Distributed pC++: Basic Ideas for an Object Parallel Language *

François Bodin,
Irisa, University of Rennes
Campus de Beaulieu, 35042 Rennes, France
Peter Beckman, Dennis Gannon, Srinivas Narayana, Shelby X. Yang
Department of Computer Science
Indiana University, Bloomington, Indiana

Abstract

pC++ is an object-parallel extension to the C++ programming language. This paper describes the current language definition and illustrates the programming style. Examples of parallel linear algebra operations are presented and a fast poisson solver is described in complete detail.

1 Introduction

This paper provides an overview of pC++, an object-parallel programming language for both shared and distributed memory parallel systems. Traditional data-parallel systems are defined in terms of the parallel action of primative operators on the elements of array data structures. Object-parallelism extends this basic model of data-parallel execution to the domain of object-oriented software by allowing the concurrent application of arbitrary functions to the elements of arbitrary distributed, aggregate data strucutres.

Because C++ is a de facto standard for a growing community of application designers, we have chosen to base the system around that language. More specifically, the target audience for pC++ are programmers who have chosen C++ over Fortran for reasons of software engineering, i.e., the construction of applications with complex, dynamic data structures not easily expressed in Fortran 77, or the need for an object-oriented framework not representable in Fortran 90. In this situation, we believe pC++ is ideal when performance is a premium and parallelism is the solution, so long as the result is portable.

Additional design objectives include:

*This research is supported by DARPA under contract AF 30602-92-C-0135, the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616, and Esprit BRA APPARC

- pC++ should have few departures from standard C++ syntax. The only new language constructs are the *Collection* `class` and vector expressions.
- A multi-threaded, Single Program, Multiple Data (SPMD) programming style should be available when needed. In addition to a data-parallelism model, a programmer can view part of the program as a single thread of control that invokes parallel operators on aggregates of data. At the same time, these parallel operators can be viewed as interacting threads of computation that have explicit knowledge of data locality.
- There should be no "*shared-address-space*" requirement for the runtime model. However, there needs to be a *shared name space* for distributed collections. This allows any execution thread to reference and access any element of any `collection`.
- Parallel programs written in pC++ should be portable across all scalable parallel systems. The High Performance Fortran Forum has now defined a standard for data-parallel programs that can be compiled across a wide variety of architectures. We have used much of the HPFF Fortran as a model for mapping data and computation to MPP systems in pC++.
- The pC++ compiler should make extensive use of optimization techniques that have been developed for transforming programs to run efficiently on parallel systems.
- pC++ programs should use of libraries of high quality, scalable, parallel algorithms. We expect that more than half of our effort will be devoted to building new libraries and integrating existing ones into the system.

pC++ is derived from an earlier experiment by Jenq Kuen Lee [6]. In its original form, pC++ proved that it was possible to build a language that provided real portability of "object-parallel" programs across a variety of shared-address-space parallel systems. A major challenge for pC++ is to show that a shared address space is not essential for good performance.

2 Collections

pC++ is based on a simple extension to C++ that provides parallelism via *distributed aggregates*[3, 6]. Distributed aggregates are structured sets of objects distributed over processors of a parallel system. Data structures such as vectors, arrays, matrices, grids, trees, dags or any other large, application specific aggregate can be described in pC++ as a *collection* class type. In a manner similar to C++ templates, collections are parameterized by the data type of the *element*s that make up the collection. Consequently, it is possible to describe structures such as a "distributed vector of real" or "distributed vector of quaternion" without modification to the concept of distributed vector. Indeed, collections of collections of objects can be defined such as a "tree of grids of finite-elements."

Parallel operations on a collection can be generated either by the concurrent application of a method of the collection elements to the entire aggregate or by the application of a parallel operator associated with the collection itself. This form of parallelism is derived from data parallelism and we call it *object parallelism.*

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 *Template* objects. A template defines a "logical coordinate system" for arranging the distributed elements in a given computation in relation to each other.[1] The mapping of collection elements to a template is specified by an *alignment* object. In the second step, the template is mapped to processors by a mechanism called *distribution.*

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 ...);
```

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 alignments and template objects.

2.1 Templates, Alignments, and Processors

Templates 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 template 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 HPFF Fortran, templates in pC++ are first class objects.[2] A template is characterized by its number of dimensions, the size in each dimension and the distribution by which the template is mapped to processors. Current distributions allowed in pC++ include BLOCK, CYCLIC, and WHOLE.

To map a two-dimensional matrix A to a set of processors one can define a two-dimensional template and align the matrix with the template and then map the template to the processors. Suppose we have a 7 by 7 template and the matrix is of a size of 5 by 5, and suppose the template will be distributed over the processors by mapping an entire row of the template to an individual processor and the i^{th} row is mapped to processor i mod P on a P processor machine, this mapping scheme corresponds to a CYCLIC distribution in the template row dimension and a WHOLE distribution in the template column dimension. pC++ has a special library class called *Processors* which is implementation dependent. In the current implementations, it represents the set of all processors available to the program at run time. The template and distribution can be defined as follows.

[1] We regret the potential confusion between the HPFF Fortran template concept and the C++ keyword *template*. To avoid any potential problem we will not use examples involving C++ templates in this paper.

[2] Because templates 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 template and align objects can be taken from another collection with which the new collection is to be aligned.

