Libraries and Tools for Object Parallel Programming.

D. Gannon[a] [1]

[a]CICA and Dept. of Computer Science, Indiana University, Bloomington, Indiana 47401 USA

Abstract

This paper describes a large project to build a complete programming environment for "object parallel" computation. We define object parallelism as the extension of data parallel computation to the domain of object oriented software. In this model, collections of objects are distributed over processors in a manner consistent with HPF and Fortran90-D. Class methods are invoked on all objects in the collection concurrently. There is a rich hierarchy of "collection" classes which builds on the algebraic structures inherent in scientific applications.

1. Introduction

The proposed High Performance Fortran (HPF) standard represents a critical advance in software design for massively parallel (MP) computer systems. While largely motivated by the preponderance of multicomputer system designs in which each processor has private memory and all inter-thread communication is accomplished by message passing, HPF is also important to any non-uniform memory access system. HPF is Fortran 90 together with a system of annotations that enables the programmer to specify the way in which data structures are distributed throughout the machine. The idea is not only simple and elegant, it will be supported by all the MP vendors.

HPF is, by design, a conservative approach to the problem of how to involve the programmer in the task of parallelization. Consequently, it is limited in its range of applications to simple array data structures and dynamic or linked structures are not considered. (These limitations were essential to getting an agreed upon subset of annotations specified that all vendors could implement in a short time frame.)

However, as applications become more complex they tend to use more complex data structures. A growing number of scientific and engineering programmers have turned to languages like C++ to build these applications because of the superior support provided for data abstractions, [6], [5]. In addition, a style of parallel programming in C++ is emerging that encapsulates parallelism within a class hierarchy and allows the user to work with mathematical structures that are closer to the application [3], [2], [4]. In this paper we will discuss the extension of the HPF ideas to a language like C++. We will argue that data distribution annotations are, by themselves, not sufficient and present a simple extension to C++ that allows an "Object Parallel" data style similar in spirit to

[1]Research Supported by NSF grant ASC-9111616 and DARPA

concurrent aggregates concept of Dally and Chien [9]. We will also discuss some of the compiler issues and a strategy for using the class hierarchy to aid in the animation of algorithm and performance behavior.

2. C++ Programming Using HPF Data Distribution Directive Style

A key idea in High Performance Fortran is to use data distribution directives to guide the compiler to an efficient generation of SPMD code for MP systems. In order to avoid confusion with the official HPF standard, which has not yet been published, we will only discuss the application of HPF-style data distributions ideas to C++ in this paper. (The interested reader can refer to the public HPF working notes from Rice University which are available by anonymous ftp from titan.cs.rice.edu. The C++ implementation described here will be made consistent with the official HPF model after the committee has finished their work.) It should also be noted that the ideas below are also found in the Kali language [8], Fortran-D [7] and Vienna Fortran [13].

The process of mapping data arrays to processors is seen as a two stage process: Arrays are *aligned* to *templates* and templates are *distributed* over *processors* . For example, consider the trivial problem multiplying an n by n matrix A time a vector x of length n using p processors. A naive solution would be to partition the the matrix into p blocks of n/p rows each of size n and partition the operand vector x and the result vector y into p blocks of size n/p. The code and annotations to do this is given below.

```
/* MATRIX*VECTOR Version 1.  Row-Blocked    */
float A[n][n];
float x[n], y[n];
/*annotation( template T[n] )              */
/*annotation( align A[:][*] with T[:])     */
/*annotation( align x[:], y[:] with T[:])  */
/*annotation( processors P[p] )            */
/*annotation( distribute T over P by(Block))*/

       for(i = 0; i < n; i++)
           for(j = 0; j < n; j++)
               y[i] += A[i][j]*x[j];
```

The abstract template provides a target for array alignment and the processors annotation gives an abstraction of virtual processors. The distribution directive specifies a function that binds nodes of the template to the virtual processors. The "Block" distribution means that the first n/p template nodes map to the first virtual processor, the second n/p nodes map to the second virtual processor node, etc.

