // additional SuperKernel Fields.
   int j;     // from Set MethodOfElement
   void hello();
};
```

This new element class inherits all of the MethodOfElement fields from all the collection in the inheritance chain of the Set collection. A way to view this is that the MethodOfElement fields and member functions form a wrapper around each collection element that allow the element to references collection specific operators and data. The ThisCollection pointer provides a way for the MethodOfElement functions to refer to the TEClass object data that is local to the thread containing the element object as a member of the local collection.

The Set collection is translated by the pC++ compiler to a template class of the form

```
template <class ELementType> TEClass Set: SuperKernel<ElementType>{
    int i;
    Set(Distribution *T, Align *A): SuperKernel(T,A){};
}
```

(Version 1.0 of the compiler does not actually generate an explicit template. Instead, the compiler expand the collection into a new class in a manner similar to the way the C++ compiler expands templates. Version 2.0 will generate a template class as shown above.)

The main program is the same except that the collection object definition is translated to

```
     Set<Element> x(&T, &A);
```

4.5 Collection Constructors. Ver 1.0+

Every collection must have a constructor and each constructor must have at least two standard arguments: a pointer to a distribution and a pointer to an alignment. These two arguments must be passed to the parent collection, which, in tern, passes them to the *SuperKernel* collection. For example, to build a collection constructor that initializes the local TEClass data one could write as follows.

```
Collection C: DistributedArray{
   int z;
   float x;
   public:
      C(Distribution *D, Align *A, int i, float j);
};

C::C(Distribution *D, Align *A, int i, float j): DistributedArray(D, A){
         z = i;
         x = j;
        }

;
```

A BUG in the version 1.0 pC++ compiler will not allow constructors to be defined inline in the Collection body.

Constructors for the ElementType of a collection are restricted as well. See section 4.3.

4.6 Communication between Collection Elements. Ver 1.0+

The elements in a collection have a *global name space*. That is, any processor object may refer to any element of any collection by using the elements integer `Ident` field or by array index notation in the case of collections that are derived from the `DistributedArray` collection class. To illustrate this, let us consider the parallel code that would implement a reduction of all elements of a collection. Over the next four subsections we will look at four ways this might be done. In the first scheme we will do a serial reduction from the main thread. In the second scheme we build a binary reduction tree from the elements and leave the total with element 0. We will make the following assumption. Each element in the collection has an addition operator and an accumulation operator defined as follows.

```
class E{
  public:
   E(){ // a ``zero'' initializer }
   E  operator +(E &);
   E & operator +=(E &);
};
```

The serial solution looks like this.

```
  Processor_Main(){
     Processors P;
     Distribution d( .... );
     Align a( .....)

     C < E >  x(d,a);   // x is the collection
```

```
    E total;
    for(i = 0; i < x.dim1size; i++);
       total += *x(i);  // x(i) returns pointer to element
};
```

The operator (int) returns a pointer to the i^{th} element if the i^{th} element lives in the address space of the executing. Otherwise it returns a pointer to a copy of the i^{th} element. This mechanism will always allow any thread to access public data from any element of any collection. However, to modify an element with a statement like

```
x(i)->y  = 3.14;
```

the element must reside in the address space of the executing thread. (This is because if the element is not local, the pointer `x(i)` will be pointing to a copy and the copy will be modified. In a similar manner, if `f()` is a member function of the element class that side effects the object, then the call

```
x(i)->f();
```

will have no effect unless the element is local. There is a special kernel function that users may use to tell if an element is local to the executing thread:

```
x.Is_Local(i)
```

returns true if the i^{th} element is in the local address space.

There is one very important property of the `x(i)` operator that must always be kept in mind. If an element `i` in a collection `x` is not local to the thread that executes `x(i)`, then `x(i)` points to a copy of the element. This copy should be viewed as a "cached" copy in a one element cache. Consequently, if a thread tries to access two non-local elements then only one may reside in the cache at any time. In other words, `x(j)` is a reference to another element after the reference to `x(i)`, then the second reference will eliminate the first. So, for example, in the code below, the second reference to `x(i)` is an error.

```
ElementType *p;
p = x(i);
printf(" %d", x(j)->Ident);  // x(j) overwrites *p
printf(" %d", p->Ident);     // error: p now points to x(j)
```

There are ways to avoid this problem by using the function `Get_CopyElem()` which is described in section 4.7.

In version 2.0 of pC++ we will change some of these rules. A new data type modifier *global* will be added. This idea comes from the CC++ language. The idea is that any data object that is of *global* type may be referenced by pointers of the form

```
global ElementType *p;
```

In this case expressions of the form `p->y = 3.14` or `p->f()` will cause a message to be sent to the processor object thread that contains the element to perform the given action on the element. Additional information on element communication can be found in section 4.7.

4.6.1 Vector Subsets of a Collection. Ver. 1.0+

Collection parallelism is based on the concurrent application of a class member function to every object in a large set. However, it is not always the case that we want every object to carry out this action. Often we only want to apply the function to a subset of the objects. pC++ allows a mechanism to select a vector subset of elements in a parallel action. We will illustrate the use of this operator below.

To implement a parallel tree reduction the collection will define the addition operator as virtual so that it may use it in the MethodOfElement section.

```
Collection C: SuperKernel{
   public:
      C(Distribution *D, Align *A);
      ElementType reduce1();
   MethodOfElement:
      virtual ElementType &operator +=(ElementType &);
      void sumFrom(int toRight);
};
```

The element function `sumFrom()` access an element a given distance to the right of itself and adds that element to its own value. To access an element at a given offset from the current, i.e. *this*, element, one must have access to the collection structure. This is accomplished through the pointer `ThisCollection` which is added to each element by the SuperKernel. There is a bug in version 1.0 of pC++: ThisCollection can not be used in an inlined MethodOfElement function defined in the body of the collection, so we must define `sumFrom` outside the collection `C` as follows.

```
void C::sumFrom(int toRight){
   ElementType *ptr_neighbor;
   ptr_neighbor = (*ThisCollection)(Ident + toRight);
   (*this) += *ptr_neighbor;
}
```

There is another way to access an element that can be found by an offset from the current position. The SuperKernel function `Self` can be used to achieve the same result by replacing the statement

```
ptr_neighbor = (*ThisCollection)(Ident + toRight);
```

with the equivalent expression

```
ptr_neighbor = Self(toRight);
```

The reduction function will run in each processors object thread and invoke parallel action on the elements. In the first step it will ask all the even indexed elements to fetch the values from the odd elements and add these to their own values. Next it will have all the elements that are multiples of 4 add the sums from the remaining even elements, etc. The result will leave to total in element 0.

```
ElementType C::reduce1(){
        int i;
        int n = dim1size;
        ElementType E;
        for(i = 1; i < n; i = 2*i)
             this[0:n-1:2*i].sumFrom(i);
}
```

Note that we have used a parallel subset operation to refer to the elements of the collections that are multiples of `2*i`. This operator uses a fortran 90 vector syntax

```
    [  base :  limit : stride ]
```

to define the range of elements in the subset. Note also that this operator is applied to a POINTER to the collection.

4.6.2 More on ThisCollection. Ver. 2.0

A standard member of the `MethodOfElement` fields that are added to each element is the `ThisCollection` pointer which allows an element to refer the the collection (TEClass) object for which it is a local collection member. In this section we will illustrate how it can be used to access local collection data members.

The third way to write the summation program is to have each element add its value into a local collection variable. In this case we create an ElementType Object `local_total` (which we assume is initialized to a “zero” value) as a member field in the Collection TEClass object.

```
Collection D: SuperKernel{
   public:
      ElementType local_total;
      D(Distribution *d, Align *A);
      ElementType reduce2();
   MethodOfElement:
      virtual ElementType &operator +=(ElementType &);
      void accumulate();
};
```

The accumulation function adds each element value directly to `local_total`.

```
void D::accumulate(){
   D<ElementType> * ptr;
   ptr = (D<ElementType> *) ThisCollection;
   ptr->local_total += *this;
}
```

Note that because *ThisCollection* is inherited from the SuperKernel, we must cast it to the type of our collection before we can access the `local_total` field.

The reduction function which runs in each TEClass processors/object thread will invoke the accumulate function on each element. It is important to understand that, within a processors object thread, this action is completely serial, so there is no "race condition" involved with the update to `local_total`. We use a simple TEClass reduction function to compute the total between threads as shown below.

