Object Oriented Parallel Programming Experiments and Results

Jenq Kuen Lee Dennis Gannon
Department of Computer Science
Indiana University
Bloomington, IN 47401

Abstract

We present an object-oriented, parallel programming paradigm, called the distributed collection model and an experimental language PC++ based on the model. In the distributed collection model, programmers can describe the data distribution of elements among processors to utilize memory locality and a collection construct is employed to build distributed structures. The model also supports the express of massive parallelism and a new mechanism for building hierarchies of abstractions. Our experiences with application programs in the PC++ programming environments as well as performance results are also described in the paper.

1 Introduction

Massively parallel systems consisting of thousands of processors offer huge aggregate computing power. Unfortunately, a new machine of this class is almost useless unless there is a reasonable mechanism for porting software to it so that the resulting code will both be efficient and scale well. FORTRAN 90 has been proposed as a standard for SIMD "data parallel" systems but it cannot be compiled well for distributed memory machines. Linda[2] is an excellent model for distributed MIMD processing, but it is not well suited to systems like the connection machine or the MasPar MP1. In this paper we describe a model of programming and an experimental programming language that we hope can be used on a variety of parallel architectures.

There are four major issues to be addressed to achieve scalability and portability. First, many of the MIMD machines available today use a collection of local memories to implement the global memory space. The access to global memory space will normally take $O(Log_2\ p)$ time where p is the number of processors, while the access to the processor's local memory will take only $O(1)$ time. Therefore the language should provide some means for programmers to utilize memory locality in their programs.

Second, algorithm designers tend to think in terms of synchronous operations on distributed data structures[4], such as arrays, sets, trees and so forth, but conventional languages provide no support for the efficient implementation of these abstractions in a complex memory hierarchy. In the distributed memory case, programmers must decompose each data structure into a collection of pieces each owned by a single processor. Furthermore, access to a remote component of the data must be accomplished through complex "send" and "receive" protocols. The global view of traditional data structures, such as matrices, grids and so on, becomes unreadable and complicated and decomposing all data structures this way leads to programs which can be extraordinary complicated.

Third, as is the case with large sequential programs, it must be possible to use a hierarchy of abstractions to relegate details to the proper level in a program[1]. By using hierarchy of abstractions in concurrent programming, it is easy to hide the low level implementation and to build libraries and reusable implementation of distributed programs.

Fourth, the language should allow programmers to specify massive parallelism. In an object-based model, programmers tend to think in terms of a distributed collection of elements. To initial a parallel action, an element method can be invoked by the collection which means that the corresponding method is applied to all elements of the collection simultaneously. This mechanism enables us to express massive parallelism and isolate the communication structure of the computation from the basic local computations.

The object-based model presented here, called *Distributed Collection Model*, will be shown to support all four of the points described above. We also de-

scribe a parallel C++ language, called PC++, with a *collection* construct as our first approximation to this model. We have implemented PC++ on a variety of machines including VAX8800, Alliant FX/8, Alliant FX/2800, and BBN GP1000 systems. Our experience with building application programs as well as experimental results on these machines are described in the paper.

The remainder of this paper is organized as follows. In section 2, we examine two programming paradigms that have formed the basis of our thinking: the *Concurrent Aggregate* model and the *Kali* programming language. We will look closely at these two languages to see how they support the issues described above and we will also examine them what is missing. Section 3 describes the *Distributed Collection Model.* Section 4 presents the parallel C++ language with the *collection* construct based on the distributed collection model. Section 5 describes several applications constructed by utilizing collection libraries. The experimental results are also presented. Finally, section 6 gives our conclusion.

2 Previous Models

Dally and Chien[1][3] have proposed an object-oriented language, *Concurrent Aggregate*, that allows programmers to build unserialized hierarchies of abstractions by using aggregates. Various distributed data structures such as trees, vectors and so forth can be built from aggregates. While we feel that this is an extremely powerful and elegant construction, there are still several things missing in CA. First, the model can not specify how the elements in an aggregate should be distributed among processors and the percentage of processor resources that should be involved in an aggregate. When the number of elements in an aggregate is significantly greater than the number of processors, simple and natural distributions specified by programmers often can best utilize memory locality resulting in optimal algorithms. The second shortcoming is that the model does not utilize the natural parallelism of an aggregate. As we will see later, the invocation of an element method by the aggregate can be used as a mechanism to explicitly control concurrency.