The parallelism is derived by an analysis of the code using the "owner computes" rule which states that if a modification to a distributed data item is executed, then it will be executed by the processor that "owns" (via the distribution mappings) that data item. In the case at hand, this means the outer loop will be parallelized. The k^{th} processor will execute

```
for(i = (n/p)*k; i < (n/p)*(k+1); i++)
    for(j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
```

The only communication required is a copy of x to each processor. However, as observed by Schreiber in [12], this is version of the program is communication inefficient, i.e. as the number of processors are scaled the time spent in communication will dominate, because it does not decrease as p increases.

As advocated by Schreiber, a better scheme is to use a blocked partition of the array into squares $n/p^{1/2}$ on a side. Unfortunately, it is nearly impossible to force the compiler to generate a parallel version of the program that achieves the optimal communication efficiency if the *owner computes* rule is to be respected. (The problem is that the vector y is modified, so the computation is based on how y is partitioned and not on how A is blocked.) Consequently, to achieve the best algorithm one must modify the program and introduce a temporary two dimensional array which can be the target of the compilers code generation strategy. The code is shown below.

```
/* MATRIX*VECTOR Version 1.  Square-Blocked    */
#define q sqrt(p)
float A[n][n];
float x[n], y[n], tmp[n][q];
/*annotation( template T[n][n], T2[n][q])      */
/*annotation( align A[:][:]  with T[:][:])    */
/*annotation( align x[:]     with T[0][:])    */
/*annotation( align y[:]     with T[:][0])    */
/*annotation( align tmp with T2)              */
/*annotation( processors P[q][q] )             */
/*annotation( distribute T  over P by(Block, Block))*/
/*annotation( distribute T2 over P by(Block, Block))*/


       for(s = 0; s < q; s++)       // parallel
           for(i = 0; i < n; i++)       // parallel
              for(j = s*(n/q); j < (s+1)*(n/q); j++)
                 tmp[i][s] += A[i][j]*x[j];

       for(i = 0; i < n; i++)       // parallel
           for(j = 0; j < q; j++)
                 y[i] += tmp[i][j];
```

With $q = p^{1/2}$, the array x is aligned with the first row of the template and y is mapped to the first column. The array *tmp* is used to contain the results of local matrix vector products of the n/q by n/q matrix blocks times the operand vector x.

The critical observation to make about this program is that we have had to modify the *source code* in addition to adding annotations in order to coax the compiler into recognizing a communications efficient assignment of work to processors. Consequently, we cannot view the process of inserting HFP style distribution annotations as sufficient for achieving high performance. This fact should be of no surprise; the processes of parallelization is not independent of algorithm design.

3. Object Parallelism

In spite of the awkward example from the preceding section, the Kali/Fortran-D/HPF model is very powerful and works well in many situations. Furthermore, there is an obvious extension to array distributions in C and C++. One very interesting extension is to allow the concurrent evaluation of an class method on an array of objects. For example, let C be a class and X be an array of objects of that class.

```
class C{
     .....
     public:
       void f(...);
       ...
   };

C X[100][100];

/* annotation( template T[100][100] )  */
/* annotation( align X[:][:] with T[:][:] */

     for(i = 0; i < 100; i++)
        for(j = 0; j < 100; j++)
            X[i][j].f(...);
```

The parallelization of this can mean the concurrent application of the function $f()$ to every object in the array. Because of the binding of $f()$ to the object, the owner computes rule extends nicely to object oriented program. In other words, the distribution directives allows objects to be assigned to processor which then *own* the objects. Invocation of an object method means that the processor that owns the object is responsible for executing the method function.

Another way to achieve parallelism in C++ is to allow range expressions by overloading the colon operator ":" and the index operator [...] so that we may use expressions of the form

```
X[0:99][0:99].f(...);
```

to mean the Fortran90-like array expression parallel form of the loop above. Unfortunately, we cannot overload the [...] operator in this way for arrays in C++ unless we define a special class. Even if we do this, the overloading of [...] is restricted in a way that would prohibit this extension.