```
ElementType D::reduce2(){
     this->accumulate();
     return pCxx_sum(&local_total);
}
```

NOTE: for Version 1.0, the `pCxx_sum()` function is not yet part of the standard library. However, everything else described in this subsection will work in version 1.0.

4.6.3 Working with the Local Collection. Ver. 1.0+

Each Processors object thread owns one TEClass representative object from the collection definition. The non-MethodOfElement member function are executed by the thread in pure MIMD style. Each such function has complete control of the "local collection" of elements that are mapped to that thread of computation. It is often the case that the easiest way to carry out some type of processing on the collection is to have each thread carry out the task sequentially on each element of its local collection.

The forth way to program the summation of the elements in a collection is to have each processor object thread compute the total of the elements in their own local collection and then we will copy the result to a second collection defined as a distributed array of elements with only one element per thread.

```
   Processors P;
   Distribution d(NumProcs(),&P,BLOCK);
   Align a(NumProcs(),"[ALIGN(V[i],T[i])]");
   DistributedArray< C > onePerThread(&d,&a);
```

One of the functions of a distributed array is to provide basic array oriented reduction function. In this case we will use the function

```
     void ReduceDim1();
```

which reduces along the first dimension so that the sums of rows of an array are reduced to column 0. In our case, because the array is one dimensional, the sum will be left in element 0.

We begin with a rewrite of the reduce2() to take a pointer to the new distributed array as an argument. We will have each thread add every element of the local collection to the total. The easiest way to say this is to use the function Is_Local() which is true if the named element is in the local collection of the thread that is executing the function. A simple loop can be used to sweep though the collection as follows.

```
    for(int i = 0; i < dim1size; i++)
       if( Is_Local(i))  local_total += (*this)(i);
```

The problem with this approach is that one must go through the entire collection to identify only the local subset. In some cases the overhead for this search is not serious. However, there is another way to do this. One may set a special buffer to the head of the local collection with a call to the ResetLocal() which takes a pointer to the collection as an argument. Then the function FirstLocal() will extract the index of the first element in the local collection and the function Local() can be used to advance the index to the next local element index. When Local() returns a negative value, we have exhausted the set of local elements. Using these functions our code can be written as

```
ElementType D::reduce3(DistributedArray<ElementType> *onePerThread){
    int loc;
    ResetLocal(this);
    for(int i = FirstLocal(); i >= 0; i = Local(i))
          local_total += (*this)(i);

    // now copy the total to the collection onePerThread and Reduce
    // that one.
    ResetLocal(onePerThread);
    if((i = FirstLocal()) >= 0)
                 (*onePerThread)(i) = local_total;
    Barrier(); // make sure all the copies are complete.
    onePerThread->ReduceDim1();
    return *(*onePerThread(0));
}
```

We will include one more example of using the local collection. Suppose you wish to broadcast one value stored in a member field of one element to the same location in all the other element of a collection. By adding the function BroadcastToElements() to our collection thread functions as shown below only need to supply the index of the element that has the data to be broadcast, the offset in bytes from the start of the element and the size of the data block.

```
Collection D: SuperKernel{
   public:
      ElementType local_total;
      D(Distribution *d, Align *A);
     void BroadcastToElements(int element, int offset, int size);
   MethodOfElement:
     ...
};
```

To implement this we build a buffer in each thread and first find the thread that has the source element. Next we copy the source data to the buffer and share the contents of the buffer with the other elements by using the pCxx_BroadcastBytes() routine. Once each thread has a copy, it can load the data into the appropriate filed of each element.

```
void D:BroadcastToElements(int element, int offset, int size){
      char *buffer = new char[size];
      int flag = 1;
      int i, j;
      char *p;
      if(Is_Local(element)){
        flag = 0;
        p = ((char *) (*this)(element)) + offset;
        for(j = 0; j < size; j++) buffer[j] = p[j];
      }
      pCxx_BroadcastBytes(flag, size, buffer);
      for(i = 0; i < dim1size; i++}
           if(Is_Local(i)){
           p = ((char *) (*this)(i)) + offset;
           for(j = 0; j < size; j++) p[j] = buffer[j];
      }
      delete [] buffer;
}
```

Note that the use of an offset into an element is not a standard C++ way of writing code. A more elegant approach would be to use a pointer to a member. However, the current pC++ preprocessor has a problem with this construct.

In the next section we discuss making accessing sub-blocks of element in more detail.

4.7 Accessing a Remote Collection Element. Ver 1.0+

The operator (...) is overloaded for collections so that one may access individual elements. If this element is local to the executing thread, then the (...) returns a pointer to the local object. If the object is not local, then (...) returns a pointer to a cache buffer containing a copy of the object. The problem with the cache buffer is that it may be overwritten. If it is necessary to keep a copy of an element for a long period, or if a thread needs copies of several elements that must remain local together, then a better function to use is `Get_CopyElem()`.

If the i^{th} element of a collection is local `Get_CopyElem(i)` returns a pointer to the element and no copy is made. Otherwise, `Get_CopyElem()` allocates space on the local heap and makes a copy of the element there. That copy will persist until it is deleted by the users. A good example is in the following `MethodOfElement` code that computes the back-solve step in a cyclic reduction operation on a vector of elements for which operators $+$, *, / and $=$ have all been overloaded. That is we solve the equation

```
(*b)*X(i-s)  +  (*a)*Xnew(i)  +  (*b)*X(i+s)  =   Xold(i)
```

for `Xnew(i)` which overwrites `Xold(i)` in `*this`.

```
void DistributedTridiagonal::backSolve(int s){
    int k;
    ElementType *v1,*v2;
    v1 = (ElementType *) ThisCollection->Get_CopyElem(index1 -s);
    v2 = (ElementType *) ThisCollection->Get_CopyElem(index1 +s);

    *this = (*this - (*b)*( *v1 + *v2 ))/(*a);

     if (!ThisCollection->Is_Local(index1 -s))  delete v1;
     if (!ThisCollection->Is_Local(index1 +s))  delete v2;
}
```

Another frequent situation is when we only wish to look at part of an element. For example, if we have an element structure that contains one or more large arrays or sub-grids of a larger structure.

```
class BigElement{
  public:
     double x[100][100];
    ...
};
```

In many situations we may only wish to access part of this structure, for example, the first row of the array filed `x`. In this case, it would be very costly to cache a complete copy of the element. There are special functions that allow us to program a thread to fetch only the part of the element that we need. The function `Get_ElementPart()` takes three parameters. The first is the index of the element, the second is the offset of the of the data field measured in `sizeof(int)` word boundaries from the beginning of the element. The third parameter is the size in terms of multiples of `sizeof(int)`. Note that the formulation does not allow for data fields that are character aligned. We will fix this in version 2.0.

As an example of how to fetch a single row of the field x from a remote element, the MethodOfElement member function below uses the this pointer to establish the correct offset.

```
#define IntOffset(p, x) ( ((int *) x) - ((int *) p))

void Grid::grabNeighborXrow(int index, int row){
  double *tmp;
  double mybuff[100];
  int size, off;
  off = IntOffset(this, &(this->x[row][0]));
  size = (sizeof(double)*100)/sizeof(int);
  tmp = (double *) ThisCollection->Get_ElementPart(index, off, size);
  for(int i = 0; i < N; i++) mybuff[i] = tmp[i];
}
```

In our future version, a user will not need to use Get_ElementPart to access part of an element. A user may simply write

```
 tmp = (double *) ThisCollection->(index)->x[row][0:99];
```

to get an entire row of a remote element.

It should also be noted that we have only used one dimensional collections in the examples in this sections. There are also versions of Get_ElementPart, Get_CopyElem() and (...) for two dimensional distributed array and higher dimensional structures will be allowed as soon as we have time to implement them.

4.8 Parallelism with Collection: Summary. Ver 1.0+

This section is mean to summarize the ways in which parallelism can be achieved in collection operations. We shall describe these in terms of a simple example.

4.8.1 Method of Element Parallelism

Let E be a class as shown below.

```
class E{
  public:
     float a, b, c;
     void f(int);
     float g(float h);
};
```

Let C be a collection where these same fields declared as virtual members in the collection.

```
Collection C: ... {
    ....
  MethodOfElement:
     virtual float a, b, c;
     virtual void f(int);
     virtual float g(float h);
};
```

In the case of MethodOfElement fields, the keyword "virtual" means that these will be supplied by the element class. We apologize for the slight twist of the usual C++ meaning. Note: In version 1.0 the declaration of these fields as virtual in the MethodOfElement section of the collection is required. However, this will not be required in version 2.0.

Suppose one has a collection defined in terms of these components and a pointer to a collection of the same type as shown below.