The Kali programming environment[4] is targeted to scientific applications. The fundamental goal of Kali is to allow programmers to build distributed data structures and treat distributed data structures as single objects. Kali thus provides a software layer supporting a global name space on distributed memory architectures. Because Kali is based on making the scientist trained in FORTRAN feel at home with parallel processing (indeed, a noble goal), it has had to forego the potential of more powerful data abstractions. Consequently, the only distributed data structure that Kali supports is the distributed array. While this may be sufficient for 90% of scientific codes, various distributed structures such as trees, sets, lists and so forth are not constructible in the language. Second, because Kali is a Fortran-like language and it does not support inheritance or other object oriented abstractions. Consequently, useful abstractions can not be inherited and reused in the construction of parallel programs. In spite of these limitations, we show that the basic ideas that the designers have introduced is very powerful and is completely extendible to an environment with more powerful data abstraction facilities.

SOS[8] is a distributed, object-oriented operating system based on the proxy principle. One of the goal of SOS is to offer an object management support layer common to object-oriented applications and languages. SOS also supports Fragmented Objects(FOs), i.e. objects whose representation spreads across multiple address spaces. Our assessment is that the SOS does not support many distribution patterns which are frequently used in the parallel programming. The creation of a distributed collection and the mapping of a global index to a local index may take longer time than necessary when supported by FOs.

3 Distributed Collection Model

The *Distributed Collection Model*, described below, is an object-based model for programmers to build distributed data structures. In the model, programmers can keep a global view of the whole data structure as an abstract data type and also specify how the global structures are distributed among different processors to exploit memory locality. The model also supports hierarchies of abstractions and thus it is easy to build libraries and reusable implementation of distributed programs. Massive parallelism is expressed by sending a message to the collection to invoke a method belonging to the class of the constituent element.

The major components in the distributed collection model are the collection, element, processor representatives, and distribution. A *collection* is a homogeneous collection of class *elements* which are grouped together and can be referenced by a single collection name. Each *collection* is also associated with a distribution and a set of active *computational representatives.* Another name for the active computational representative is *processor representative* which represents virtual processors at run time. The elements in

a collection will be distributed among the processor representatives. How the elements of the collection are distributed among computational representatives is described by the *distribution* associated with the collection.

A new idea involving hierarchies of abstractions is employed in the *Distributed Collection Model* in order to build useful abstractions. In the *Distributed Collection Model* not only can methods of collections be described and inherited by the other collection, but also the element class can inherit knowledge of the algebraic and geometric structure of the collection. Furthermore algebraic properties of the element class can be used by methods of the collection without detailed knowledge of implementation in the element class. For example, we can have a *DistributedList* Collection of which the basic element forms a *SemiGroup* by describing the general properties of a *SemiGroup* element in the Collection. We then can build a *ParallelPrefix* method for the collection based on the operator of the abstract *SemiGroup* element without knowing the specific details about element class. Consequently ParallelPrefix can be a generic library operator which can be applied to any user defined element class that has an associative operator. This ability of a collection to describe the abstraction of elements gives us great flexibilities in building libraries of powerful distributed data structures.

The computational behavior in the distributed collection model can be described as follows. Any message sent to an element of a collection is, by default, received, processed, computed, and replied to by the processor representative which owns the particular element. To initiate a parallel action, an element method can be invoked by the collection which means that the corresponding method is applied to all elements of the collection logically in parallel (with each processor representative applying the method to the elements that it owns.) If the number of elements of the collection is large relative to the number of processors, the compiler is free to use whatever technique is available to exploit any "parallel slack" to hide latency. A *DOALL* operator can also be used to send a message to a particular subset of the processor representatives in the distributed collection.

The model also supports a group of useful operators. The *reshape* operators support dynamic realignments of the distribution of collections at run time. The model supports operators for an element to refer to other elements in its neighborhood and methods to refer to an element of the whole collection.

4 The PC++ Language

In this section, we present a parallel C++ language with a *collection* construct that we use as a testbed for experimenting with algorithm descriptions and compiler optimizations. A set of built-in abstractions of distributed data structures and several examples are also presented.