In principle it is not hard to adapt the HPF annotations to the C and C++ environment. As described above, there are problems with the extension of Fortran 90 style vector expression syntax, but this is a minor issue. Of greater significant are the problems associated with pointers. For example, given the specification of a two dimensional array as a vector of pointers to rows of objects, as is often done in C programs,

```
C *X[100];

/* annotation( template T[100] )          */
/* annotation( align X[:]  with T[:]     */
/* annotation( processors P[100] )        */
/* annotation( distribute T[:] over P[:] by (Block) */
....
   for(i = 0; i < n; i++)
         X[i] = new C[100];
```

we are saying that the array of pointers is distributed, but this does not say anything about the objects being referenced. If we assume that the *new* operator allocates within the same memory domain as the pointer to which it is being assigned it may be possible to build a consistent semantics. A more serious problem is the realignment of the one dimensional array of one dimension arrays over a two dimensional template. It is our feeling that this beyond the scope of the intent of the HPF directives. (Similar problems exist in Fortran 90 which allows array subsections to be described by pointers which can then be passed to subroutines.)

4. Concurrent Aggregates: Collection Classes

A more challenging exercise than specifying a C extension to the semantics of HPF annotations is to see how the concept of data distributions can be incorporated into the object oriented features of C++. In this section we show that by extending the C++ language with a special type of concurrent aggregate class [9] we can generalize the ideas embodied in HPF.

Our C++ extension is called pC++ and it is based on adding one new type of special class called a *Collection* to the language. A Collection is a structured set of objects from a given class. Collections represent the mathematical abstractions that we associate with distributed aggregates. For example, the library collection type *DistributedArray* is the pC++ analog of the Fortran 90 array. Associated with DistributedArray are the standard intrinsics associated with Fortran 90 array operations including element-by-element arithmetic operations such as addition, multiplication and basic mathematic operations as well as reductions and other parallel operators that involve the structure of the array. For example, the Fortran 90 array operations below

```
   real, dimension(100,100):: x,y
   real, dimension(100)::cf
   cf = sum(x*cos(y), dim=1)
```

say that cf is the vector of values obtained by first taking the pointwise product of each element of x with the cosine of the corresponding element y and then summing along the first dimension. The same expression in pC++ can be written as

```
DistributedArray<float> x(..[100,100], ...);
DistributedArray<float> y(..[100,100], ...);
DistributedArray<float> cf(..[100], ...);
cf = (x*y.Cos()).sumReduce(0);
```

Notice that the constructor for the arrays above resembles of a C++ template declaration. A DistributedArray is an indexed collection. the parameter between the angle brackets specifies the type of the object that sits at each position in the array. Also note that the constructor parameters have not been fully specified. To complete the specification of a DistributedArray we need a *TemplateClass* and alignment functions where by TemplateClass we mean the abstract hpf template (which is not to be confused with the C++ template construct!)

A TemplateClass specifies an abstract grid of positions, an array of virtual processors and a distribution function that maps the grid to the processor array. In other words, the template class corresponds to the second stage of the HPF mapping. The syntax for its constructor takes the form

```
TemplateClass T( <size of target-template>,
                 < processor-array>,
                 <distribution-function>);
```

The size and processor-array specifications are denoted by *vector constant* expressions of the form

```
[3, 4] or [p] or [3*q, 99, 99/p], etc.
```

The distribution function is the same as the HPF mappings:

```
[Block, Cyclic, Whole]
```

where $Whole$ is used for the HPF * distribution.

Collections are created with a constructor of the form shown below.

```
CollectionClass<element-type> C(TemplateClass T,
                                  <collection-size>,
                                  <align-function> )(<element-constructor>);
```

The format of the collection size specification depends on the nature and shape of the collection. The alignment functions specify a mapping between the elements of the collection and the positions on the template grid.

The primary generalization that collections make over HPF distributed arrays is that collections can define arbitrary topologies and mathematical structures. As we shall see in the remainder of this paper, the variety of shapes and forms of distributed concurrent aggregates is, indeed, very large.