```
C < E >  X( .... );
C < E >  *p = X;
```

The simplest form of parallelism is to invoke a member function on each element of the collection as in in the expressions below.

```
X.f(3);
p->f(3);
```

These operations may be called from the main thread of computation or from the Processors object threads provided that each processors object representative makes the same call.

Parallel operations on element fields are also allowed as long as the operation is well defined on the elements. For example

```
X.a =  X.b + X.c - X.g(32.1)
```

The general rule is that if `X` is a collection of type `C` with distribution `d` and alignment `a` and if `y` is a member field of type `T`, then the expression `X` is of type `C < T >` with the same distribution and alignment.

NOTE: There is a bug in version 1.0. The expression `X.g(X.a)` is valid and means that the `a` field of the i^{th} element is passed to the i^{th} instance of the invocation of `g()`. However, the compiler currently rejects this construct. This will be fixed as soon as possible.

Member functions can be applied to vector subsets of a collection as shown below.

```
p[ i : j : k].f();
p[ i : j : k].a  = p[r : s : t].b + p[u: v: w].g(33.3);
```

Observe that a pointer to a collection is necessary to parse the expression. also, note that the extent and bounds of the operation are determined by the `[...]` operation on the left hand side of the assignment.

In version 1.0 it is not possible to have more than 1 dimensional vector sub-ranges. This problem will be addressed in version 2.0.

4.8.2 Thread Parallelism

Collection member functions that are not of type MethodOfElement are executed by the threads of the Processors Object in the same way that TEClass member functions are executed. That is, they are executed in SPMD/MIMD mode. For example, let the collection C structure take the form shown below.

```
Collection C: ... {
    int i, j;
  public:
    void h(int k){ i = k; j = MyProc()}
    ....
  MethodOfElement:
     virtual void f(int);
};
```

Let a collection X be be declared as follows.

```
    Processors P;
    Distribution d(..., &P, ...);
    Align a(....)
    C < E >  X(&a, &d, ...);
```

Each thread defined by the Processors object P will own a representative of the base collection TEClass defined by the non-MethodOfElement fields of C Then, if we invoke the operator

```
   X.h(3);
```

each thread will execute h(3) locally and modify its private copies of the variables i and j. Access to the local collection of element on each thread is described in the preceding sections.

4.9 The Organization of a pC++ Program. Ver 1.0+

Every pC++ program has a very special structure. The first rule, which is valid for version 1.0, is all pC++ constructs must be contained in one file. In later versions we hope to have mechanisms that allow us to have pC++ constructs spread across multiple source files.

The second rule for pC++ programs is that the following special order must be preserved for a pC++ file.

```
#include "pCxx.h"

// any other include files you may need would go here.
// defined constants and standard C++ global variables can go anywhere.
#define myPi = 22.0/7.0
```

```
int my_x = myPi;

// Next all of the definitions required for collection elements are included
// This includes all source code for element method functions
// as well as the text of the element class definitions.

#include "myElementClass.h"

// next include the collection definitions starting with
// the kernel and, if needed the distributed array collection definitions.

#include "kernel.h"
#include "distarray.h"

// next are the definitions and functions required for your collection
// defintions.

#include "myCollectionClasses.h"

//the main program

Processor_Main(){
  ....
}
```

4.9.1 I/O in pC++ programs. ver 1.0

Version 1.0 has a very weak I/O system. There reason for this. In order to maintain portability, we have restricted I/O in version 1.0 because there are, as yet, no standard mechanisms to address parallel file system calls on all the platforms we wish to support. We are working on defining a standard parallel extension to the streams package, but that will take a while.

For Version 1.0, output to `stdout` can go anywhere but input from `stdin` and any other file I/O is restricted to the main thread.

5 Programming with Collections. Ver 1.0+

In this section we include two simple examples of programming with pC++.

5.1 Example: Matrix-Matrix Multiply. Ver 1.0+

Let us fist revisit the matrix-matrix multiply example discussed in the SPMD programming section. We will build our first version out of matrices which are defined as distributed arrays of a simple class to represent a single matrix element.

```
class Double{
 public:
  double value;
  Double(){ value = 0; }
  Double(double x){ value = x;}
  Double &operator *(Double &x){ return Double(value*x.value); }
  Double &operator +=(Double &x){ value += x.value; return *this;}
};
```

The matrix collection, shown below, builds a matrix sized according to the dimension in the alignment object. The idea behind the matrix-matrix algorithm in computing

```
P = M * N
```

is to have each element of the matrix `P` grab the right row of `M` and the right column of `N` and form the dot product to obtain its value. Consequently, the operation is performed using "method of element" parallelism

```
Collection Matrix: DistributedArray{
    // a constructor is needed.
    Matrix(Distribution *T, Align *A): DistributedArray(T, A){}

  MethodOfElement:
    void matMul(Matrix<ElementType> &M, Matrix<ElementType> &N){
       int myrow = this->index1;  // first index from DistributedArray
       int mycol = this->index2;  // second index from DistributedArray
       int maxcol = M.dim2size;   // should be your "n" parameter
       if(maxcol != N.dim1size) err("col dim of M must = row dim of N");
       value = 0;
       for (int j; j < maxCol; j++)
              *this += M(myrow, j) * N(j, mycol); // Get_Element implicit
       }
```

The complexity of this code should be compared with the message passing `matMul()` in the SPMD programming (section 3.2.4). Note that these two programs implement the same basic algorithm, the version above is much simpler. In addition, the message passing version was tied to the way the data was distributed to memory. In this version the distribution and alignment of the data are completely independent of the the code. In the main program below we have used a distribution and alignment scheme that places the rows of the second matrix along the columns of the first. Consequently, no communication takes place if the matrices are square. However, we emphasize that the code will run no matter how the data is aligned.

```
Processor_Main()
{
        Processor P;
        Distribution T(m, p, &P, BLOCK, BLOCK);
        Align A(m, n, "[ALIGN( domain[i][j], T[i][j])]" );
        Align B(n, p, "[ALIGN( domain[i][j], T[j][i])]" );
        Matrix<Double> M(&T, &A, m, n);
        Matrix<Double> N(&T, &B, n, p);
        Matrix<Double> P(&T, &A, m, p);

        P.matMul( M, N);
}
```

One problem with this program is that it would run like a dog. There are too many function calls for each multiply-add. Here is how to make it fly. Let's replace Double as the element type with

```
#define N 100
class BlockDouble{
   double data[N][N];
   BlockDouble &  operator *(BlockDouble &x){
         call a linpack BLAS-3 dense matrix-matrix multiply
         tuned for the architecture
         }
   BlockDouble & operator +=(BlockDouble &x){
         call the BLAS-3 dense matrix accumulate.
         }
};
```

Now by replacing "Double" by "BlockDouble" in the `Processor_Main()` and you have a serious blocked matrix-matrix multiply that will do $O(N^3)$ flops for $O(N^2)$ data moves. This can be very fast.

5.2 Example: Batcher's Bitonic Merge Sort. Ver 1.0+

Another classic example is the Batcher Bitonic Merge Sorting algorithm. A bitonic sequence is a string of values that can be seen as the concatenation of an increasing sequence followed by a decreasing sequence. The Botonic merge operates by dividing a bitonic sequence of length k into two sequences of length $k/2$. The item at position i is compared to the item at position $i+k/2$ and the smaller is kept at position i and the large is kept at position $i+k/2$. The results is two bitonic sequences, each of length $k/2$, with the property that every element of the left hand sequence is less than every element of the right hand sequence. (See Knuth for a proof.) We then recursively merge the two smaller sequences and we will eventually see the entire sequence sorted. But to make this work you need a bitonic sequence to start with. The way you make this happen is to start with sorted sequences of lenght r with one increasing and the other decreasing. When concatenated, they form a bitonic sequence of length $2r$. Doing this with $r=1$, $r=2$, $r=4$ and so on you can build up larger bitonic sequences.

The code below works with collections of element where each element is a sub-sequence of smaller objects. The subsequences will be called *Pvector* objects, which as shown below, contain a vector of simple objects with an integer key.