4.1 Collection Constructs

The *Collection* construct is similar to class construct in the C++ language[5]. The syntax can be described as follows:

```
Collection <collection_name>:<inherited_collection>
          { method_list; };
```

To illustrate how a collection is used consider the following problem. Let us design a distributed collection which will be an abstraction for an indexed list (one dimensional array). We would like each list element to have two fields: *value* and *average*. We first define the collection structure.

```
Collection DistributedList : Kernel {
  MethodOfElement:
    self(...):
  Public:
    DistributedList(ProcSet,shape,distribution);
};
```

There are three points to observe here. First, the keyword *MethodOfElement* refers to the method *self()*. This specification means that *self()* is a method that can be used as part of the basic elements of the list to refer to the structure as a whole. (Methods declared this way are virtual and public by default.) The second point to consider is the super "collection class" called Kernel. This is a basic built-in collection with a number of special methods that are fundamental to the runtime system which will be described in greater details later. The third point to observe is the constructor method and its three parameters. We will postpone discussion of this briefly.

Next we must define the basic element of our distributed list. This is done as follows.

```
class element: ElementTypeOf DistributedList {
  int value, average;
  update(){
   average=1/2*(self(1)->value+self(-1)->value);}
};
```

The keyword *ElementTypeOf* identifies the distributed collection to which this element class will belong. As mentioned above the reason that the element needs such an identification is that we may inherit functions that identify the algebraic structure of

the collection. In this case, we have defined a simple method *update()* which makes the *average* of the current element equal to the numerical average of the *value* field of its two neighbors. Notice that *self()* is treated like a pointer (in the C sense) into our ordered list and that we can add offsets to access the field variables of sibling elements in the collection.

At this point we have not indicated how collections are distributed and what role the programmer plays in this. Following the mechanism in Kali, we break the process down into three components. First each collection must be associated with a number of processors which we view as indexed in some manner. Second we must describe the global shape of the collection and third, we must describe how the collection is partitioned over the processors.

In each case we use a special new C built-in class called a vector constant, or *vconstant* in short, to denote explicit vector values. Vector constants are delimited by "[" and "]". For example [14, 33] is a vector element of Z^2 whose first component is 14 and whose second component is 33. If we wish to declare a distributed list G to be viewed as being distributed over a one dimensional set of $MAXPROC$ processors and we wish the list to be indexed as a list of M elements which are distributed by a block scheme (each processor representative getting a sequential set of $M/MAXPROC$ elements) we invoke the collection constructor as follows

```
DistributedList<element> G([MAXPROC],[M],[Block]);
```

The common distribution keywords other than *Block* are *Whole* which means no distribution, *Cyclic* which is a round-robin rotation and *Random* which is random. To pursue this further, observe that if we wish to build a $[M, M]$ array that is distributed by blocks of rows to processors we can use declarations of the form

```
Darray<element> G1([MAXPROC],[M,M],[Block,Whole]);
Darray<element> G2([MAXPROC],[M,M],[Block,Whole]);
```

which means that processor representative 1 will get G1[1 .. (M/MAXPROC), 1 .. M] and G2[1 .. (M/ MAXPROC), 1 .. M] and representative 2 will get the next $M/MAXPROC$ rows, etc.

In the case like the one above when we have two conforming instances of the same distributed collection, we can use a form of data parallel expression that allows the structures to be treated as a whole. For example,

```
G1.average = G2.average + 2 * G1.average;
```

represents a parallel computation involving the *average* field of the corresponding elements of the respective collections.

4.2 Kernel Collection

The PC++ language begins with a primitive data structure called *Kernel* collection. The *Kernel* is the root of the hierarchy of collections. The hierarchy of collections derived from kernel is shown in Figure 1.

There are four arguments associated with the *Kernel* when we create a new instance of structure. They are the collection variant, the array of processor representatives, the size of the whole collection, and the distribution scheme. The collection variant can be FixedArray, GrowingList, or SynSet. The variant *FixedArray* means the whole collection can be seen as a distributed array. It is very similar to the conventional array except for that the elements of the collection are distributed among processors representatives. The variant *GrowingList* means the number of elements in the collection will grow dynamically from parallel phase to parallel phase. If the variant is either FixedArray or GrowingList, the elements are indexed and can be read or written through the indexes. Finally the variant *SynSet* means a set of unordered elements. The number of elements in a *SynSet* collection can be either increased or decreased at run time. This kind of structure is particularly useful to served as buffers in the producer/consumer problems or the state space of searching problems.