```
Processors P;
Template myTemplate(7, 7, &P, CYCLIC, WHOLE);
```

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

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

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

We may now align the vectors to the same template. 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], myTemplate[0][i])]");
Align YAlign(5, "[ALIGN( Y[i], myTemplate[i][0])]");
DistributedVector<Complex>  X(&myTemplate, &XAlign);
DistributedVector<Complex>  Y(&myTemplate, &YAlign);
```

The alignment and the template form a two stages mapping, as illustrated in Figure 1. The array is mapped into the template by the alignment object and the template definition defines the mapping to the processor set.

Because all of the standard matrix-vector operators are overloaded with their mathematical meanings. This 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.

2.2 The Structure of a Collection

Collections are a special type of class with the following syntax.

```
Collection NameOfCollectionType: Kernel{
    private:
      < private data fields and method functions>
```

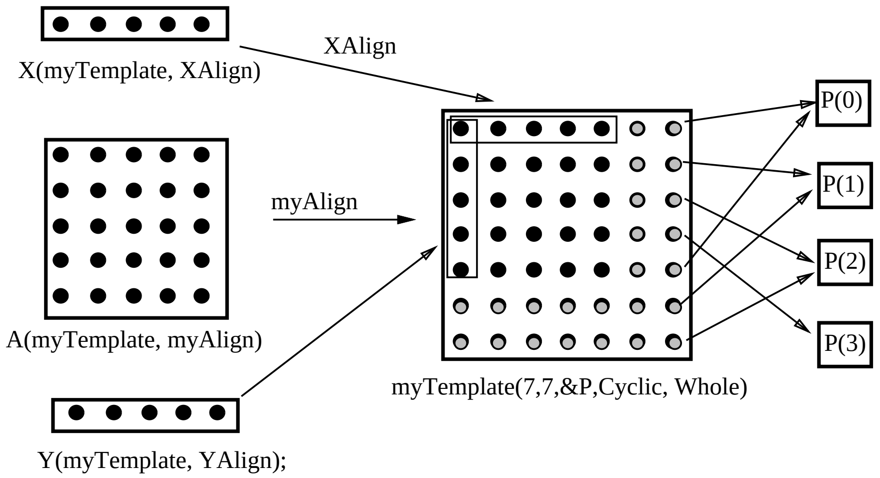

Figure 1: Alignment and Template

```
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>
};
```

- Collections are derived from a special machine dependent root class called *Kernel*. The `Kernel class` uses the template 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 field and member functions that are copied into each element as protected 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.

2.3 Control Flow and Parallelism

A `collection` is a set of elements which are objects from some C++ `class`. The primary form of parallelism in pC++ is the application of a collection `MethodOfElement` function or an element class method to each element of the `collection`. In other words, let C be a `collection class` and E is a standard C++ class. If c is declared to be of class `C<E>`, that is, c is a collection that has elements of class E, and $f()$ is a `MethodOfElement` function of C or a method function of E, then the **object parallel** expression

```
c.f();
```

means

```
for all e in c do
     e.f();
```

The alignment and template distribution functions partition the `collection` and map a subset of element to each processor. This subset is called the *local collection*. If sufficient processor resources exist, all of these element function invocations in `c.f()` can happen in parallel. If there are fewer processors than elements, then each processor will sequentially apply the method function to the subset of element in its local collection.

Note that if `f()` returns a value of type T when applied to an element, the expression `c.f()` will return a value of type `C<T>`. Likewise, if `x` is a public field of E of type S, the expression `c.x` is of type `C<S>`.

Because all collections are derived from the `Kernel class`, they all inherit an indexing function and a subset operator. The expression `c(i)` returns a pointer to the i^{th} element if it is in the local collection of the processor evaluating the expression, or it returns a pointer to a buffer which contains a copy of the i^{th} element if it is not local. The expression

```
c[a:b:s].f();
```

means concurrently apply `f()` to $e_i \in c$ for i in the range $[a, b]$ with step size s.

As with C++, all pC++ programs begin execution with a single control thread at `main()`. When a `collection` function is encountered, the control thread forks a separate thread onto each processor. These *processor threads* must be synchronized before returning from the `collection` function where the main control thread continues.

For element `class` functions and `MethodOfElement` functions this synchronization is automatic. However, for `public`, `private` and `protected` functions of the `collection`, the processor threads are said to be executing in asynchronous, *Single Program, Multiple Data*, SPMD, mode. This means

- all variables visible within the scope of the function are private to the processor thread and
- the programmer is responsible for the synchronization between processor threads at the end of the function.

In the next subsection we first give a simple example to illustrate all of the points described above. The sections that follow explore different aspects of pC++ in more detail.

2.4 Hello World

To demonstrate the basic extensions to C++ provided by pC++ we consider a simple example consisting of a set of elements that each print a simple “Hello World” message to the standard output stream. We will build a simple *linear set collection* with one constructor method and a private field *size*. Our linear set will add a field to each element called *myindex* and add a special function to the element class *sayHello()* which will print the value of *myindex* and call a work routine *doWork()* from the element. The definition of the `collection` is given below.

```
Collection LinearSet: public Kernel{
         int size;
      public:
         LinearSet(Template *T, Align *A) :
                     Kernel(T, A, sizeof(ElementType)){
             int i;
             size = T->dim1size;
             for (i = 0;  i < size; i++)
                  if (this->Is_Local(i)) (*this)(i)->myindex = i;
              Barrier();
           };
     MethodOfElement:
         int myindex;
         virtual void doWork( void );
         void sayHello(){
            printf("Hello World from %d\n", myindex);
            doWork();
            };
  };
```