```
class E{
  public:
   int key;
};

class Pvector{
 public:
   E part[BLKSIZE];
   int sortdir;
   void Qsort(int dir);
};
```

The Pvector class contains a member function `Qsort` which is used to sort the local vector of object in the direction indicated by an integer flag. There is also a field `sordir` which is used to record the last direction the list was sorted. `Qsort` is implemented with the standard C library quicksort. (See the TestSuite directory for the complete source code.)

The collection of Pvector objects is based on the DistributedArray collection library as snown below.

```
#include "kernel.h"
#include "distarray.h"

Collection SortedList: DistributedArray{
  public:
   SortedList(Distribution *D, Align *A);
```

```
   void merge(int distance);
   void sort();
  MethodOfElement:
   E tmp[BLKSIZE];
   virtual E part[BLKSIZE];
   int savedir;
   virtual void Qsort(int dir);
   int cond(int a, int b, int v1, int v2);
   void localSort(int flg);
   void grabFrom(int dist);
   void localMerge(int dist, int flg);
   void localCheck();
   // needed for dist. array.  not used here.
   double Dot(ElementType &arg1){ return 0.0; }
   ElementType &operator +=(ElementType &rhs){ return *this; }
};
```

Within the TEClass thread functions we have the two main components of the sorting algorithm `sort()` and `merge()` which accomplish the bitonic merge and bitonic sort. Sort is the toplevel function. Its main job is to set up a sequence of bitonic merges until the entire list is merged into one sorted list. This function begins by calling an element function `localSort()` which calls `Qsort` on each element. Local sort is designed so that if it is called with its parameter equal to 1, then it does the initial sort so that even indexed element are sorted in increasing order and odd numbers elements are sorted in decreasing order. This means that we have established bitonic sequences consisting of pairs of elements consisting of one even element and the following odd numbered element.

```
void SortedList::localSort(int flg){
 if(flg)
  Qsort(Ident % 2, 0, BLKSIZE-1);
 else
  Qsort(!savedir, 0, BLKSIZE-1);
}
```

The sort function is then very simple.

```
void SortedList::sort(){
   int i, p;
   this->localSort(1);
   for(i = 1; i < dim1size; i = 2*i)  merge(i);
   Barrier();
};
```

The `merge(i)` function is as described above. It is designed to merge bitonic sequences of length *2*i* performing the exchange operation of distance i and then recursively merging the resulting subsequences.

```
void SortedList::merge(int dist){
   int flg;
   flg = 1;
   while(dist > 0){
        this->grabFrom(dist);
        this->localMerge(dist, flg);
        dist = dist/2;
        flg = 0;
       };
   this->localSort(0);
}
```

There are two parallel element functions invocations called in `merge()`. The first of this is a function `grabFrom(dist)` which, for each element, extracts a copy of the element a distance `dist` away into a local buffer `tmp`. Note that this copy of the data to a second array disqualifies this implementation from being a classical "in place" sort. We will return to this issue later. This is the function that does all the inter-element communication. Note that depending on which stage one is in the algorithm the sign of the offset alternates. That is, in the first stage even elements access the element to the right and odd elements access the element to the left. In the second stage, elements 0 and 1 access elements 2 and 3 respectively and visa versa.

```
void SortedList::grabFrom(int dist){
   int v,i, offset;
   E *T;
   void * addr1, *addr2;
   v = (Ident/dist) % 2;
   if(v ) offset = -dist;
   else   offset = dist;
   T = &((*ThisCollection)(Ident+offset)->part[0]);
   for(i = 0; i < BLKSIZE; i++) tmp[i] = T[i];
}
```

The `localMerge()` algorithm selects the values from the `tmp` array to keep in the local array. The most complex part of the Bitonic merge algorithm is keeping track of where one should generate an ascending sequence and where one must generate a descending sequence. During the initial ascending merge operations, the order must reverse at each step. While, for each recursive "sub-merge" the sordirection must be the same as on the previous step. The local field `savedir` records the previous direction of the merge and the parameter `flg` in the local merge routine below is used to distinguish the initial merge for a give value of `dist`.

```
void SortedList::localMerge(int dist, int flg){
   int v, v2, i;
   if(flg) savedir = v2 = (Ident/(2*dist)) % 2;
```

```
    else v2 = savedir;
    v = (Ident/dist) % 2;
    for(i = 0; i < BLKSIZE; i++){
        if(v == v2) && (part[i].key > tmp[i].key)) part[i] = tmp[i];
        if(v != v2) && (part[i].key <= tmp[i].key)) part[i] = tmp[i];
        }
};
```

The main program for this example is given below.

```
void Processor_Main()
{
  Processors P;
  Distribution T(SIZE,&P,BLOCK);
  Align A(SIZE,"[ALIGN(V[i],T[i])]");
  SortedList<Pvector> X(&T,&A);
  int i, n, error;
  double temps;
  n = SIZE*BLKSIZE;
  X.sort();
}
```

This program will sort 10^6 items in about 2.5 seconds on a 256 node CM-5 or Paragon. The standard unix C library qsort on a sparc 10 will complete the task in 25 seconds. Running the direct qsort on a single node of the CM-5 requires 104 seconds. Conseqently we see a speed-up of only about 40. This is cleary not the best that can be done. Part of the problem is the complexity of the Batcher Bitonic Sort ($O(n * log^2(n))$ vs. $O(n * log(n)$.

5.3 C++ and HPF Style Fortran. Ver 2.0

One of the principal roles played by C++ in programming massively parallel computers is in the design of distributed data structures that are either too complex or simply inconvenient to program in Fortran. However, the mainstay of scientific and engineering programming is Fortran. The reasons for this are more than cultural. The restrictions in the Fortran type-system prohibit the use of pointers beyond the simple alias mechanisms allowed by Fortran90. Consequently, Fortran compilers can perform aggressive data-dependence-based optimizations which are not possible in C++. In addition, code generation technology and optimized intrinsic library functions present in Fortran are much better than their C++ counterparts. With the present generation of Fortran90 compilers, vector and array operations will be easy to compile into efficient code. We believe that it is essential to support a portable programming environment that mixes Fortran with C++ across the entire range of supercomputing platforms. [3]

[3] This point of view is also supported in the research community. The soon to be published conclusions of the 1993 NSF/ARPA/DOE Grand Challenge Meeting include several strong recommendations for more

There is no fundamental problem in linking Fortran to C++ programs. C++ has a standard facility for representing interfaces to other languages. However, with the founding of the High Performance Fortran Forum and the publication of the High Performance Fortran (HPF) extensions to Fortran90, we have a new problem. In the following sections, we explore and propose a mechanism for linking HPF programs to parallel C++ programs.

5.4 Interfacing HPF and pC++

Let us consider the ways in which this interface might operate:

- HPF calling C++/pC++:
  - *a.* A direct call to a C++ function from the main thread of a Fortran program. This presents no real problems that did not exist before, unless an HPF distributed array is passed as an argument.
  - *b.* A C++ function is called as an HPF extrinsic function in a parallelized loop. Again this is only interesting if an HPF distributed array is passed as an argument.

  The HPF extrinsic procedure interface which allows calls to non-HPF subprograms as extrinsic procedures may well suffice for the above two situations provided the C++ subprogram is designed to be executed on a single processor. In both cases we assume that the C++ function will not attempt to return a value of any type other than a base type like `int` or `double`. It may be possible to view a returned C++ structure as a Fortran90 instance of a user defined type, but returning a C pointer would be a big problem.

- pC++ calling HPF:
  - *a.* A call to a HPF parallel subroutine from a pC++ sequential main thread, *i.e.*, a pC++ main program.
  - *b.* A call to a Fortran "node program", *i.e.*, a sequential or vector subroutine that is called from a parallel section of a pC++ program.

Of these we will focus on designing an interface for the case of pC++ calling Fortran. The data structure manipulation and encapsulation properties of C++ and the fast computing speed of Fortran make it natural for Fortran subroutines to be called as library routines and for C++ to act as the main controller.

Let us first consider case *a.* A pC++ element class which contains three arrays with two of the arrays being a special type `FArrayDouble` and one being a conventional C++ array is given below.

```
class MyElement {
  public:
   FArrayDouble x, y;
   double z[100];
```

investigation of mixing object-oriented style with the more traditional programming styles.

```
   MyElement();
};
```

The special type FArrayDouble is used to contain double precision Fortran90 arrays and their descriptor structures. To define the constructor for the class we assume that x is a one dimensional array of a size of *100* and y is a two dimensional array of a size of 200×50.