The syntax of a collection declaration is given by

```
Collection collection_name: inherited_collection{
    \\ constant data and private functions.
  public:
     < Aggregate functions and operators >
  MethodOfElement:
    <Element type functions and operators >
};
```

All non-element data associated with a collection must be constant as it is duplicated on each processor and is visible to each element. This is because the philosophy behind the collection concept is that real variable data is all contained in the elements.

There are two types of method functions associated with the collection. The first are the public definition global functions. These include

- . A constructor for the collection and a destructor if needed.
- A list of method functions to access elements of the collection from the global thread of computation. Examples include the index operators like [ ].
- A list of parallel operators and functions that operate on the collection as a whole. Examples include reduction, parallel prefix,broadcasts and parallel matrix algorithms.
- A list of graphical and instrumentation hooks that simplify the task of doing program algorithm animation and performance analysis.

The second class of methods are called *MethodOfElement* functions. There are two types of these.

- A list of method functions and operators that are required of the element type for the collection operators to work correctly. For example, the sum reduction operator assumes that each element must have an associated sum operator defined. These element requirements are listed here to satisfy the strong typing of C++.
- A list of method functions that can be called by the elements to access information about the global structure of the collection. Examples of this type of function include functions to access the neighbors in a grid, the global index of an element in an array or the children or parent in a tree.

The principle parallel operation associated with a collection is the concurrent evaluation of element method functions. For example, to write a simple Jacobi iteration on a two dimensional square grid, one can use a Grid collection and define an element class as a vertex as follows.

```
class vertex  ElementTypeOf Grid{
   public:
     float f, oldx, newx;
     float relax(float lambda){
return f - lambda*(4.0*oldx + self(0,1).oldx +
                   self(1,0).oldx + self(0.-1).oldx + self(-1,0).oldx);
        };
```

The vertex class is said to be of *ElementTypeOf* Grid. This implies that the vertex can use the MethodOfElement functions from the Grid collection class. One of these is the *self(i,j)* function which returns a reference to the neighbor offset by (i,j) of the calling element. A parallel Jacobi style iteration can now be written in the form shown below.

```
Grid<vertex> G(.....);
  while(...){
       G.newx = G.relax(0.01);
       G.oldx = G.newx;
       }
```

The concurrency comes in the invocation of relax() on all elements of the collection in parallel and in the parallel assignment. Synchronization is accomplished by an implied barrier at the end of each collection operation.

Inheritance plays a major role in the definition of collections. The basic collection library, shown in Figure 1, contains many of global structures that are encountered in scientific and engineering applications. As the library develops further, we expect this hierarchy to grow. An essential feature of its design is user extensibility. At each level in the hierarchy users should be able to "subclass" new collection types as needed for special applications. In the following sections we will discuss the individual subclasses in greater detail.

4.1. Matrix Vector Revisited

To illustrate the ideas presented so far we will return to the matrix vector product example from section 2. Because we are dealing with mathematical structures related to linear algebra we will start with a basic distributed matrix collection which can be viewed as a distributed array with the matrix algebraic abstractions. The key definitions for the matrix time vector operation are given below.

```
Collection DistributedMatrix: DistributedArray{
     // TemplateClass T; inherited from Distributed Array
     // int size;        inherited from Distributed Array
   public:
     ...
     DistributedVector<Vtype> operator *(DistributedVector<Vtype> x){
        DistributedArray<Vtype> tmp(T, [size,size], [i,j]->[i,j]);
        tmp.duplicate(x, 0);  /* spread x over the rows of tmp */
        tmp = (*this)*tmp;    /* elementwise parallelism       */
        return tmp.sumReduce(1);          /* row wise reduction */
```