There are several important operators that are found in the kernel. A *DOALL* operator can be used to have a particular set of processor representatives execute a message concurrently. A *DoSubset* operator can be used to send a message to a set of elements in the collection.

4.3 Relations Between Collections and Elements

In this section, we describe the relations between collections and elements. Let's begin with the following definition.

Definition 1 A collection method is *primitive* if

1. All the usages of *ElementType* in the collection method do not require virtual element methods declared in the *MethodOfElement* regions.
2. It invokes only *primitive* collection methods in the current collection.

We classify the collection into two categories. One is called a *complete element collection* and the other is called a *field element collection.* Let E be a plain C++ object, and *Coll* be a collection, we call $Coll < E >$ a *field element collection.* If F is an object and declared as

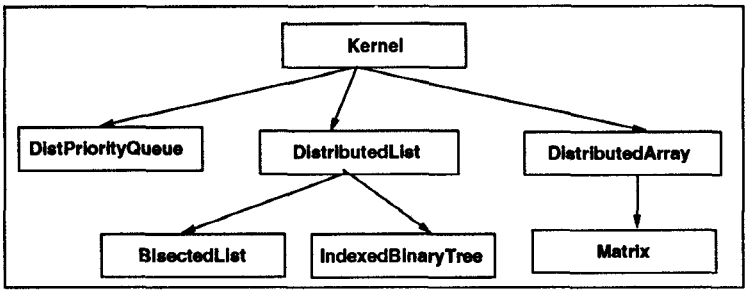

Figure 1: Collection Hierarchies of Distribution Objects

```
class F ElementTypeOf Coll { method_list  };
```

we called $Coll < F >$ a *complete element collection.* An element in a complete element collection will automatically inherit the virtual element methods and instances described in the collection it inherited and the element of a field element collection will not inherit any virtual instances and methods from the collection. Furthermore, a field element collection can only invoke the collection methods which are *primitive* as described in definition 1.

There are three ways that a field element collection can be formed. The first way is through the reference of the field of a complete element collection. For example, in Figure 2, W is a complete element collection of type $Sequence < element >$ and W.first_field forms a field element collection of type $Sequence < int >$. Second, a field element collection can be constructed by inheriting the shape from an existing collection instance. For example, W1 is constructed from W and of type $Sequence < int >$. Finally, we can construct the field element collection directly if the collection has a constructor which is primitive.

In summary, we have the following relations:

- If a class F is declared as *ElementTypeOf* a collection C_1, then it can not be declared as *ElementTypeOf* another collection C_2.
- Let E be a plain C++ class, then E can form different kinds of field element collections. For example, we can have $Coll_1 < E >$ and $Coll_2 < E >$ at the same time.
- A collection $Coll$ can form different complete element collections as well as field element collections. We can have $Coll < E_1 >$,$Coll < E_2 >$, and $Coll < E_3 >$ at the same time.

```
Collection Sequence : Kernel {
  Sequence(vconstant *P,vconstant *G, vconstant *D);
  Sequence(Sequence  &ExistingCollection);
  sorting();          // assume it is primitive
  ...
 };
class element ElementTypeOf Sequence{
  int  first_field, second_field;
  ...
};
Sequence<element>  W([MAXPROC],[N],[Cyclic]);
foo() {
 Sequence<int>    W1(W);
 (W.first_field).sorting();
 W1= W.first_field + W. second_field;
 W1.sorting();
}
```

Figure 2: Examples of Different Element Collections

4.4 Examples

A smoothing algorithm implemented by using a *DistributedArray* is shown below. Each of the arrays are distributed by [Block, Block] among processors. This is the Kali notation for partitioning the elements so that blocks of M/MAXPROC by N/MAXPROC elements are assigned to each processor representative.