We may use any C++ `class` as the element type of the `collection` so long as *doWork()*, which is declared as virtual in the `collection` definition, is present. For example,

```
class  MyElement{
    public:
      float x, y;
      MyElement(float xinit){
          x = xinit;
          };
      void doWork(){
          y = x*x;
          };
      MyElement & operator +(MyElement & Rhs);
      MyElement & bar(int i);
```

```
};
```

A main program which will create a collection of this type and do a parallel invocation of each of the *sayHello* function is given below.

```
#include "kernel.h"
#include "distarray.h"
#define SETSIZE 5
main(){
      Processors P;
      Template myTmplate(SETSIZE, &P, BLOCK);
      Align myAlign(SETSIZE, "[ALIGN( domain[i], myTmplate[i])]" );
      LinearSet<MyElement>  G(&myTemplate, &myAlign, 4.0);

      G.sayHello();
}
```

The result of this computation will be five "Hello World from .." messages and each element object will set its y variable to 4.0 .

To see how this works, we describe the behavior line by line. *main()* starts a single, logical thread of control, initializing the template and alignment objects.

The constructor for the LinearSet is then called. The first task of the constructor is to call the kernel constructor to initialize the `collection`. Notice that the constructor was invoked in the main program with the additional argument, 4.0. Additional constructor arguments are passed to the element constructor which in this case has one parameter, a `float`. The constructor initializes the size field on each processor's local copy of the `collection` structure with the value extracted from the template. It then initializes the `myindex` field in each element of the `collection`.

```
LinearSet(Template *T, Align *A) : Kernel(T, A, sizeof(ElementType)){
     int i;
     size = T->dim1size;
     for (i = 0;  i < size; i++)
             if (this->Is_Local(i))  (*this)(i)->myindex = i;
     Barrier();
 };
```

To do this, two special `Kernel` functions are needed: *Is_Local(index)* and *Barrier()*. As soon as the main control thread enters a public function from a collection the execution model changes: each processor now operates on its own local portion of the collection in SPMD mode. The user may view this as if the original control thread had split into a number of independent threads, one per processor. These *processor threads* must always be synchronized at the end of the collection function before the return to the main control thread. The `Is_Local(index)` predicate returns true if the named element is in the local memory associated with this thread of the computation.

Another idea borrowed from HPFF Fortran is the "owner computes" rule. pC++ requires that only the "owner," i.e. the local processor thread, of an element may modify fields or invoke functions that modify the elements state.

The function *(*this)(i)* returns a pointer to the local element with global name i. If the element accessed by `(*this)(i)` is not local the function returns a pointer to a buffer containing a copy of the i^{th} element. [3]

The *Barrier()* function causes each thread to wait until all others reach this point in the program. Upon return from the `collection` constructor operation we return to the single main thread of execution.

The final line of the program is

```
G.sayHello();
```

This invokes `sayHello()` in *object parallel* mode on all elements of G. The function is applied to each element in an unspecified order. Operationally, each processor invokes the function sequentially on that portion of the collection that is local.

In addition to printing the "hello from ..." message, each element function also calls *doWork()*, which modifies the field y in each element. Each parallel operation is implicitly barrier synchronized, i.e. all processors wait until "sayHello" has been invoked for all elements of G.

Exploring this example collection a bit further, we note that because the member values x and y in the element class `MyElement`, of the collection G, are public, one can write

```
G.x = G.x + G.y
```

This is a parallel addition for each element of the `collection`. Furthermore, we have also overloaded the plus (+) operator within the element, so the expression $G + G$ defines a new `collection` of the same type. Also the function *bar* returns a reference to an object of `class MyElement`. Consequently, the expression

```
(G.bar() + G).sayHello()
```

is valid. In these cases of *implicit* collections, the distribution and alignment are inherited from the *leftmost* `collection` in each subexpression.

2.5 Building Abstract Collections

Library collections are designed to be applied to many different types of elements. For example, matrix multiplication can be written without referring explicitly to the type of the element. This is accomplished in pC++ by using the `ElementType` keyword as a place-holder for a class that can be supplied later. However it is often the case that a `MethodOfElement` function defined within a `collection` must refer to some property of the element in order to complete its task. The key word `virtual` is used to indicate methods of the element that `collection` requires. Consider the pC++ library `collection class` "Distributed Block Vector". This `collection` is used for blocked vector operations and it is usually used with an element that is vector type object. The idea behind these classes to be able to exploit well tuned sequential `class` libraries for matrix vector computation. Many of these are becoming available in both the public domain and commercial sources. A blocked vector is a segment of a large vector that

[3] Note that potential race conditions can occur at this point. One processor may be modifying a local element while another processor is reading a copy that may be out of date. It is up to the programmer to make sure that this does not cause problems.

resides on a processor. By supplying a well tuned *Vector* `class` as the element type, very good performance can be obtained.

