OBJECT-ORIENTED METHODS FOR PARALLEL EXECUTION OF ASTROPHYSICS SIMULATIONS *

Dennis Gannon, Shelby X. Yang, Paul Bode,
Vladimir Meñkov, and Suresh Srinivas
Department of Computer Science, Indiana University
Bloomington, Indiana 47405, U.S.A.

1 INTRODUCTION

Object-oriented technology now forms the basis for a very large part of the software development industry. This is especially true in the areas of computer graphics, data base, graphical interface design and modern operating systems. We are also beginning to see the emergence of object-oriented methodology in scientific programming.[1] For parallel programming of scientific applications, there exist a number of new tools to help build portable, object-oriented software. Most of them are extensions to C++, the dominant object-oriented programming language. They include COOL [3], CC++ [5, 4], Mentat [13], Charm++, and C** [15]. In addition, there are C++ libraries, such as P++ [17], which provide support for advanced data structure handling in parallel systems. Such libraries are an essential part of applications requiring adaptive structured grids.

In this paper we describe another C++ extension for parallel programming called pC++ [16, 11, 2, 10, 12] and its use in Grand Challenge Cosmology Consortium (GC^3) simulations. Section 2 of this paper contains a brief overview of pC++. In section 3 and 4 we describe the implementation of two GC^3 N-body codes in pC++. Finally, we discuss ideas of how pC++ can be used in adaptive mesh refinement applications.

2 OVERVIEW OF pC++

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

*This research is supported in part by ARPA under contract AF 30602-92-C-0135, the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616, and the National Science Foundation and ARPA Grand Challenge Award, "The Formation of Galaxies and Large-scale Structure."

[1]For example, see the proceedings of the 1993 and 1994 Object-Oriented Numerics Conference.

- parallel object-oriented software that runs without modification on all commercial scalable parallel Multiple Instruction Multiple Data (MIMD) systems,
- an interface to Single Program Multiple Data (SPMD) libraries such as ScaLapack++ [8] and P++,
- an interface to High Performance Fortran (HPF),
- and an interface to "multithreaded" C++ languages like CC++.

We emphasize that pC++, or other object-oriented parallel programming languages, should not be viewed as a replacement for Fortran90 or HPF. Rather, object-oriented parallelism should be a tool to express the types of parallelism that cannot be easily expressed in Fortran. To accomplish this task pC++ exploits the two major characteristics of object-oriented design: encapsulation and inheritance.

Encapsulation reflects the principle of object-oriented language: closely related variables and functions are grouped together into a user-defined data type called *class*. An instance of the data type is called an object and the associated functions are called member functions or methods of the class. Invoking a member function of an object is often viewed as sending the object a message to carry out a specific task. In this way an object-oriented program can be visualized as a network of objects each of which manages its own data structure and operates collectively by sending messages to other objects. This metaphor of communication lends itself perfectly to parallel execution, and most parallel programming extensions to C++ rely on this model.

Inheritance allows a class to inherit selected data members and member functions of other classes, rather than reimplementing them. The inheritance mechanism used to build distributed data structures in pC++ is the same as in C++, *i.e.*, through class derivation.

pC++ has two basic extensions to the C++ language: a mechanism to describe how operations can be invoked over a set of objects in parallel, and a mechanism to refer to an individual object.

A data structure called *collection* is used to describe a set of objects distributed over the processors of a parallel machine. To build a collection of some class type `T`, which is called an *element* class in pC++, one needs a distribution and an alignment object. The distribution object defines a grid and a mapping from the grid to the physical processors on a parallel machine. The alignment object specifies the shape, size, and the mapping of the element objects to the grid points. In addition, a processor object, `Processors`, is needed to represent the set of processors available to use.

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

creates a one dimensional grid of a size of 100 which is mapped to the processors of the machine by blocks. If there are 20 processors, grid positions 0 through 4 are mapped to processor 0, positions 5 through 9 are mapped to processor 1, ..., etc. The alignment aligns the logical vector `X[0:19]` with the grid positions `D[10:29]`. Given a distribution, an alignment and the class type of the element objects, it is easy to build a collection. One of the base collections that pC++ collection library provides is the `SuperKernel` collection. It is designed to be used as the base collection for all other collections. It builds arrays of element objects and provides global name space for the element objects. The declaration

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

creates a collection called `MyCollection` which is a set of objects of type `T`.

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

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

where function `foo()` is a member function of the class, a parallel application of `foo()` to all the elements would take the following form:

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

To access an individual member of a collection, one can use the overloaded operator `()` which returns a *global* pointer to an element. For example,

```
MyCollection(i)
```

returns a global pointer to the ith element in the collection. A global pointer is a pointer that spans the entire address space of a distributed memory machine. In this way, any object can have a global address. The following function call

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

is a remote invocation, it sends a message to the processor that contains the ith element and a thread on that processor executes the function. The remote invocation feature is an essential part of pC++ 2.0 and is not part of the current pC++ 1.0 distribution.