The smoothing algorithm works by having an array of elements each does a local 5-point star relaxation and returns an array of values defined by the method update.

```
class E ElementTypeOf Darray {
    float  v;
    update(){
     return(4.0*v -(self(1,0)->v+self(-1,0)->v+
           self(0,1)->v+self(0,-1)->v));  }
};
Darray<E> A([MAXPROC,MAXPROC],[M,N],[Block,Block]);
Darray<E> W([MAXPROC,MAXPROC],[M,N],[Block,Block]);
smoothing() {
   for (i=1; i <MAX_ITERATION ;i++)
     { W->v = A->update();
       A->v = W->v;
```

```
    }
}
```

Next we will show a max-finding example. This example will illustrate two features of the language. The first feature is the ability of a collection to inherit algebraic structure. We will define a collection called *IndexedTree* in terms of the *DistributedList* collection defined earlier. An indexed tree is a tree where each node has an index to identify it. The root is index 1, the second level has indices 2 and 3, the third level has indices, 4,5,6,7 and, in general, the i^{th} level has indices $2^{(i-1)}$ through $2^i - 1$. The index will be set up properly inside the constructor *IndexedTree()*. This is done as follows.

```
Collection IndexedTree : DistributedList{
        IndexedTree();
     MethodOfElement:
        int  this_index;
        lchild() { return( self(this_index)); }
        rchild() { return( self(this_index+1)); }
}
```

We leave it to the reader to verify that this definition of the left and right child will give an ordering consistent with the numbering scheme described above. Our Max-Find will work by distributing the n elements to be searched in an indexed tree. Starting with the bottom level of the tree, each element will ask for the maximum found by its children.

```
class element : ElementTypeOf IndexedTree {
     float  v;
   public:
     float local_max(){
        if(v < lchild()->v) v = lchild()->v;
        if(v < rchild()->v) v = rchild()->v;
        return v;
        };
};
IndexedTree<element> X([MAXPROC], [N], [Block]);
float max_find(N){
 for(i = log2(N)-1; i > 0; i--)
   X.DoSubset([2**i:2**(i+1)-1:1],X.local_max);
 return(X[1].local_max());
 }
```

The *DoSubset* operator is the second feature illustrated in this example. The first parameter is the set of elements over which the second method is to be applied. In this case, our indexing scheme on the tree allows the tree levels to be described as a range of values.

5 Applications

5.1 Matrix Multiplication

In this section, we will show a simple matrix multiplication algorithm written in PC++ language. We will begin with the creation of a matrix collection as follows:

```
Collection matrix : DistributedArray {
      matrix();
      matmul(matrix *B1, matrix *C1);
  MethodOfElement:
     dotproduct(matrix *B, matrix *C);
  };
```

The matrix collection is derived from distributedarray and has a constructor matrix() with three arguments, the processor arrays, the size of global arrays, and the distribution schemes. The operator matmul is to multiply two given matrixes, B and C, and stores the result in the matrix which invokes the computation. This can be done by computing the dotproduct of the ith row of B and jth column of C for every element of which index is (i,j) in the current matrix. This is shown below:

```
matrix::matmul(matrix *B,matrix *C) {
      this->dotproduct(B,C);
}
```

One thing is worthy of notice is that the dotproduct() described in the collection is a virtual function of the element and will be overloaded when the real element is declared. The invocation of the virtual element method dotproduct() by the collection means the method is applied to all elements of matrix collection. We now show a basic element with a field name x below:

```
class elem  ElementTypeOf matrix {
    float x;
    // overload product
    friend elem operator *(elem *, elem *);
    elem operator +=( elem &); // overload +=
    dotproduct(matrix<elem> *B,matrix<elem> *C){
      int i,j,k,m;
      i= thisindex[0]; j= thisindex[1];
      m = B->GetSizeInDim(1);
      for (k=0; k< m; k++)
        *this += B(i,k)*C(k,j);
    }
}
```

We have used the power of C++ to overload the operator "*" and "+=" to make it easy to express the dotproduct in the usual manner. Because *elem* is declared as an element of matrix, it inherits *thisindex* from the distributedarray collection. *Thisindex* will

give the index of the current element. *operator ()* is also overloaded in the distributedarray for the access of a particular element given indexes.

Finally, we show the main program below:

```
matrix<elem>   A([MAXPROC],[M,M],[Block,Whole]);
matrix<elem>   B([MAXPROC],[M,M],[Block,Whole]);
matrix<elem>   C([MAXPROC],[M,M],[Block,Whole]);
main() {  A.matmul(&B,&C);  }
```