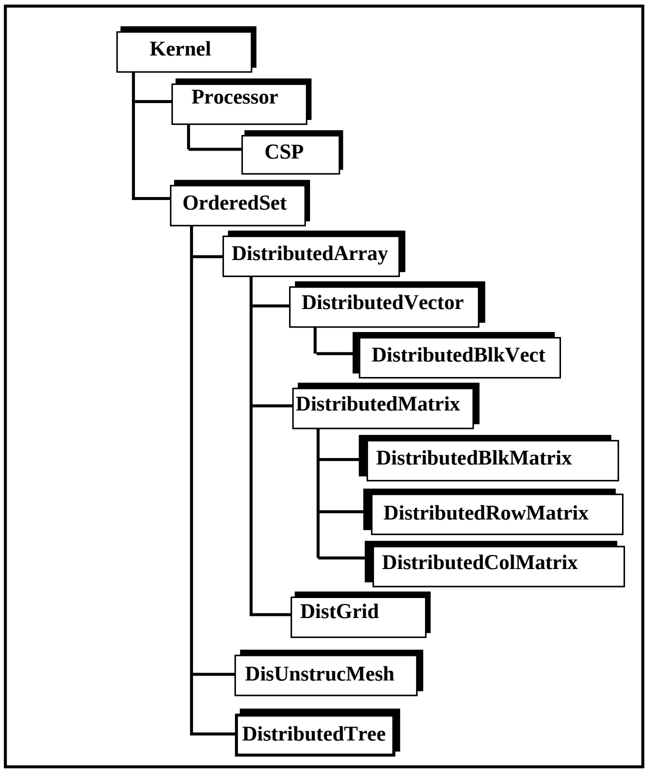

Figure 1. Partial Collection Library Inheritance Tree.

```
        }
     ...
}
```

The matrix-vector product operator shown here is very simple. It works by making a temporary distributed array which holds a copy of x in each row. This is accomplished by using library functions associated with the DistributedArray collection type. The multiplications are then done in parallel by invoking the parallel array multiply operator from the distributed array class. The products are then added together along rows to form the final result.

The *Row Blocked Version* of the program now takes the form

```
TemplateClass T( [n,n], [p], [Block, Whole]);
DistributedMatrix<float> M( T, [n,n], [i,j]->[i,j]);
DistributedVector<float> x( T, [n], [i]->[i,0]);
DistributedVector<float> y( T, [n], [i]->[0,i]);

   y = M*x;
```

This is completely equivalent to the first HPF example in section 2.

The *Square Blocked Version* of the program can be written as

```
TemplateClass T( [n,n], [q,q], [Block, Block]);
DistributedMatrix<float> M( T, [n,n], [i,j]->[i,j]);
DistributedVector<float> x( T, [n], [i]->[i,0]);
DistributedVector<float> y( T, [n], [i]->[0,i]);

   y = M*x;
```

This is completely equivalent to the second HPF version in section 2, but with the difference that we did not have to change the source code. This is because the matrix-vector product operator was written as a parallel algorithm to begin with.

Note, however, that both versions of this program contain a potential inefficiency. The multiply algorithm specifies that the vector x is to be expanded to a n by n distributed array by duplication from the first row of the template. Clearly, only one copy is needed in each processor. An alternative formulation that makes this optimization explicit would make the template array the same size as the processor array and specify that the elements of the matrix are from a class *float_DenseMatrix* and are of size n/q by n/q. In other words the matrix is now a matrix of matrices. Similarly, we make the vectors each vectors of vectors of length n/q. Because the size of the matrix is now identical to the processor array, each processor will only get one copy of the vector segment it needs. The code for this version of the matrix vector computation now takes the form shown below.

```
TemplateClass T( [q,q], [q,q], Ident);
DistributedBlkMatrix<float_DenseMatrix>
                      M( T, n, [q,q], [i,j]->[i,j])(n/q, n/q);
DistributedBlkVector<float_Vector> x( T, n, [q], [i]->[i,0])(n/q);
DistributedBlkVector<float_Vector> y( T, n, [q], [i]->[0,i])(n/q);

   y = M*x;
```

The only difference between a DistributedBlkMatrix and a DistributedMatrix is in the way the global indexing function works. (The blocked matrix is only of size q by q but it should be indexed as an matrix of size n by n.) The matrix times vector code is identical (inherited) from the the DistributedMatrix class.

There are two advantages of the blocked scheme shown above. First, the elementwise operations in the matrix multiply step,

```
        tmp = (*this)*tmp;    /* elementwise parallelism       */
```

are now products of an object of class *float_DenseMatrix* time an object of class *float_Vector*. These are each scalar operation and can be coded as level-3 BLAS finely tuned for peak performance on each processor of the system. The performance improvement can be dramatic as shown in [10].