Distributing and blocking a vector does not change its functionality. For example, the operator `sub()` is a blocked vector subtraction and, when called as

```
DistBlkVector<Vector> A,B,C;
C.sub(A,B)
```

it does the vector operation $C = A - B$. Because *sub* operates on collections it must, in turn, invoke a function that knows how to do a subtraction of elements. In the library, `sub()` is defined as

```
Collection DistBlkVector: DistributedArray
{
   float dotProduct(DistBlkVector<ElementType>);
    double dotprodtemp; // used for dot product
   ....
 MethodOfElement:
    int index1;  // the index of "this" element.
    ...
    void sub(DistBlkVector<ElementType> &arg1,
             DistBlkVector<ElementType> &arg2){
                this->elementSub(*(arg1(index1)),*(arg2(index1)));
        };
    virtual void elementSub(ElementType &arg1, ElementType &arg2);

    void LocalDotProduct(Vector<ElementType> &arg){
        ThisCollection->dotprodtemp += this->elementDot(*arg1(index1));
        };
    virtual void elementDot(ElementType &arg);
   .....
}
```

The `sub()` function applies the element subtraction function `elementSub()` to copies of the corresponding elements in the argument collections. `elementSub()`, a `virtual ElementType` member function, does the actual elementwise subtraction. Consequently, any `class` that is provided as the element type of a `DistBlkVector` `collection` must provide this function.

`virtual` can also be used in a `class` that is the basic element of a `collection` to refer to the element field declared in the `collection`. However, we do not encourage this practice.

3 The Kernel Class

Before proceeding further with examples of the language it is essential to take a brief look at the functions provided by the *kernel class*.

The role of the kernel is to provide a global name space for the `collection` elements and also provides method for managing parallelism and `collection` element accesses. The kernel methods are the following:

```
int Install_Collection(Template *T, Align *A, int sizeelem);
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 startof, int size);
int *Get_ElementPart(int index1, int index2,int startof, int size);
int *Get_CopyElem(int index);
int *Get_CopyElem(int index1, int index2 );
```

The `Install_Collection()` takes a template and an alignment `class` to create the `collection`. For each `collection` in the program, there is a special data structure that is created and stored on each processor of the system. This "local `collection`" object contains all the elements that are assigned to the processor by the initial alignment and template distribution. In addition the local `collection` object contains a table that describes how non-local elements may be accessed. This table is very similar to the distributed manager used in shared virtual memory systems [8] [14]. Every element has a *manager* processor that is responsible for keeping track of where an element resides and every processor knows who the manager of each element is. The *owner* of an element is the processor that has that element in its local collection.

The `Get_Element(i)` method returns the i^{th} `collection` element. If the element is local then a reference to the element is returned. Otherwise a buffer, owned by the `collection`, is loaded with a copy of the remote element. The protocol consists of two requests:

1. Ask to the manager of the element which processor currently owns the element.
2. Load a copy of the element from the processor that owns the element.

The reason for this two stage manager-owner scheme is to simplify dynamic collections and to provide a simple mechanism for load balancing. Our initial experiments on the CM-5 have shown that the added latency introduced by this scheme is very small (see [13]).

`Get_ElementPart()` and `Get_CopyElem()` are similar. The former returns part of the elements and the latter allocates a new buffer to keep a copy the element. However there is no coherence for element copies and, at this time, no methods for updating a remote element is provided. However this has not been ruled out for the future.

The current kernel is implemented on the NCSA CM5 using the CMAM communication package [2], the Intel Paragon using NX.

4 Collection Function SPMD Execution

The greatest potential hazard for programmers using pC++ lies in managing the interface between the single control thread of the main program and the "multi-threaded SPMD" functions of the collections. In particular, if a `public collection` function returns a value, care must be taken to insure each thread returns the same value! For example, consider the case of a standard parallel reduction like the dot product.

```
DistBlkVector<Vector>  X, Y;
float alpha;
```

```
alpha = X.dotproduct(Y);
```

If `dotproduct()` is called from the main control thread then each processor thread will compute a value for the function. Because the result is assigned to a single scalar, the operation is not well defined unless all of the results are identical. This problem is easy to understand if one considers the runtime behavior on a distributed memory multicomputer. The main control thread is run concurrently on each processor. All global variables and the stack are duplicated on each processor. (It is the job of the compiler to make sure that any problems with a concurrent execution of the "sequential code", such as I/O, are properly handled.)

To solve the problem of making sure the collection member functions all return a consistant value, the library provides a family of special reduction functions. Consider the case of the `dotproduct()` function.

```
double DistBlkVector::dotproduct(
       DistBlkVector<ElementType> & arg)
 {
    dotprodtemp = 0.0;
    this->LocalDotProduct(arg);  // called for each element
    return( sumReduction(dotprodtemp));
 }
```

To execute `X.dotproduct(Y)` each processor thread first sets the variable `dotprodtemp` in the corresponding local collection to zero. Then data parallel action of `LocalDotProduct(arg)` computes the element level dot products. This computation is a simple sequential iteration on each processor thread which accumulates the values into the `dotprodtemp` variable of the collection:

```
ThisCollection->dotprodtemp += this->elementDot(*arg1(index1));
```

The pointer `ThisCollection` is a inherited from the DistributedArray and it provides a way for each element to identify the collection to which it belongs. The `sumReduction` function computes and returns the total of the arguments on each processor thread.

The reader may object to the fact that we have left the burden of synchronizing thread and returning consistent values to scalar functions up to the programmer. The alternative is to use a language extension that requires each collection function to use a special form of the `return` statement. Experience with applications should tell us which is the correct path.

Other functions that can be used to synchronize and assure consistent returned values are `multReduction`, `andRedcution` and `orReduction`. Functions like `broadcastBuffer` can be used transmit a block of data from one processor thread to all the others.