This version of matrix multiplication is known as a "point" algorithm, i.e., the basic element contains a single floating point value. The experienced parallel programmer knows that point parallel algorithm perform very poorly on most machines. The reason is that the total amount of communication relative to arithmetic is very high. Indeed, in our example, the references to $B(i,j)$ and $C(k,j)$ in the inner loop may involve interprocessor communication. If we have two interprocessor communications for a single scalar multiply and scalar add, you will not see very good performance. Taking this program almost as is (the only change involved substituting a function name for the operator because overloading is not completely supported at the time of this writing) and running it on the BBN GP1000 we see the problem with a point algorithm very clearly. Table 1 shows the performance of the matrix multiply in megaflops and the speed-up over the execution on one processor. The size of the matrix was 48 by 48. The speed-up reported in this paper is calculated with respect to the execution of PC++ programs on a single node. Fortunately there has been a substantial amount of recent work on blocked algorithms for matrix computation that we can take advantage of. If we replace our basic element in the matrix collection by a small matrix block, we can achieve a substantial improvement in performance. The only difference in the two codes is that we have a element of the form

```
class elem  ElementTypeOf matrix {
     float x[32][32];
     friend elem operator *(elem *, elem *);
     elem operator +=( elem &);
     dotproduct(matrix *B,matrix *C);
     ...
     };
```

The main difference is that the overloaded "*" operator is now written as a 32 by 32 subblock matrix multiply. Because this operator has a substantial amount more work associated with it than the original point operator, we now have enough parallel slack to mask both the latencies of communication and other system overhead. Currently the process of replacing the basic element in the collection by a matrix block is done by programmer. In the future we plan to have it done automatically by the compiler in most of the cases. We now list the GP1000 times in Table 2 for a matrix of size 1024 by 1024. Running this same blocked code on the 12 processor I860 based Alliant FX2800 we have the results in Table 3. It must be noted that we cheated slightly in reporting the speedup number for the FX2800. The actual speed of this code on one processor is 3.98 Mflops when run with the I860 cache enabled. Unfortunately, in concurrent mode the caches are all disabled so we have based the speedups on the one processor concurrent mode times.

Many readers may be concerned that the Mflop numbers reported above are very small relative to the actual peak rate of the machine. In fact, for the BBN GP1000, the node processor is the old M68020 which is not very fast. The numbers above are very good for that machine. For the I860 we see two problems. First the Alliant code generator is not yet mature and the cache problem must be fixed before we can get better numbers. However there is another solution to this problem. Because of the modular nature of the PC++ code it is very easy to use assembly coded kernels for operations such as the 32 by 32 block matrix multiply. To illustrate this point we have done exactly this for the old Alliant FX/8 with 4 processors. The results are shown in Table 4. These numbers are very near the peak speeds of 32 Mflops for this machine. We intend to build a library of hand coded assembly routines for the I860 processor to speed performance on standard collections running on the FX2800 and the Intel IPSC.

5.2 Summary of Performance Results

In this section, we present a summary of four application programs, including a conjugate gradient method, a fast poisson solver, a 0/1 knapsack algorithm, and a traveling salesman algorithm.

Our first example is a *Conjugate Gradient* method for a two dimension finite difference operator. We use a *Grid* collection to represent a simple M by M mesh on the unit square. Every element of the grid collection corresponds to one vertex in the grid. This vertex element contains one component of each of the solution, right hand side, and four auxiliary vectors needed by the CG algorithm. We also use a set of special boolean functions to tell the boundary conditions. The elements of the *Grid* collection also have a *localdot()* method. It is used to compute the inner product operation before doing a global sum over the entire Grid collection. Again as in the case of the matrix multiplication, we have a pointwise version and a blocked version. We show a blocked version below, where float_array is a C++ class that has all the basic arithmetic operators overloaded for "vector" operations. Table 5 and Table 6 show the performance

	p=1	p = 2	p = 3	p = 4	p = 6	p = 8	p = 12	p = 16
Mflops	0.0041	0.0073	0.0104	0.01298	0.01087	0.01435	0.0083	0.0081
Speed-up	1.0	1.78	2.54	3.165	2.65	3.5	2.02	1.97

Table 1: Pointwise Matrix Multiply of size 48 by 48 on the GP1000

	p=1	p = 2	p = 3	p = 4	p = 6	p = 8	p = 12	p = 16
Mflops	0.0775	0.1547	0.2319	0.3091	0.4627	0.6166	0.9137	1.21187
Speed-up	1.0	1.99	2.99	3.98	5.97	7.95	11.8	15.6