```
MyElement::MyElement():x(100),y(200,50)
{...}
```

To use this class in a pC++ program that calls an HPF subroutine, we need to define the Processors object and to specify the size and shape of the collection. In the following example, a one dimensional collection of a size of *64* with elements of type MyElement is constructed and an HPF subroutine, FFUN, is called.

```
extern "HPF" void FFUN(FArrayDouble&,
               FArrayDouble&,double*);
main(){
 Processors P(64);
 Distribution D(64,&P,BLOCK);
 Align A(64,"[ALIGN(T[i],D[i])]");
 HPFCollection<MyElement> C(&D,&A);
 FFUN(C.x,C.y,C.z);
  ...
}
```

The important point to consider is what the passed arguments look like in terms of HPF distributed arrays. In this case, they can be defined by the HPF directives.

```
!HPF$ PROCESSORS P(64)
      double precision x(100*64),
     &  y(200*64,50),z(100*64)
!HPF$ DISTRIBUTE x(BLOCK),y(BLOCK,*) ONTO P
!HPF$ DISTRIBUTE z(BLOCK) ONTO P
```

It is relatively easy to see how the blocked decomposition corresponds to our simple collection of array blocks. When we have a two dimensional array of virtual processors, we would have a pC++ program as below:

```
main(){
 Processors P(64,32);
 Distribution D(64,32,&P,BLOCK,BLOCK);
 Align A(64,32,"[ALIGN(T[i][j],D[i][j])]");
 HPFCollection<MyElement> C(&D,&A);
 FFUN(C.x,C.y,C.z);
  ...
}
```

In this case, the HPF function would see these structures as defined by the following sequence of directives.

```
!HPF$ PROCESSORS P(64,32)
      double precision x(100*64,32),
     &  y(200*64, 50*32),z(100*64,32)
!HPF$ DISTRIBUTE x(BLOCK,BLOCK) ONTO P
!HPF$ DISTRIBUTE y(BLOCK,BLOCK) ONTO P
!HPF$ DISTRIBUTE z(BLOCK,BLOCK) ONTO P
```

The reader will note that we have not described how to create arrays that are distributed with anything other than a BLOCK distribution and array sizes are restricted to be multiples of the processor array size. There are ways around these restrictions, but they will not be discussed here.

5.5 Working with Connection Machines node Fortran on the CM-5

In this section we consider case *b* in which a Fortran node program is called from a parallel section of pC++. One of the problems in experimenting with HPF compilers is that only a few are now beginning to emerge. At the time of this writing, we do not have access to any of these systems, so we have conducted our experiments with Thinking Machines Co. CM Fortran (CMF) on the CM-5. CMF is a reasonable approximation to the spirit of HPF.

There are three major issues that we want to address:

- Fortran array memory allocation,
- Fortran array element access, and
- Fortran array interprocessor communication.

As it can be seen later on in this section, the first two are addressed through new C++ classes, and the third through a new pC++ collection.

A CM-5 processing node consists of a Sparc microprocessor and four vector processor accelerators (vector units). To make use of these vector units, arrays have to be allocated in the memories of the vector units. The Connection Machine Run Time System (CMRTS) provides functionalities for allocating space on the vector unit memories. However, the allocation is not so trivial since one also needs to be concerned about how array elements will be partitioned among the four vector units and being compatible with CMF. We use CMRTS function "CMRT_intern_detailed_geometry" to create a geometry and "CMRT_allocate_heap_array" to allocate memory. Once the arrays are allocated, array descriptors returned by "CMRT_allocate_heap_array" are passed to Fortran subroutines as arguments. Computations are done in Fortran subroutines. These Fortran arrays can also be directly manipulated by pC++ control programs through overloaded C++ operators.

We design a special C++ array class for each Fortran data type. The following class defines an array class for Fortran double precision arrays:

```
class FArrayDouble {
  // private part of class
 public:
  double *d;
  FArrayDouble() {d = 0.0;};
  FArrayDouble(int i);
  FArrayDouble(int i,int j);
  FArrayDouble(int i,int j,int k);
  FArrayDouble(int i,char layout1);
  FArrayDouble(int i,int j,char layout1,
   char layout2);
  FArrayDouble(int i,int j,int k,
   char layout1,char layout2,char layout3);
  ~FArrayDouble();
  double& operator()(int l);
  double& operator()(int l,int n);
  double& operator()(int l,int n,int m);
};
```

The private part of the class is used to store memory addresses of an array on the four vector units. The information will be used for accessing each individual array elements in pC++, and, in general, should not be of concern to the users. The class constructors that require arguments allocate memory on vector units when invoked. The arguments, i, j, and k specify the dimension sizes of the array and `layout1`, `layout2`, and `layout3` correspond to the CMF compiler directive for array layouts. A layout can be either "SERIAL" or "NEWS." A "NEWS" dimension means that this dimension will be distributed across the four vector units. A "SERIAL" dimension means this dimension will be packed into one vector unit's memory. An array with only "SERIAL" dimensions signals that it should be allocated in the Sparc chip's memory and thus is not allowed in the current implementation. Such an array can be allocated just like a normal C++ array which is always allocated in the Sparc chip's memory. The constructors that do not require the layout information use the default "NEWS" layout. The overloaded operators are used for accessing array elements.

The method we use to access array elements on the vector units from the Sparc chip differs from that of CMF. In the CMF assembly code, array element access involves computing "send addresses." In our approach, we use "subgrid_dimension," "offchip_position," "subgrid_axis_increment," and "subgrid_size" provided by CMRTS to access each individual array element. It turns out that our method is as fast as CMF when all axes are declared "NEWS" and can be a few times faster than CMF when some axes are declared "SERIAL." For example, the speed for accessing a 64×64 double precision array is 5.68×10^5 bytes/second in pC++ and 3.81×10^5 bytes/second in CMF when the first dimension is declared "NEWS" and the second "SERIAL."

Besides memory allocation, we also provide methods for inter-element communication for the Fortran arrays. We design a special pC++ collection for this purpose:

```
Collection Fortran : public SuperKernel {
 public:
  Fortran(Distribution *D,Align *A);
  void GetFArray(int index,int offset,
                 FArrayDouble &buffer);
  void GetFArray(int index1,int index2,
      int offset,FArrayDouble &buffer);
};
```

The `Fortran` collection is derived from the special root collection `SuperKernel`. The communication function `GetFArray` fetches an `FArray` array from a collection element denoted by the given index(es).

A Fortran subroutines is declared as a method of an element class. An invocation of a Fortran subroutine declared this way results in separate invocations of the Fortran subroutine on each processor. The Fortran subroutine is thus similar to a message-passing CMF "node program" though explicit message-passing is strongly discouraged in the Fortran subroutine. Communication between collection elements is accomplished in the pC++ control program. Since Fortran compilers usually add under scores to subroutine names, the pC++ compiler will need to generate a "wrapper function" for each Fortran subroutine.

By way of illustration, the following shows how a Fortran subroutine is called from pC++ utilizing the Fortran interface.

```
extern "C" {
 //Fortran subroutine is declared external
 void integrate_seg_(double*,double*,
                        int&,double&);
}

class Segment {
 public:
  double seg_sum;
  int length;
  FArrayDouble x, y;
  Segment();
  void integrate_seg();
};

Segment::Segment():x(length),y(length) {
  //FArrays are allocated in constructor
}

void Segment::integrate_seg() {
  // wrapper invokes Fortran subroutine
```

```
  integrate_seg_(x.d,y.d,length,seg_sum);
}

void main() {
  Processors P;
  Distribution D(64,&P,CYCLIC);
  Align A(64,"[ALIGN(T[i],D[i])]");
  Fortran&lt;Segment&gt; F(&D,&A);
  F.integrate_seg();
}
```

The Fortran subroutine is given below:

```
      subroutine integrate_seg(x,y,
     &   length,seg_sum)
      double precision seg_sum
      integer length
      double precision,array(length)::x,y
cmf   layout x(:news), y(:news)
      seg_sum = SUM(x*y)
      return
      end
```

In this example, the Fortran subroutine computes the dot-product between two one-dimensional arrays x and y. The x and y arrays are passed to the Fortran subroutine as "explicit-shape" arrays. However, the interface does not prevent us from declaring the arrays as "assumed-shape" arrays.

One shortcoming of the current Fortran interface implementation is that the pC++ control program cannot access Fortran COMMON blocks. This requires that all Fortran global variables be declared in the pC++ control program and passed to Fortran subroutines as arguments. Nevertheless, it does allow communication between Fortran subroutines through COMMON blocks. It is important to note, however, that pC++ programming model allows more than one pC++ collection element to be allocated on one processor, meaning Fortran subroutines called by different collection elements will access the same COMMON block. Data in the COMMON block are shared by elements on that processor. Potential race-condition may arise if a programmer is not careful. The best way to solve this problem is to allocate only one collection element on one processor.

At the time of this writing, the pC++ compiler cannot generate the wrapper for the Fortran subroutine as well as the extern statement, so a user has to hand code them. In the future, a user would write something like

```
extern "HPF" void Segment::integrate_seg(FArrayDouble &x,FArrayDouble &y,
                                         int &length,double &seg_sum);
```

to indicate that integrate_seg is a Fortran subroutine and the compiler will generate the necessary code.

5.6 Working with Intel node Fortran on the Paragon

We have also ported the Fortran interface to the Intel Paragon. Unfortunately, Fortran90 type compiler is not available on the Paragon, so we can only experiment with the Intel Fortran77 compiler. Since the Fortran interface is designed to be portable, most of what we have discussed for the CM-5 still applies for the Paragon. The pC++ program in the previous section can be directly ported to the Paragon without any change except that the layout specifications are ignored on the Paragon. FArrays become normal C++ arrays since there is no vector unit attached to the i860 processor. Nevertheless, the overloaded operators make array element access conceptually easier. The Fortran90 style subroutine has to be changed with vector expressions being converted to do-loops.

6 The Demo Examples. Ver 1.0+

The standard distribution of pC++ comes with a series of Demo "Benchmark" examples in the subdirectory TestSuite that have been run on a variety of machines. In the sections below we discuss a number of these codes.

6.1 BM-1: Block Grid CG

The first benchmark illustrate a "typical toy problem", grid computation. The computation consists of solving the Poisson equation on a two dimensional grid using finite difference operators and a simple conjugate gradient method without preconditioning. Though this problem is very simple, it does illustrate important properties of the runtime system associated with an important class of computations.

In the program we have used an early prototype of our *DistributedGrid* collection class. In addition we have also used a common blocking technique that is often used to increase the computational granularity. This involves placing a square sub-grid on each processor. Given a problem of size N^2 and P^2 processors with $P¡N$, there are two ways to do this. The most natural is to define the grid as being of size N by N and mapping it to a P by P processor array by a BLOCK BLOCK distribution through the template. Unfortunately our compiler is not yet powerful enough to do the optimization required to eliminate the element-to-element access functions on the local sub-grids that will yield vectorizable code. An alternative is to make the grid size P by P and set the grid elements to be sub-grids of size M by M; $M = \frac{N}{P}$.

The collection element then takes the form

```
class GridBlock{
    double f[M][M], u[M][M],  p[M][M],
           q[M][M], r[M][M], Ax[M][M];
    double up_edge[M], lo_edge[M], le_edge[M], ri_edge[M];
    ....
};
```

where f is the initial data, u is the solution, and the remaining variables are temporaries used in the conjugate gradient algorithm or are, as in the case of the *edge* vectors, buffers used to store boundary values from neighboring grid blocks. The heart of the algorithm is given by the subroutine below. (We have overloaded operators -, +, and * for arrays here to simplify the text presented here. Our next version of the compiler will do this automatically.)

```
void conjugategradient(int num_iters, DistributedGrid<GridBlock> &G){
  int i;
  double gamma, alfa, tau, gamnew, beta;

  G.Ap();                          // q = A*p
```

```
    G.r = G.f-G.q;
    gamma = G.dotprod(R,R);
    G.p = G.r;
    for(i = 0; i < num_iters && gamma > 1.0e-12 ; i++){
       G.Ap();                        // q = A*p
       tau = G.dotprod(P,Q);          // tau = <p,q>
       alfa = gamma/tau;
       G.r = G.r - alfa*G.q;
       gamnew = G.dotprod(R,R);       // gamnew = <r,r>
       beta = gamnew/gamma;
       G.u = G.u + alfa*G.p;
       G.p = G.r + beta*G.p;
       gamma = gamnew;
    }
  }
```

Communication occurs only in the function *Ap* which applies the finite difference operator and in the *dotprod* function. In *Ap* the communication is all based on nearest neighbors and in the *dotprod* function the communication is all based on tree reductions.

6.2 BM-2: A Fast Poisson Solver

The Block Grid CG algorithm represents one type of PDE algorithm. Another common method is to use line oriented communications. For example, a standard fast Poisson solver involves applying a Fast Forier Transform based sine transform to each column of the array of data representing the right hand side of the equation. This is followed by solving tridiagonal systems of equations associated with each row. When this is complete, another sine transform of each column will yield the solution.

In this case it is best to view the grid as a distributed vector of vectors, where the vector class takes the form

```
class Vector{
      double data[N];
  public:
      fft();
      sineTransform();
};
```

The distributed vector collection must have a special function for solving tridiagonal systems of equations. In our case we use a standard cyclic reduction scheme. This is accomplished by building a collection class *DistributedTridiagonal* which is a subclass of *DistributedVector*. This class has a public function *cyclicReduction()* which takes two parameters which correspond to the diagonal and off-diagonal elements of the matrix which are stored in each element.

```
Collection  DistributedTridiagonal: public DistributedArray{
     public:
        DistributedTridiagonal(Template *T, Align *A, int n);
        void cyclicReduction(ElementType *diagonal,
                             ElementType *offdiagonal);
        ...
     MethodOfElement:
        ElementType *diag;     // the matrix diagonal
        ElementType *offd;     // the subdiagonal
        void backSolve(int s);
        void forwardElimination(int n, int s);
        ...
};
```

The cyclic reduction function, shown below, is organized as two phases of *log(n)* parallel operations. The first phases is the forward elimination, the second phase is the back solve step. At the first stage of the elimination, only the even elements participate. At the next stage only the elements with index a multiple of 4 participate, etc. We use the pC++ vector notation to select the subsets.

```
void DistributedTridiagonal::cyclicReduction(
                             ElementType *diagonal,
                             ElementType *offdiagonal){
     int s;
     int n = this->dim1size-1;
     this->diag = new ElementType(diagonal);
     this->offd = new ElementType(offdiagonal);

     for(s = 1; s < n/2; s = s*2){
            this[2*s:n-1:2*s].forwardElimination(n, s);
     }
     for(s = n/2; s >= 1; s = s/2){
            this[s:n-1:2*s].backSolve(s);
     }
}
```

Communication only happens within the element function forwardElimination and backSolve. In the forward elimination case, the active elements copy the element data s units away and algebraically eliminate them from the system. In the back solve, the values of the elements that are s units away are copied and are used to compute the local value.

The final poisson solver uses two collections F and U which represent the initial data and the final solution. Both are of type distributed tridiagonal of vector which is easily mapped to a one dimensional template. The main part of the program is shown below.

```
void FastPoissonSolve( DistributedTridiagonal<Vector> &U,
                       DistributedTridiagonal<Vector> &F){
        Vector diag, offdiag;
        ....
        F.sineTransform(U); // U <- sineTform(F);
        U.cyclicReduction(diag, offdiag);
        U.sineTransform(F); // F <- sineTform(U);
        U = F;
}
```

6.3 BM-3: The NAS Embarrassingly Parallel Bechmark

The NAS benchmark suite was designed to test the suitability of massively parallel systems for the needs of NASA Ames Research. Of the nine codes in the suite, four have been translated to pC++ and we report on two of them here. (A more complete report on this suite will be published when all have been translated and tested.)

The easiest of these is the "Embarrassingly Parallel" program. This code generates 2^{24} complex pairs of uniform (0, 1) random numbers and gathers a small number of statistics. In the benchmark the random numbers are created by two special portable functions called *vranlc()* and *randlc()*, and a main function *compute_pairs(int k)* is used to compute the numbers based on a different value of the seed parameter k. The main part of the computation is a loop of the form

```
for(i = 1; i < nn; i++) compute_pairs(i);
```

Our approach is to divide the work into a set of computational engines with one engine per processor. This "set" of Engines will be a collection of elements of type `Set<GaussianEngine>`. Each GaussianEngine object will compute $\frac{1}{NODES}$ of the total where $NODES$ is the number of processors.

The Engine element class is shown below.