5 The Collection Library

The `collection` library for pC++ is designed to provide a set of primitive algebraic structures that may be used in scientific and engineering computations. Organized as a `class` hierarchy, the `collection class` tree is rooted with the kernel. There are two basic types of collections: dynamic and static. Within the category of static structures, we have all the standard array, matrix and grid classes. These are described in detail

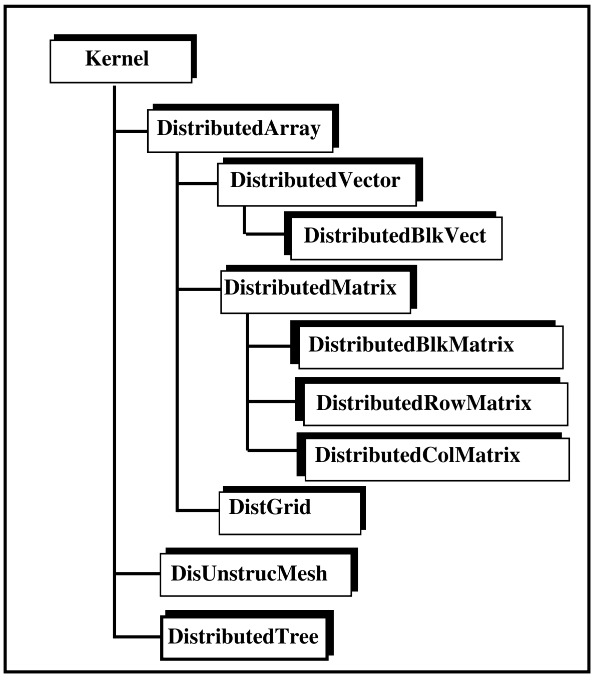

Figure 2: The Collection Hierarchy

below. The other category includes `collection` types like trees, unstructured meshes dynamic lists and distributed queues. This later category is still far from complete in design and will be a subject of intense research over the next few years.

There are two aspects to the problem of designing a parallel library for a language like pC++. First, one needs a basic set of element classes for numerical computation that are well tuned to the individual processors of each parallel system. Second, one must have a set of collection definitions and the associated parallel operators that are both scalable and easy for the user to extend. In addition, these collections must be easy to integrate with other C++ numerical packages, like the LaPack++ system being designed at Oak Ridge and Tennessee. Consequently, one can expect the definitions shown below will change as these other packages come available.

5.1 The Distributed Array

The *DistributedArray* `class` provides essential operators for manipulating arrays of objects. Distributed arrays can be of arbitrary dimension and the element `class` type is restricted only by the nature of the application. More accurately, distributed arrays have a number of special global operators. These can only be used if the underlying element has the required properties. For example, it is not possible to find the maximum value of a one dimensional array of elements if the elements do not have a total order based on the $>, =, <$ operators.

The Distributed array operators also include a generalization of the Fortran 90 array section operators. By this we mean that if A is a DistributedArray of dimension k, the expression

$$A[l_1 : u_1 : s_1, l_2 : u_2 : s_2, ..., l_k : u_k : s_k]$$

refers to the subdomain of the array specified in the i^{th} dimension by the lower-bound, upper-bound, stride triplet $l_i : u_i : s_i$.

As with Fortran 90, a subarray can be used whereever a full array can be used. For example, if M is defined by

```
DistributedArray<...>  *M;
M = new DistributedArray(&A, &T);
```

where the alignment A gives the dimension of M to be 2, then if

```
(*M).sin(x)
```

means to apply the element function `sin()` to all the elements of the `collection`, then

```
M[1:100:2, 1:100:2].sin(x)
```

means apply `sin()` to the odd indexed lattice points in M. Similarly, one can use $M[i : m : k, s : t : r]$ in any expression where one can use $*M$.

The core distributed array functions include:

- `ElemenType *operator()(int i,...)` returns a pointer to either the $(i^{th}...)$ element if it is local to the executing processor, or a pointer to a buffer containing a copy of the $(i^{th}...)$ element if it is non-local.
- `DistributedArray<ElementType> &saxpy(a, x, y)` does pointwise scalar element a times array x plus array y assigned to `this` array. Other standard BLAS-1 Vector operators are supported.
- `void Shift(int dim, int distance)` parallel shifts an array by the given distance in the given direction (with end around wrap).
- `void Broadcast()` broadcasts the value in location $(0, 0, ..)$ to the entire array.
- `void BroadcastDim(int I)` broadcasts the values along dimension I where $I = 1, 2, ...$
- `void Reduce()` does a parallel reduction of the array leaving the result in location $(0, 0, ..)$.
- `void ReduceDim(int I)` does a reduction along dimension I where $I = 1, 2, ..$ and leaves the result in the orthogonal hyperplane containing $(0, 0, ...)$.

All operations that require element "arithmetic" assume that the base element `class` has operators for basic arithmetic and some cast operators that will allow a real number to be case in to the element type.

As mentioned previously, if the operators like `+, =, -, *, /`, etc. are defined for the array element type then the collection level operations are also well defined.

5.1.1 The Matrix Collection Classes

Many applications use arrays are used to represent matrices, tensors and vectors. To simplify programming, we have defined two types of distributed matrix and vector collections.

The *DistributedMatrix* and *DistributedVector* classes are derived from the DistributedArray `collection class`. Their functionality provides access to well tested and tuned parallel linear algebra operations. More specifically, these collections provide BLAS-3 level operators for distributed arrays that represent matrices and vectors. For example: for `DistributedMatrix` we have

- `DistributedMatrix &operator *(Distributed Matrix &)` gives the matrix product rather than the pointwise array product.
- `DistributedVector &operator *(Distributed Vector &)` is the matrix vector product.
- `DistributedVector &transProd(Distributed Vector &)` is the matrix vector product using the transpose of this matrix.