Table 2: Blocked Matrix Multiply of size 1024 by 1024 on the GP1000

of this algorithm on the GP1000 and Alliant FX2800 respectively for different block sizes and distribution strategies.

```
Collection Grid: DistributedArray{
   Grid(vconstant *P,vconstant *G,vconstant *D);
   double dotprod(int, int);
 MethodOfElement:
   int top_edge();
   int bottom_edge();
   int left_edge();
   int right_edge()
   double localdot(int, int);
};
class SubGrid ElementTypeOf Grid{
   float_array f[N][N], u[N][N],e[N][N];
   float_array p[N][N], q[N][N], r[N][N];
   void Ap();  // A*p
   double localdot(int, int);
};
```

Our second example is a fast poisson solver. We represent the grid arrays U and F each as a *linear array of vectors.* A *vector* is a special class with a number of builtin functions including FFTs and sine transforms, and *linear array* is a collection with a builtin function *CyclicReduction* to solve a tridiagonal system in parallel across the collection. We associate the columns of the grid with the vectors and the row dimension goes across the collection. The basic arithmetic operators "*", "+", "/", and "+=" in the *vect* class are used to overload the primitive operators in the *CyclicReduction* which solves a tridiagonal system. We have executed this code on both the alliant FX2800 and the BBN GP1000 and the results are shown in Table 7. The structures are shown below:

```
Collection LinearArray: DistributedArray{
   ...
   CyclicReduction(ElementType a,ElementType b);
};
class vector ElementTypeOf LinearArray{
   int n;        // size of vector.
   float *vals; // the actual data.
 public:
   vector();
   float &operator [](int i){ return vals[i]; }
   vector &operator =(vector);
   vector operator *(vector);
   vector operator +(vector);
   friend vector operator *(float, vector);
   void SinTransform(vector *result);
};
```

Our third examples is a 0/1 knapsack algorithm. The algorithm is based on the algorithm developped by Sahni[7] to solve one-dimensional 0/1 knapsack problems. The algorithm can be divided into three phases. In the first phase of the computation, we set up a distributed array G of m different objects and a distributed array S of #proc profit vectors. Next, each processor representative in the distributed collection G performs dynamic programming on its local collection $Knap(G_i, C)$ and generates a profit vector stored in the ith profit vector of S respectively. Finally, we combine the resulting profit vectors. Table 8 shows the experimental result with the number of objects to be 16*10124 and the capacity to be 400. The element and collection structures are shown below:

```
class  object ElementTypeOf Darray {
   int   weight,profit;
 };
class ProfitVect ElementTypeOf Darray {
   int  v[C+1];
   combine(int stride);
 };
 Darray<object>      G([MAXPROC],[m],[Block]);
 Darray<ProfitVect> S([MAXPROC],[MAXPROC],[Block]);
```

Our last example is that of a traveling salesman algorithm. The algorithm is based on LMSK algorithm[6]. We uses *DistributedPriorityQueue* collections from PC++ collection libraries to manage the searching state space. Every element of the collection will be a state in the state space tree. There are two

	p=1	p = 2	p = 3	p = 4	p = 6	p = 12
Mflops	2.26	4.26	6.48	8.56	12.78	22.81
Speed-up	1.0	1.88	2.86	3.78	5.65	10.09

Table 3: Blocked Matrix Multiply of size 1024 by 1024 on the FX2800

	p=1	p = 2	p = 3	p = 4
Mflops	7.91	15.69	23.11	30.74
Speed-up	1.0	1.98	2.92	3.88

Table 4: Blocked Matrix Multiply of size 1024 by 1024 on the FX/8 with assembly BLAS3

important operators, priority() and expand(), in the *state* element. *Priority()* is used to overload the virtual function described in the *DistPriorityQueue* to decide the priority of elements in the collection. *Expand()* selects a splitter and expands the current state into two states, each with smaller subsets of tours. The experimental result is shown in Table 9. The element structure is shown below:

```
class state ElementTypeOf DistPriorityQueue{
    int  lower_bound,rank, reduce_matrix[N][N];
  public:
    int priority() {return(lower_bound);}
    expand(DistPriorityQueue *Space_Queue);
};
```

Because of space limitations we were unable to go into many details about the examples described above. Interested readers should contact the author at leejenq @cs.indiana.edu for an extended version of this paper.