The second advantage of this scheme is that we now have a way to work with distributed sparse matrices at very little cost. Assume you have a sequential class *float_SparseMatrix* for which sparse matrix operations, such as matrix-vector products, have been defined. By changing one line of the program

```
DistributedBlkMatrix<float_SparseMatrix> M( T,n,[q,q],[i,j]->[i,j])
                                            (n/q, n/q);
```

our dense matrix program now can function with sparse matrices. The resulting distributed data structure is illustrated in Figure 2. Of course, many algorithms appropriate for dense systems may not be optimal for sparse matrices, but there are many cases where the best algorithms are the same. For example, the NAS SparseCG benchmark uses a classical conjugate gradient algorithm and results from an implementation based on the idea above are illustrated in [11].

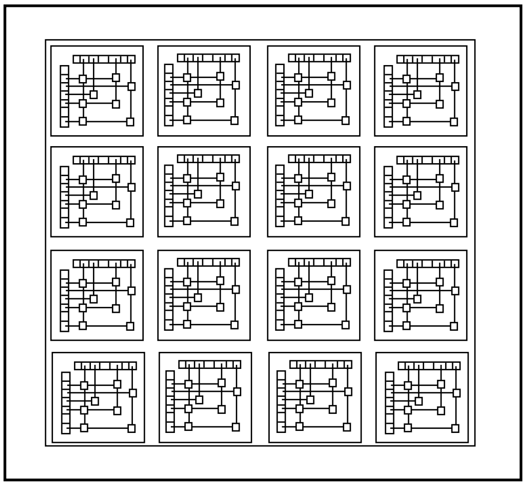

Figure 2. A Distributed Matrix of Sparse Matricies.

5. The Real Challenge: Dynamic Structures

The collection hierarchy described in section 3 includes a number of data types other than matrices. For large scale simulations, meshes or grids are the critical data structure. The most challenging problems involve grids that are locally refined and dynamic in nature. For example, Multigrid algorithms and various adaptive mesh refinement techniques are becoming increasingly important in problems where the simulation must consider multiple length scales. Illustrated in Figure 3, these grids are linked together in a tree structure. Algorithms that operation on these structures typically work on only one level at a time and the structure of the tree is dynamic. In the following paragraphs we sketch a possible implementation in terms of the collection hierarchy. Unlike the discussion in the preceding sections the ideas here are all currently unimplemented. At a later date we will have final implementation details and experimental results.

If we begin by assuming that each mesh cell is described by a C++ class *cell* the problem is reduced to defining the global topology. The structure is composed of a tree of meshes of cells, so one possibility is to define an *AMR_Grid* as

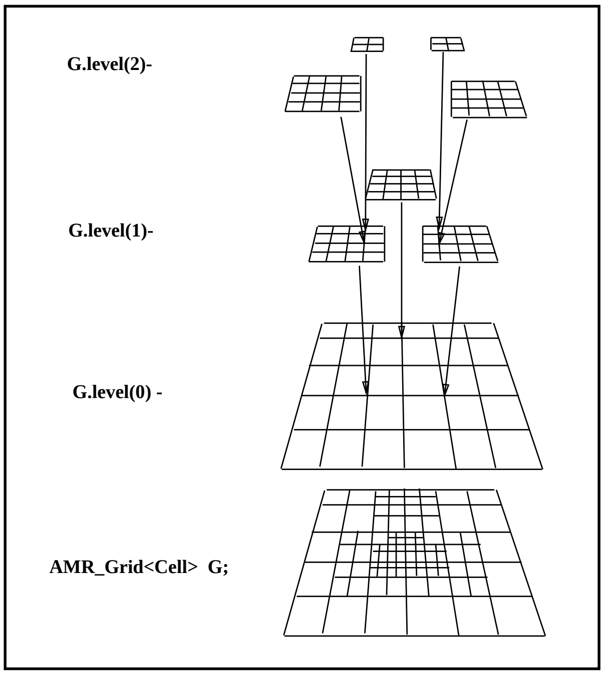

Figure 3. Adaptive Mesh Refinement Collection.

```
typedef   DistributedTree<DistributedGrid<Cell>> AMR_Grid;
```

however this may not be perfectly suited to algorithms that sequentially go through the levels of the tree and treat each level as set of grids that may be processed concurrently. An alternative formulation is

```
typedef  OrderedSet<DistributedGrid<Cell>>  Level;

Class AMR_Grid{
     Level L[MaxLevel];
     ...
};
```