```
class GaussianEngine{
  public:
   double  q[nq];  // final result statistics.

   double randlc(double *x, double * a);
   void   vranlc(int n, double *x, double a,double *y);
   void   compute_pairs(int kk);
};
```

To make a simple set collection we subclass *DistributedArray*. The main function, *computeGauss()*, calls an element function *do_work()* which does all the real computation. The final result statistics are kept in a local array q in each processor. A function sum_results is used to do the reduction necessary to sum these values. This involves a simple tree reduction.

```
Collection Set: DistributedArray{
  public:
    void computeGauss(void);
  MethodOfElement:
    virtual double q[nq];
    void   do_work(void);
    virtual void   compute_pairs(int kk);
    void sum_results();
};

void Set::do_work(void){
  int k;
      for(k = index1+1; k <= nn; k = k+NODES){
            this->compute_pairs(k);
      }
}
```

The main routine `computeGauss()` consists only of computing the random numbers with `do_work()` and then summing the resulting statistics with `sum_results()`.

6.4 BM-4: the NAS Sparse CG Benchmark

A far more interesting benchmark in the NAS suite is the random sparse conjugate gradient computation. This benchmark requires the repeated solution to $A*X = F$, where A is a random sparse matrix. There are two ways to approach the parallelization of this task. The best, and most difficult, is the matrix as the connectivity graph for the elements in the vector. By a preprocessing step one can partition the vector into segments of nearly equal size that minimizes the communication between subsets. The resulting algorithm can have the same low communication to computation ratio as our simple grid CG example above.

The second approach is not as efficient as the scheme that optimizes locality, but it a more direct implementation of the benchmark. We shall represent the matrix as a distributed random sparse matrix. More specifically, we shall represent the matrix as a Distributed Matrix of Sparse Matrices.

The *DistributedMatrix* collection class in the pC++ library allows any class that has the algebraic structure of a ring to be the element of the matrix. Furthermore, a p by p matrix whose elements are k by k matrices has all the same mathematical properties of a kp by kp matrix. Indeed, this is the key property used in the blocking algorithms used in most BLAS level 3 libraries. Consequently, we can select the element class to be a sparse matrix as shown below.

```
class SparseMatrix
{
  int rows, columns;
```

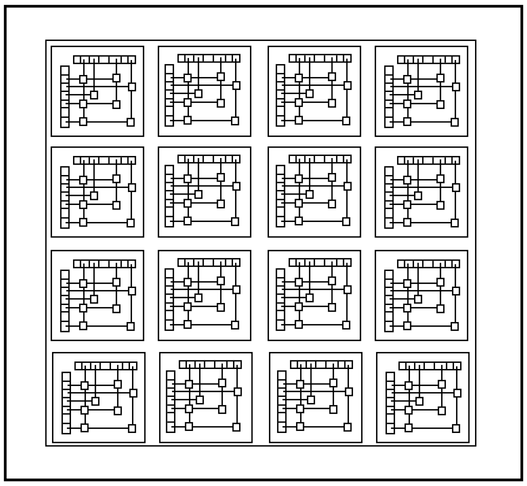

Figure 3: Alignment and Distribution

```
    data_ptr * r; // a vector of pointers to row lists
    data_ptr * c; // a vector of pointers to column lists

  public:
    ...
    void amux(Vector& Vout, Vector& Vin);
    ...
  };
```

`SparseMatrix` is a simple double linked sparse matrix with a special function `amux(Vout, Vin)` that can be used to compute the sparse matrix multiply $Vout = A*Vin$. The collection object that is the core of the computation is then

```
DistributedMatrix< SparseMatrix >  A(.... )
```

which is illustrated in Figure 6.4.

This distributed matrix of sparse matrices can then be used like an ordinary matrix. In particular, an efficient matrix multiply can be easily implemented that will multiply the distributed matrix by a distributed vector.

The benchmark kernel is then nearly identical to the Grid conjugate gradient algorithm from the first benchmark.

The primary communication consists of a broadcast of the *Vin* vector to each row of sparse blocks and then a reduction sum to compute the *Vin* results. The dot product reductions on the distributed vectors make up the remaining communication.

The standard NAS version of this benchmark comes in a test size of $n = 1400$ and the full test of $n = 14000$. We have run the full size problem only on the CM-5. All other statistics report the results for the small version.

7 Known Bugs in Version 1.0

The initial version of the pC++ parser still has a number of bugs. It is our hope that most of these will be fixed by the end of the first quarter of 1994. We will list here the bugs that we know about. There probably are many more.

- The operators { `,` `.*` and `->*` } cannot be overloaded.
- Overloaded functions must have the same return type.
- an external declaration followed by a real one as in

  ```
  extern int c;
  int c;
  ```

  will result in a double declaration of the variable.
- Member functions that are declared as inline outside a class declaration will not be inlined unless they are also declared as inline within the class declaration.
- A member field in a class may not be used before it is defined. For example, the compiler will complain about the following code.

  ```
  class A{
      int usei(){ return i;} .
      int i;
  };
  ```

  A related problem is with templates that references a member function or field of a class parameter.
- Static member initialization does not work. For example,

  ```
  struct ZZ{
      static int aa;
  };
   int ZZ::aa = 1;
  ```

- Using a symbolic parameter name more than once in a prototype will cause a parse error.

  ```
  int doublename(int a, float a);
  ```

- A basic block which occurs as the first thing in a function will not work.

  ```
  void foo() {
     { int x,y;
     }
  }
  ```

- The C++ alternate constructor form for base types has been temporarily disabled.

```
int i = int(33);  // use  (int) 33; instead.  fix coming soon.
```

In addition to these errors, templates and collections cannot be used together in version 1.0. Exception Handling is not supported on most platforms.

8 The pC++ Kernel Class

The pC++ Kernel class provides a number of functions. Most of these functions are low-level functions which usually will not be used by pC++ users. The definition for the class is given in the following with functions and variables private to the class omitted:

```
class Kernel {
    Kernel(Distribution *T,  Align *A, int element_size);
    ~Kernel();
    int Is_Local(int index);
    int Is_Local(int index1, int index2);
    int *Get_Element(int index);
    int *Get_Element(int index1, int index2 );
    int *Get_ElementPart(int index,int offset, int size);
    int *Get_ElementPart(int index1, int index2, int offset, int size);
    int Kernel::Get_ElementPart_Int(int index, int offset);
    float Kernel::Get_ElementPart_Float(int index, int offset);
    double Kernel::Get_ElementPart_Double(int index, int offset);
    int Kernel::Get_ElementPart_Int(int index1, int index2, int offset);
    float Kernel::Get_ElementPart_Float(int index1, int index2,
                                        int offset);
    double Kernel::Get_ElementPart_Double(int index1, int index2,
                                          int offset);
    int *Get_CopyElem(int index);
    int *Get_CopyElem(int index1, int index2 );
};
```

This describes the *alpha* version of the library. Note that there are instances of one and two dimensional construtors and functions, but nothing beyond that. This will be fixed in future releases. It is easy.

`Kernel(Distribution *T,  Align *A, int element_size)` is the constructor of the class. Distribution and Align are two class objects that the constructor needs as arguments. "element_size" is the size in bytes of an element class. Note: this last value is required for the current version of the compiler, but it will be droped in future releases.

`Is_Local(int index)` checks whether a pC++ collection element of an index "index" is stored in local processor's memory. It returns "true" if the element is in the local memory and "false" otherwise. `Is_Local(int index1, int index2)` is the two-dimensional implementation with "index1" being the first index and "index2" the second.

`Get_Element(int index)` returns a pointer to a collection element with the index, "index." If the element is local it returns a pointer to the element. If the element is not local, on a distributed-memory machine, it fetches the element and stores it in a local buffer and then returns a pointer to the buffer, or simply returns a pointer to the element on a shared-memory machine regardless of the element is local or not. `Get_Element(int index1, int index2)` is the two-dimensional implementation.

`Get_ElementPart(int index, int offset, int size)` returns a pointer to a segment of an element. Like `Get_Element`, it fetches the segment of the element if necessary. "index" again is the index of the element. "offset" is the offset of a field of the element with respect to the beginning of the element. "offset" has to be given in terms of integers. "size" is the size of the field in terms of integers. As an example, if myElement is declared as

```
class myElement {
 public:
   int i, j;
   double a[10];
   void foo();
}
```

the offset of the field a and the size are then given by