6 Conclusion

In this paper, we have presented an object-oriented, parallel programming paradigm and an experimental language PC++ based on the paradigm. We have also presented the experimental results. There are still many challenges remaining in the compiler optimization issues. The problems include optimizing the access functions for distributed collections, automatically replacing the basic element in the collection by a matrix block of elements, and the choice between locality and randomization. We are investigating these issues. In addition to that, we are building a rich set of distributed collections as libraries. This will be an important step toward a better parallel programming environment.

References

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

[2] Nicholas Carriero and David Gelernter. *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, April, 1989

[3] William Dally. *A VLSI Architecture for Concurrent Data Structures*, Kluwer Academic Publishers, Massachusetts, 1987

[4] Charles Koelbel, Piyush Mehrotra, John Van Rosendale. *Supporting Shared Data Structures on Distributed Memory Architectures*, Technical Report No. 90-7, Institute for Computer Applications in Science and Engineering, January 1990.

[5] Bjarne Stroustrup. *The C++ programming Language*, Addison Wesley, Reading, MA, 1986

[6] J. D. C. Little, K. Murty, D. Sweeney, and C. Karel. *An Algorithm for the Traveling Salesman Problem*, Operations Research, No 6, 1963.

[7] Jong Lee, Eugene Shragowitz, and Sartaj Sahni. *A Hypercube Algorithm for the 0/1 Knapsack Problem*, Proceedings of the 1987 International Conference on Parallel Processing, pp. 699-706, 1987.

[8] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot *SOS: An Object-Oriented Operating System - Assessment and Perspectives*, Computing Systems, Vol. 2, No 4, pp. 287-337, Fall 1989.

M=16,N=16, Block,Whole	p = 1	p = 2	p = 4	p = 8	p = 16
Time	83.15	42.68	21.85	11.69	8.14
Speed-Up	1.0	1.94	3.80	7.11	10.21
M=4,N=64, Block,Block	p = 1	p = 2	p = 4	p = 8	p = 16
Time	80.13	—	20.5	—	6.99
Speed-Up	1.0	—	3.9	—	11.46
M=4,N=128, Block,Block	p = 1	p = 2	p = 4	p = 8	p = 16
Time	****	—	85.06	—	28.67
Speed-Up	1.0	—	4.0*	—	11.9

Table 5: 20 CG Iterations on Grid of Size 256 by 256 on the BBN GP1000.

M=12,N=32, Block,Whole	p = 1	p = 2	p = 3	p = 4	p = 6	p = 12
Time	31.1	16.69	11.52	8.94	6.35	3.63
Speed-Up	1.0	1.86	2.69	3.47	4.89	8.56

Table 6: 384 CG Iterations on Grid of size 384 by 384 on the FX2800.

FX2800,	p = 1	p = 2	p = 3	p = 6	p = 12	
Time	7.51	3.85	2.63	1.40	0.723	
Speed-Up	1.0	1.95	2.86	5.36	10.38	
GP1000,	p = 1	p = 2	p = 3	p = 6	p = 12	p = 24
Time	70.48	35.66	24.03	12.65	7.48	6.36
Speed-Up	1.0	1.97	2.94	5.57	9.42	11.08

Table 7: Fast Poisson Solver on a 256 by 256 grid.

M=16k, Capacity=400		**p=1**	**p=2**	**p=4**	**p=8**	**p=16**
Alliant FX/8	time(seconds)	99.23	50.09	26.23	-	-
	speed-up	1	1.98	3.78	-	-
Alliant FX/2800	time(seconds)	12.58	6.39	3.30	1.79	-
	speed-up	1	1.97	3.81	7.02	-
BBN GP1000	time(seconds)	125.84	63.85	33.36	18.58	12.15
	speed-up	1	1.97	3.77	6.77	10.35

Table 8: 0/1 Knapsack Problem

Number Of City = 25		**p=1**	**p=2**	**p=4**	**p=8**	**p=12**
Alliant FX/8	time(seconds)	60.04	32.61	18.55	-	-
	speed-up	1	1.84	3.23	-	-
Alliant FX/2800	time(seconds)	3.29	1.90	1.16	0.96	1.08
	speed-up	1	1.73	2.83	3.40	3.02
BBN GP1000	time(seconds)	81.26	45.72	28.60	20.75	21.68
	speed-up	1	1.77	2.84	3.92	3.74

Table 9: Traveling Salesman Problem with 25 Cities