Both of these schemes use a type of nested concurrent structure to represent the tree of meshes. Other than the fact that this is not yet implemented in our compiler, there is nothing inherently wrong with the idea. The primary difficulty with these schemes that of defining what processor alignment and distribution means in this case. This is an open research problem.

A slightly easier problem to work on is the adaptive Barnes-Hut or multipole algorithms used in N-body computations. While closely related to the AMR problem above (see Edelsohn [1] for an interesting analysis), it also represents a different use of an adaptive tree. In the Barnes-Hut algorithm, space is initially partitioned in a manner similar to that in Figure 4 and a tree that represents a quadtree recursive partitioning of the domain is constructed. The majority of work in the algorithm involves traversals of this tree.

A parallel implementation requires two data structures. One is a vector of particles

```
DistributedVector<particle>  particles;
```

and the other is a dynamic tree of nodes which represent either region summary information or references to particles.

```
DistributedTree<Node>  BH_Tree;
```

The most important part of this computation is the design of the distributed tree collection. It is essential that the tree be constructed in such a way that particles that are mapped to a given processor are associated with leaves in the distributed tree that are resident on that processor. This will allow us to exploit spatial locality and optimize the tree traversal routines.

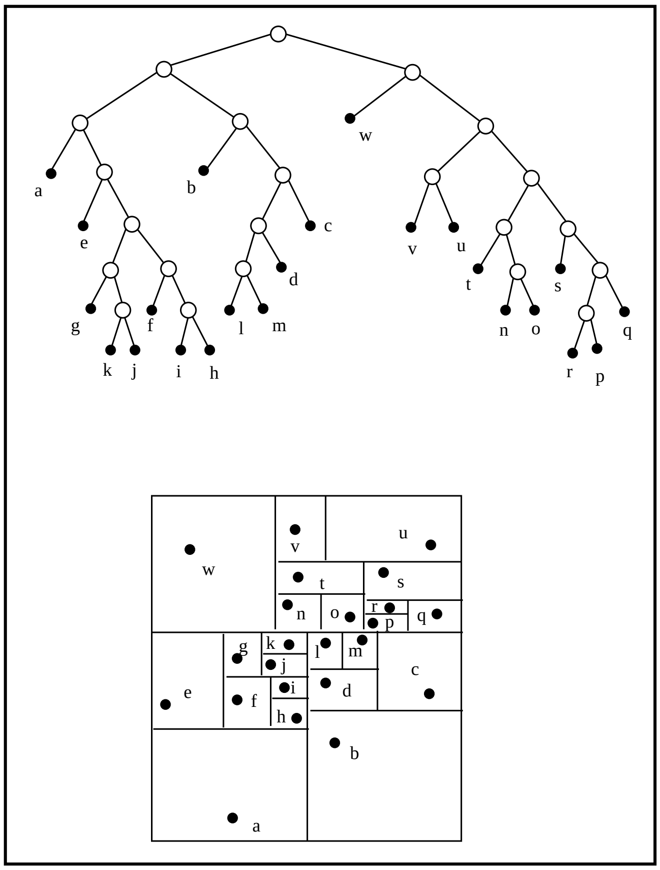

Figure 4. A Typical Barnes-Hut Data Structure for N-Body Problems.

6. Visualization Issues

One of the principle tools required by programmers when tuning applications for high performance on massively parallel systems is a mechanism for understanding how performance relates to algorithm and data structure design. The current state of the art is to monitor message traffic between processors and use graphical techniques to debug algorithm and its performance.