A larger project to build a distributed LAPACK project is underway. This work will be based on adapting the SCALAPACK library being developed as part of a consortium involving Berkeley, Illinois, Oak Ridge and Tennessee. In addition we will work with the Illinois Splib project.

5.1.2 Blocked Matrices and Vectors

A Blocked Distributed Matrix *DistBlkMatrix* is much like a *DistBlkVector*. The key idea is the observation that many matrix computations on parallel machines partition a large matrix into an array of submatricies, where each submatrix is located on one processor. Many matrix algebra operations for large matrices can be decomposed into operations on smaller matrices, where the elements of the smaller matrix are submatrices of the original. For example, given a `class` *Matrix* with all the standard properties of algebraic matrices overloaded, we can create a q by q distributed matrix of matrices, each of size p by p. The operation

```
Processor P(q*q);
Template T(q, q, &P, Block, Block);
Align A(q, q, "[ALIGN( domain[i][j], T[i][j])]" );
DistributedMatrix< Matrix >  M(&T, &A, p,p);
DistributedMatrix< Matrix >  N(&T, &A, p,p);
M = M*N;
```

is mathematically equivalent to standard matrix multiplication on objects declared as:

```
Processor P(q*q);
Template T(p*q, p*q, &P, Block, Block);
Align A(p*q, p*q, "[ALIGN( domain[i][j], T[i][j])]" );
DistributedMatrix< float >  m(&T, &A);
DistributedMatrix< float >  n(&T, &A);
m = m*n;
```

In the first case there is one element per processor and it is a p by p matrix. In the second case there is a p by p submatrix assigned to each processor. The advantage of the first scheme is that the basic matrix element operation can come from a library well tuned for the individual processor while in the second case it is up to the compiler to automatically block the resulting code in an efficient manner. (A research project is in progress to make the transformation from the second form to the first automatically.)

The only disadvantage of the Matrix-of-Matrices solution is that access to an individual element can be more difficult. To fetch the (i, j) element of the matrix m one uses $m(i, j)$. To fetch the same value from M requires

```
M(i/p, j/p)(i%p, j%p);
```

To simplify this task of translating the index and sub-array notation from the mathematical level to the matrix-of-matrix references, we have introduced the `collection` `DistBlkMatrix`. The matrix corresponding to M would be written as

```
Processor P(q*q);
Template T(q, q, &P, Block, Block);
Align A(q, q, "[ALIGN( domain[i][j], T[i][j])]" );
n = p*q; // the size of the mathematical matrix
DistBlkMatrix< Matrix >  M(&T, &A, n,n);
```

In this case a reference to the value of an individual element is given by `M.get(i,j)` or `M.put(i,j, value)`. Columns can be accessed with the function `M.col(int i)` which returns a `DistBlkVector` which is defined in terms of

```
DistributedVector<Vector>
```

for some uniprocessor vector `class`. Similarly for rows and diagonals. Multiplication of `DistBlkMatrix<Matrix>` by `DistBlkVector<Vector>` works as expected so long as the Matrix*Vector element operations are well defined.

Another reason for using a *matrix-of-matrices* structure is for distributing sparse matrices or block structured matrices. For example, if one has a `class` *SparseMatrix* that works with the `class` Vector, then the `collection`

```
DistBlkMatrix<SparseMatrix>  S(....);
DistBlkVector<Vector>  X(..), Y(...);
Y = S*X;
```

can be a handy way to construct a `collection` that has the global structure of a sparse matrix but still allows simple algebraic operators to be applied at the source level. For example, this is used in the pC++ implementation of the NAS sparse CG benchmark [10].

Along these lines, other useful collections for sparse structured matrices would be

```
DistBandedMatrix< TridiagonalMatrix >.
```

These will eventually be added to our library.

6 Parallel Tridiagonal Systems and A Fast Poisson Solver

In this section we describe the construction of a simple parallel algorithm for solving tridiagonal systems of equations and then show how it can be applied to solving the Poisson equation.

6.1 Tridiagonal Solvers

A standard parallel algorithm for solving diagonally dominant, tridiagonal systems of linear equations is cyclic reduction. This method uses Gaussian elimination without pivoting. The algorithm orders the elimination steps with the odd elements first, followed by the elements that are not multiples of four, followed by the elements that are not multiples of eight, etc. At each stage all the eliminations can be done in parallel provided that the updates are done correctly. (See [4] for details.)

One way to program this in pC++ is to build a special subcollection of the DistributedVector to represent the tridiagonal system of equations. For simplicity we shall assume the matrix is symmetric and all diagonal elements are identical and all offdiagonal elements are equal. (These assumptions are sufficient for the PDE example that follows). Though the tridiagonal matrix can be described by two numbers, the diagonal, and the offdiagonal, the elimination process will destroy this property. Consequently, we will need two vectors of coefficients, one for the diagonal, a, and one for the offdiagonals, b. Also to simplify the boundary conditions in our program, we will assume $n = 2^k$ for some k and the vector length is $n + 1$ and the problem size is $n - 1$. Our solution vector will be in the range of indices from 1 to $n - 1$ and it will be initialized with the right hand side values.