Usually, a user would want to create a specialized collection that has properties appropriate for the application at hand. The task of building a new derived collection is almost the same as building a derived class in C++. A collection definition takes the form shown below.

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

There are two types of data and member functions in a collection definition. Data members and functions labeled as *MethodOfElement* represent new functions and data that are added to each element class. The member functions are invoked just as if they are member functions of the element class. Other data member not labeled as MethodOfElement are duplicated on each processor and the functions are invoked in the SPMD mode.

In the next two sections we will illustrate the use of collection to define parallel data structures for the N-body simulation codes which are a part of the GC^3 project. We will also show how our prototype Fortran interface is used in one of the codes, the Self-Consistent Field code.

3 THE SELF-CONSISTENT FIELD CODE

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

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

density ρ and potential Φ are expanded in a set of basis functions. The basis set is constructed so that the lowest order members well-approximate a galaxy obeying the de Vaucouleurs $R^{1/4}$ projected density profile law. The algorithm used in the SCF code is described in detail in Hernquist and Ostriker 1992 [14].

The original SCF code was written in Fortran77 by Lars Hernquist in 1991. In 1993, the code was converted to Thinking Machines CM Fortran by Greg Bryan. Expirements conducted on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) by Greg Bryan indicate that with 10 million particles the CMF code can achieve 14.4 Gflops on 512 nodes of the CM-5 (private communication).

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

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

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

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

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

The basis sets ρ_{nlm} and Φ_{nlm} are given by

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

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

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

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

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

3.1 The pC++/Fortran Version of the SCF Code

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

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

The new data type `FArrayDouble` is a predefined class to contain Fortran double precision arrays. On the CM-5, `FArrayDouble` represents parallel arrays — arrays allocated in the vector processor accelerators' memories. In the above declaration, `x`, `y`, `z`, `vx`, `vy`, `vz`, `ax`, `ay`, `az`, `mass` are one-dimensional arrays that contain the positions, the velocities, the accelerations, and the masses of particles belonging to a `Segment` object, and `plm`, `clm`, `dlm`, `elm`, `flm`, `dplm` contain the expansion coefficients and the values of the polynomial. The `sinsum` and `cossum` contain the local sums and eventually the global sums of expansion coefficients. They are serial arrays on the CM-5 and are allocated in the Sparc chip's memory. Among the class member functions (Fortran subroutines), `compute_polynomial` computes the polynomials and local sums, `compute_acceleration` computes the acceleration for each particle, `update_position` and `update_velocity` update the positions and velocities of particles.

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

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

Functions declared here are pC++ functions. Their main functionality is to handle the I/O. `InParticles`, `InParameters`, and `OutParticles` read the input files and write to the output files, and `read_segment` and `write_segment` are called by `InParticles` and `OutParticles` to perform parallel I/O on both the CM-5 and Paragon. Functions that are already defined in element class `Segment` are declared as virtual functions in this collection declaration. The inherited `Fortran` collection is a parent collection which handles inter-element communication. `Fortran` itself is derived from the `SuperKernel` collection.

The main program is listed below:

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

CPU time in seconds spent in main loop without I/O.			
No. of Particles	pC++/CMF on CM5	CMF on CM5	pC++/F77 on Paragon
512	5.4 (76.6)	6.6 (40.9)	5.8
5120	8.6 (75.1)	9.9 (40.9)	25.0
51200	45.8 (89.1)	50.3 (57.4)	332.5
Total CPU time in seconds with parallel I/O			
51200	49.1 (89.1)	58.2 (57.4)	396.3

Table 1: *Execution time in seconds for 100 time steps on 32-node partitions with the expansions truncated at nmax = 6 and lmax = 4. Numbers in parentheses are amount of memory used in MBytes. The total time includes reading the input model (2.87 MBytes), initializing the system, evolving the system for 100 evolution steps, and writing two output files (2.87 MBytes and 2.46 MBytes).*

```
   X.compute_polynomial();
   X.ReduceDoubleAdd(offset,variable_count);
   X.compute_acceleration();
   X.update_velocity();
   X.OutParticles(n);
  }
}
```

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

3.2 Timing Results

Our experiments with the pC++/CMF SCF code was conducted on the 32-node CM-5 at the University of Maryland and the 88-node Paragon at Indiana University. For comparison, we also ran the CMF SCF code on the CM-5. The results of these experiments are listed in Table 1.

As it can be seen in Table 1, the pC++/CMF code is, in fact, slightly faster than the CMF SCF code in all cases. This can be explained by that the pC++/CMF code is able to use a faster global vector reduction function. The timing results show that our Fortran interface implementation is very efficient. Given the results shown in Table 1 and experiments conducted on NCSA's CM-5 using larger partition sizes, we conclude that the pC++/CMF code can achieve at least the 14.4 Gflop speed of the CMF code.