```
offset = ((char*)&myElement.a[0] - (char*)&myElement) / sizeof(int);
size = 10*sizeof(double)/sizeof(int).
```

`Get_ElementPart_Int(int index,int offset)`, `Get_ElementPart_Float(int index,int offset)`, and `Get_ElementPart_Double(int index,int offset)` return an integer, a floating point, or a double floating point field, respectively. "index" is the index of the element and "offset" is the offset of the field in integers.

Similar to `Get_Element(int index)`, `Get_CopyElem(int index)` fetches a copy of an element. However, instead of using the only buffer to store the fetched copy, it creates a new buffer for the fetched copy. This copy can be deleted or overwritten only by the user.

8.1 The pC++ SuperKernel collection

The SuperKernel collection inherits Kernel class, therefore, all functionalities available to the Kernel class are available as "methods of collection" in Superkernel. SuperKernel is designed as a base collection so that new collections can be derived from it. It provides a few basic element access functions. The declaration of the SuperKernel is as follows:

```
Collection SuperKernel : public Kernel {
 public:
  int dim1size;
  int dim2size;
  SuperKernel(Distribution *T, Align *A);
  ElementType  *operator()(int i);
  ElementType  *operator()(int i, int j);
 MethodOfElement:
  int Ident;
  int index1;
  int index2;
```

```
  SuperKernel<ElementType> *ThisCollection;
  ElementType *Self(int i);
  ElementType *Self(int i, int j);
};
```

In the "methods of collection" part (part before the MethodOfElement statement), "dim1size" and "dim2size" denote the first and second dimension sizes of the collection. They are set by the alignment parameter, Align, in the collection constructor. Operators `(int i)` and `(int i, int j)` are overloaded. So instead of using `Get_Element()`, a user uses these operators to fetch elements. Suppose we construct a one-dimensional collection of myElement by calling SuperKernel constructor,

```
  SuperKernel<myElement> S(&D, &A);
```

S(10) then returns a pointer to the 11th element of the collection and `S(10)->a` will return a pointer to the array "a." Meanwhile, `S(10)->a[5]` will return the 6th element in the array.

In the "methods of elements" part (part after the MethodOfElement statement), "index1" and "index2" denote the indexes assigned to a collection element. These indexes can be accessed by an element. For example, in a two-dimensional collection, function foo() can be defined as

```
void myElement::foo() {
  if(index1 == 1) printf("Hello from element (%d, %d)\n", index1, index2);
}
```

In this example, all elements with index1 = 1 print the message.

The pointer "ThisCollection" provides a way for the methods of element to access methods of collection. Without it methods of element cannot execute a method of collection. We can modify the above example to allow foo() access a collection element:

```
void myElement::foo() {
  a[5] = (*ThisCollection)(2,3)->a[5];
}
```

In this example the value of a[5] of collection element (2,3) is assigned to a[5] of the current element. Since every element executes this operation, the value of a[5] of element (2,3) is broadcast to other elements in the collection. Note that currently this is not an efficient way to broadcast, because `(*ThisCollection)(2,3)` simply invokes the overloaded operator which fetches the entire element (2,3) and returns a pointer to the copy. In our future release, the compiler will convert this to `Get_ElementPartDouble` so that only array element a[5] is fetched.

The "Self" functions in methods of element provide another convenient way for an element to address other elements in a collection. The integers "i" and "j" represent index offsets to the current element. As an example, Self(1,1) would return the element with indexes (index1+1, index2+1), where index1 and index2 are the indexes of the current element.

9 The DistributedArray Collection

DistributedArray provides the a number of functions for the user-interface. The definition of the collection and some useful functions are given below. Functions and variables internal to the collection are omitted.

```
Collection DistributedArray : public SuperKernel {
 public:
  DistributedArray(Distribution *T, Align *A);
  void Assign(int i, ElementType &e);
  void Assign(int i, int j, ElementType &e);
  DistributedArray<ElementType>
                &operator = (DistributedArray<ElementType> &rhs);
  void Shift(int dim, int distance);
  void Reduce();
  void ReduceDim1();
  void ReduceDim2();
  void Broadcast();
  void BroadcastDim1();
  void BroadcastDim2();
  double DotProd(DistributedArray<ElementType> &arg1);
 MethodOfElement:
  void WhereAmI();
  virtual double Dot(ElementType &arg1);
  virtual ElementType &operator +=(ElementType &rhs);
  double dotprodtemp;
};
```

`DistributedArray(Distribution *T, Align *A)` is the constructor of the collection. It has class objects Distribution and Align as its arguments.

`Assign(int i, ElementType &e)` copies an element of DistributedArray element type, argument "e," to an element of an index "i" of the invoking

`DistributedArray object.  Assign(int i, int j, ElementType &e)`

is its two-dimensional counter part.

Operator "=" is overloaded. It copies a DistributedArray object to another DistributedArray object.

`Shift(int dim, int distance)` performs a circular shift of collection elements parallel to a given dimension of the collection by a given distance. Elements shifted out at one end are shifted in at the other end. For example, Shift(1, 2) will shift elements parallel to the first dimension by a distance of 2.

`Reduce()` applies a global reduction to collection elements. Elements are summed together, the result is stored in element "0," the very first element in the collection. It is assumed that the total number of elements in the collection is a power of two. A user needs to define function &operator +=(ElementType &rhs) before invoking this function.

ReduceDim1() applies a reduction parallel to the first dimension of the collection. Elements are summed together along columns, if the first index is defined as the row index and the second as the column index. The results are stored in elements in the first row. It is assumed that the number of rows is a power of two. A user needs to define function &operator +=(ElementType &rhs) before invoking this function.

ReduceDim2() applies a reduction parallel to the second dimension of the collection. Elements are summed together along rows. The results are stored in elements in the first column. It is assumed that the number of columns is a power of two. A user needs to define function &operator +=(ElementType &rhs) before invoking this function.

Broadcast() broadcasts element zero to all others elements in a collection. After the broadcast, each element gets an exact copy of the first element in the collection. It is assumed that the total number of elements in the collection is a power of two.

BroadcastDim1() broadcasts elements parallel to the first dimension. Elements in the first row are broadcast to elements in other rows. It is assumed that the number of rows is a power of two.

BroadcastDim2() broadcasts elements parallel to the second dimension. Elements in the first column are broadcast to elements in other columns. It is assumed that the number of columns is a power of two.

DotProd(DistributedArray<ElementType> &arg1) performs dot-product between elements of "arg1" and that of the invoking DistributedArray object. A user needs to define function Dot(ElementType &arg1) before invoking this function. This function can be used, for example, if each DistributedArray element is a segment of a vector that is distributed across the DistributedArray elements. A "local dot-product" is performed by the Dot function, which computes the dot-product between elements of the invoking object and the corresponding elements of "arg1." The results of the local dot-product are added together by a global tree reduction to form the sum of the dot-product. The result of the sum is then broadcast to each element and stored in "dotprodtemp."

WhereAmI() prints out indexes of an element and the ID of the processor on which the element resides.

Virtual function Dot(ElementType &arg1) is called by function DotProd and needs to be defined by the user. Its implementation depends on the data structure of the element. As an example, element class myElement can be defined as follows:

```
class myElement {
 public:
   int i, j;
   double a[10];
   double Dot(myElement &arg1);
};
double myElement::Dot(myElement &arg1)
{
```

```
  int i;
  double dot_sum;
  for(i = 0; i < 10; i++)
    dot_sum += this->a[i]*arg1.a[i];
  return dot_sum;
}
```

The function computes the dot-product between array "a" of two collection elements. A user needs to define a dummy `Dot(myElement &arg1)` if function DotProd is to be invoked.

Virtual function `&operator +=(ElementType &rhs)` is called by the reduction functions and needs to be defined by the user. Its implementation depends on the data structure of the element. As an example, class myElement can be defined as

```
class myElement {
 public:
   int i, j;
   double a[10];
   myElement &operator +=(myElement &rhs);
};
myElement& myElement::operator +=(myElement &rhs)
{
  int i;
  for(i = 0; i < 10; i++)
    {
      this->a[i] += rhs.a[i];
    }
  return *this;
}
```

Here the overloaded operator sums the corresponding elements of array "a" together.

Acknowledgments

This research is supported by DARPA under contract AF 30602-92-C-0135 from Rome Labs, National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616.

Special thanks to Bernd Mohr at University of Oregon for proofreading the manuscript and pointing out several errors in an earlier draft.