We feel that, as data structures become more complex, it becomes increasingly difficult to use a processor-message based model. In particular, applications where multiple thread of computation are bound to a single processor, one wants to see inner-thread communications. This situation can occur when a data distribution function maps multiple objects to a single processor and the compiler cannot merge these into a single thread. Furthermore, in the case of very complex structures like the AMR computations described in the last section, the user is more interested in how the communications relate to a graphical display of the data structures.

There are a variety of ways of dealing with this problem, but the object oriented features of C++ provide a unique set of opportunities. Indeed, graphical display software is one area where object oriented techniques are now the standard practice.

Our proposal is to instrument the basic collection hierarchy with built-in measurement and visualization software. A user that builds with or subclasses from the hierarchy will inherit the built-in visualization routines. In this way, standard graphics routines can be implement for graphs, trees and grids that are best suited to these applications. If the user does not like the standard displays, it is easy to override the default routines. While this idea is as yet untested, we feel the potential for this approach is very great.

7. Conclusion

In this paper we have sketched the extension of HPF style data distributions and data parallel programming to a type of object parallel programming. By adding a special concurrent aggregate class to C++ we have shown how linear algebra computations are easily described. In addition, we have illustrated how libraries of collection types can be created and how they can be used in a variety of complex applications.

A preliminary version of the pC++ compiler is now running on several shared address space machines. These include the BBN GP1000, the BBN TC2000, the Alliant FX/8 and FX/2800. Many of the ideas described in this paper represent new directions for pC++ that will be implemented in the next version of the compiler. Our initial targets will be the TMC CM-5 and the Intel Paragon. We expect to release a public domain version of the compiler in the Fall of 1993. This compiler will actually be a preprocessor that generates C++ or C code that executes in a SPMD model. At the same time we will release an initial version of our collection hierarchy library.

8. REFERENCES

1 D. Edelsohn, Hierarchial Tree-Structures as Adaptive Meshes, *Syracuse Center for Computational Science, Technical Report, CRPC-TR91186/SCCS-193*, Nov. 1991.

2 D. Balsara, M. Lemke, D. Quinlan, AMR++, A C++ Object Oriented Class Library for Parallel Adaptive Mesh Refinement Fluid Dynamics Applications. *Technical Report, Univ. of Colorado Denver*, 1992.

3 M. Lemke, D. Quinlan, P++, a C++ Virtual Shared Grid Based Programming Environment for Architecture Independent Development of Structured Grid Applications, *Technical Report 611, Gesellschaft fur Mathematik und Datenverarbeitung MBH*, Feb. 1992.

4 W. Bain, Aggregate distributed Objects for Distributed Memory Parallel Systems, *Proceedings DMCC V, 1990*

5 J. Peery, K. Budge, A. Robinsan, D. Whitney, Using C++ as a Scientific Programming Language, Report Sandia National Laboratories, Albuquerque, N.M. 1991.

6 R. Schoenberg, M++, An Array Language Extension to C++, Dyad Software Corp. Renton, Wa, 1991.

7 G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, M.-Y. Wu, FORTRAN D Language Specification, *NPAC Technical Report, Syracuse University*, 1991.

8 C. Koelbel, P. Mehrotra, J. V. Rosendale. *Supporting Shared Data Structures on Distributed Memory Architectures*, Technical Report No. 90-7, Institute for Computer Applications in Science and Engineering, January 1990.

9 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.

10 J. K. Lee and D. Gannon, *Object Oriented Parallel Programming: PC++*, Proceedings of Supercomputing 91 (Albuquerque, Nov.), IEEE Computer Sciety and ACM SIGARCH, 1991, pp. 273–282.

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

12 R. Schreiber, Scalability of Sparse Direct Solvers. In J. A. George, J. R. Gilbert, and J. Liu, editors, *Sparse Matrix and Graph Algorithms*, IMA, University of Minnesota, 1992.

13 B. Chapman, and P. Mehrotra, and H. Zimma, Vienna Fortran - A Fortran Language Extension for Distributed Memory Multiprocessors, *ICASE Technical Report 91-72*, 1991.