In the `collection` structure below, we have placed the coefficient values directly in the element, and we have defined one parallel function *cyclicReduction*, which takes pointers to the two matrix coefficients. The method of element functions we need are a way to set the local coefficient values at the start of the algorithm, a function to do the forward elimination phase of the elimination and a method to do the "backsolve" phase.

```
Collection  DistributedTridiagonal: public DistributedVector{
     public:
        ....
        void cyclicReduction(ElementType *diagonal,
                             ElementType *offidagonal);
     MethodOfElement:
        ElementType *a;
        ElementType *b;
        virtual ElementType(ElementType *);
        void setCoeff(ElementType *diagonal,
                      ElementType *offdiagonal){
                         a = new ElementType(diagonal);
                         b = new ElementType(offdiagonal);
                      }
        void backSolve(int s);
        void forwardElimination(int s);
}
```

Note that the `setCoeff()` function requires a special constructor that takes a pointer to an `ElementType` object and allocates a new copy of the objected referenced. This is because the `forwardElimination()` function modifies the values of $*a$ and $*b$. Because the `collection` does not know about types of constructors that are available for the `ElementType` it must assume the existence of one.

The cyclic reduction function, shown below, divides the process into $log(n)$ parallel stages of forward elimination followed by $log(n)$ parallel stages of the backsolve procedure.

```
void DistributedTridiagonal::CyclicReduction(
                             ElementType *diagonal,
                             ElementType *offidagonal){
     int s;
     int n = this->dim1size-1;
     this->setCoeff(diagonal, offidaigonal);

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

The forward elimination is an element function that access neighbor elements s position in both directions. Mathematically, it is equivalent to the eliminating X_{i-s} and X_{i+s} from the middle equation in

$$\begin{array}{llllll} bX_{i-2s}+ & aX_{i-s}+ & bX_i & & & = this_{i-s} \\ & bX_{i-s}+ & aX_i+ & bX_{i+s} & & = this_i \\ & & bX_i+ & aX_{i+s}+ & bX_{i+2s} & = this_{i+s} \end{array}$$

Translating this to pC++, we have

```
void DistributedTridiagonal::forwardElimation(int n, int s){
     int k;
     ElementType *v1,*v2;
     ElementType c;

     if(s == 0){
         if(index == 0 || index == n)  *this = (ElementType) 0;
         return;
        }

     v1 = (ElementType *) ThisCollection->Get_CopyElem(index1 -s);
     v2 = (ElementType *) ThisCollection->Get_CopyElem(index1 +s);

     c = (*b)/(*a);
     *a = *a -c*2*(*b);
     *b = -c*(*b);
     *this += -c*(*v1 + *v2);
     if (!ThisCollection->Is_Local(index1 -s)) delete v1;
     if (!ThisCollection->Is_Local(index1 +s)) delete v2;
}
```

The backsolve step is equivalent to solving for X_i in

$$\begin{aligned} bX_{i-s} + aX_i + bX_{i+s} &= this_i \\ X_{i-s} &= this_{i-s} \\ X_{i+s} &= this_{i+s} \end{aligned}$$

and assigning the result to $this_i$.

```
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;
     }
```

It should be noted that, in general, this is not a very efficient algorithm because of the large cost of copying data from one processor to another.

A more interesting tridiagonal solver would be a blocked scheme where each element represents a set of rows of the system rather than a single row. Numerous techniques exist for this case, but we do not explore them here.

6.2 A Fast Poisson Solver

Consider the problem of solving the Poisson Equation on a square domain in two dimensions based on a "5-point", uniform grid, finite difference scheme. In other words, we seek the solution to the linear system

$$4U_{i,j} - U_{i-1,j} - U_{i+1,j} - U_{i,j-1} - U_{i,j+1} = F_{i,j}$$

with j, i in $[1...n-1]$, given $n = 2^k$ with boundary conditions

$$U_{0,i} = U_{i,0} = U_{i,n} = U_{n,i}$$

for i in $[0...n]$.

The algorithm we will use was first described in terms of parallel computation by Sameh and Kuck [15] consists of factoring the 5-point stencil, finite difference operator by first applying a *sin* transform to each column of the data. These transforms can all be applied in parallel. The result is a decoupled system of tridiagonal equations. Each tridiagonal equation involves only the data in one row of the grid. After solving this system of $n-1$ equations, a final *sin* transform along each column completes the solution. (see [4] for the mathematical details.)

The easiest way to do this is to view the array U as a distributed vector of column vectors,

```
DistributedVector< Vector >  U(.....);
```

The first and last steps of the parallel algorithm require a sine transforms operation on each column in parallel. Our Vector `class` has such a function and it can be applied as follows

```
U.sinTransform(wr,wi,p, &temp)
```

where wr and wi are special arrays of size $log(n)$ by n that must be initialized to hold the primitive n^{th} roots of unity, p is a "bit reversal" permutation and $temp$ is a temporary vector that is passed by the user so that the sin transform does not need to spend time allocating and deallocating the needed storage.

The middle stage of the solution process requires the solution to n systems of tridiagonal equations where the right-hand-side data is given by the rows of the distributed array. In the example above, we illustrated the solution of a single distributed system of tridiagonal equation using a `collection` of the form

```
DistributedTridiagonal<Element>  T(...);
```

where Element was a `class` that represented one component of the solution. All we required of Element was that the standard numerical operators (+, -, *, /, =) were well defined and that there was a zero assignment (`= (Element) 0`). Consequently, the `class` Element could also be Vector and, in this case the function, *cyclicReduction* is solving a vector of tridiagonal equations in parallel.

The main program program for the fast Poisson solver now takes the form shown below.

```
float *wr[LOGMAXVECSIZE], *wi[LOGMAXVECSIZE];
int p[MAXVECSIZE];

main(){
   Template Temp(NBPROC,&P,BLOCK);
   Align    A(GRIDSIZE,"[ALIGN(F[i],T[i])]");
   DistibutedTridiagonal<Vector> U(&Temp,&A,MAXVECSIZE);

   initialize(&U);

   n = MAXVECSIZE;
   // initialize coeff arrays wr, wi and p for FFTs
   log2n = mylog2(n);
   initw(n/2,log2n-1,wr,wi,p);

   poisson_solve(n,wr,wi,p, &U);
   }
```

The poisson solve function, shown below, need only initialize the coefficient arrays for the tridiagonal system. These coefficients correspond to the eigenvalues of tridiagonal system $[-1, 4, -1]$ which is one slice of the finite difference operator. Hence, it is easy to compute that the coefficients for k^{th} equation are

$$a_k = 4 - 2 * cos(\frac{\pi k}{n})$$
$$b_k = -1$$

```
void  poisson_solve(int n, float **wr, float **wi, int *p,
        DistributedTridiagonal<Vector> *U)
{
     int k;
     Vector a(n);
     Vector b(n);
     Vector temp(n):
     for(k = 1; k < n; k++){
            a[k] = 4.0 -2.0* (float) cos((double) pi*k/n);
            b[k] = -1.0;
            }
     U->sinTransform(wr,wi,p, &temp);
     U->cyclicReduction(&a,&b);
     U->sinTransform(wr,wi,p, &temp);
}
```

7 Conclusions

This paper presents the basics of an "object-parallel" programming language pC++. The focus has been on a complete discussion of the language extensions and the associated semantics as well as providing an introduction to the standard library of linear algebra collection classes.

We have not cited any performance results here. These are described in other papers [7], [9], [11], [13] and new results for various benchmarks will be available soon. The compiler currently runs on any standard scalar unix system as well as the Thinking Machines CM-5. A version for the Intel Paragon will be available by the middle of 1993. A group led by Jenq Kuen Lee at NTHU in Tiwan has ported a version of pC++ to the N-Cube. Other ports are planned and public releases of the software are planed starting in July of 1993.

The compiler takes the form of a C++ restructurer. This means the input language is pC++ and the output is standard C++ in the form needed to run on multicomputer systems. The `Kernel` class contains most of the machine specific details, so building new ports of the system is very easy. The compiler is written using the Sage++ compiler toolkit [12] which will also be distributed.

We expect that pC++ will evolve. The current version does not allow collections of collections and there is no support for heterogenous, networked parallel systems. Because we feel that both are essential for future applications, we will move in that direction. This may take the form of making pC++ a superset of the CC++ dialect proposed by Chandy and Kesselman at Cal Tech.

References

[1] T. von Eiken, D. Culler, S. Goldstein, K. Schauser, *Active Messages: a Mechanism for Integrated Communication and Computation*, Proc. 19th Int'l Symposium on Computer Architecture, Australia, May 1992.

[2] D. Culler, T. von Eiken, *CMAM - Introduction to CM-5 Active Message communication layer*, man page, CMAM distribution.

[3] A. Chien and W. Dally. *Concurrent Aggregates (CA)*, Proceedings of the Second ACM Sigplan Symposium on Principles & Practice of Parallel Programming, Seattle, Washington, March, 1990.

[4] R. Hockney and C. Jesshope, *Parallel Computers.* Adam Hilger Ltd., Bristol, 1981.

[5] High Performance Fortran Forum, *Draft High Performance Fortran Language Specification*, Version 0.4, Nov. 1992. Available from titan.cs.rice.edu by anonymous ftp.

[6] J. K. Lee, *Object Oriented Parallel Programming Paradigms and Environments For Supercomputers*, Ph.D. Thesis, Indiana University, Bloomington, Indiana, June 1992.

[7] J. K. Lee and D. Gannon, *Object Oriented Parallel Programming: Experiments and Results* Proceedings of Supercomputing 91 (Albuquerque, Nov.), IEEE Computer Society and ACM SIGARCH, 1991, pp. 273–282.

[8] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Thesis, Yale University, Sept. 1986

[9] D. Gannon and J. K. Lee, *Object Oriented Parallelism: pC++ Ideas and Experiments* Proceedings of 1991 Japan Society for Parallel Processing, pp. 13-23.

[10] D. Gannon, J. K. Lee, On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions, *Proceedings Conpar-Vap*, Lyon, Sept. 1992.

[11] D. Gannon, *Libraries and Tools for Object Parallel Programming*, Proceedings, CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, Sept., 1992 Saint Hilaire du Touvet, France. To appear: Springer-Verlag, lectures on computer science.

[12] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. Srinivas, *Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools*, Technical Report, Indiana University.

[13] F. Bodin, D. Gannon, *pC++ for Distributed Memory: CM-5 Communication Performance Experiments* Technical Report, Indiana University.

[14] Z. Lahjomri and T. Priol, *KOAN: a Shared Virtual Memory for the iPSC/2 hypercube* CONPAR/VAP, Sept. 1992.

[15] A. H. Sameh, S. C. Chen, and D. J. Kuck, *Parallel Poisson and Biharmonic Solvers*, Computing 17 (1976), 219-230