4 THE PARTICLE MESH CODE

Another N-body code in the dossier of the GC^3 group is the Particle Mesh (PM) code (see Ferrel and Bertschinger 1993 [9]). The particle-mesh method used in the PM code computes long-range gravitational forces in a galaxy or galaxy cluster system by solving the gravitational potential on a mesh. The three-dimensional space is discretized by a three-dimensional grid. An average density for each grid point is then computed using the Nearest Grid Point scheme in which the density value at a grid point is the sum of all masses of the particles nearest to that grid point. Once the density values at the grid points are known, Fourier transforms are performed to compute the potential values at the grid points. And finally, the potential values at the grid points are interpolated back to the particles and the particle positions and velocities are updated. Two data structures that are natural to the particle-mesh method are: a one-dimensional particle list and a three dimensional mesh.

4.1 The Particle List Collection

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

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

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

The `Segment` class is defined as

```
class Segment {
 public:
  int particle_count; // Number of particles in the segment
  double x[particle_count],y[particle_count],z[particle_count],
         vx[particle_count],vy[particle_count],vz[particle_count],
         mass[particle_count];
  ...
};
```

where `x`, `y`, `z`, `vx`, `vy`, `vz`, and `mass` represent the position, the velocity, and the mass of a particle, respectively.

The `ParticleList` collection is defined as

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

The function `sortParticles()` is a routine that sorts the particles in lexicographic order according to their positions. The function `pushParticle()` uses the gravitational force to update the positions and velocities of the particles. The argument passed to `pushParticle()` is a collection designed for the mesh data structure. The function `updateGridMass()` is used to add the mass of a particle to the total mass of the mesh point to which it is closest. The importance of this function is that it first loops through the particles local to a segment and accumulates a local total mass for each mesh point. It then adds the local total mass to the mesh point's total mass by a remote update operation on the appropriate mesh point. Because remote updates are expensive, this operation should be infrequent. This is why the particles are sorted to form a list in the beginning.

During the sorting, the particles within each segment are sorted first using the quick sort library sorting function. Then a global sorting is performed using the bitonic sort algorithm [6].

4.2 The Mesh Collection

The mesh is logically a three dimensional array of mesh points each containing the value of density and position. Because FFT is used to solve the gravitational potential equation, the data structure is designed as a two-dimensional collection with each element contains an array of mesh points that have the same x and y coordinates but different z coordinates:

```
class MeshElement {
 public:
  double mass[nz], position[nz];
};
```

The collection `Mesh` is defined as follows:

```
Collection Mesh : SuperKernel {
 public:
  Mesh(double lx, double ly, double lz, Distribution *T, Align *A);
  void computePotential();
  ...
};
```

where function computePotential() computes the gravitational potential using the total mass at each mesh point and lx, ly, lz are the numbers of mesh points in the three x, y, and z dimensions, respectively.

To calculate the gravitational potential (or forces), discrete Fourier transform is used twice: first the density is transformed into the wavenumber domain and then the potential (or force component) is transformed back into spatial domain.

FFT transform in the x-, y-, and z-directions is performed by the Mesh collection. The FFT in the z-direction poses no problems, as for each MeshElement with indices (i,j) the whole array of mesh points with indices (i,j,*) is located in that single MeshElement. Each MeshElement performs the z-FFT on its data without inter-element data communication, and load balancing is also ensured.

To perform FFT in the x-direction, the values of density at mesh points with indices (*,j,k) are collected by the collection element, Mesh(i,j). It then performs sequential FFT and sends back the resulting potential values to the mesh points from which it collected the density values.

To ensure good load balancing, we want all collection elements to participate the x-FFT. If it is known that the number of mesh points in the x-direction is equal to the number of mesh points in the z-direction, we can let all Mesh(*,j) send their kth density value to Mesh(k,j). If the number of mesh points in the x-direction is not equal to the number of mesh points in the z-direction, we can let Mesh(*,j) send their kth density value to Mesh(l,j) where l = (j + k) % lx. The y-direction FFT can be performed in a similar fashion.

4.3 The Main Simulation Loop

In its simplest form the simulation is given by the code below:

```
main(){
 Processors P;
 Distribution DP(segment_count, &P, BLOCK);
 Align AP(segment_count, "[ALIGN(particlelist[i], DP[i])]");
 ParticleList&lt;Segment&gt; particlelist(&DP, AP);
 Distribution DM(x_count, y_count, &P, BLOCK, WHOLE);
 Align AM(x_count, y_count, "[ALIGN(mesh[i], DM[i])]");
 Mesh<MeshElement> mesh(&DP, AP);

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

The main loop involves parallel computation on both `Mesh` and `ParticleList`. First the potential is computed in parallel on the grid. Second, the particle velocities and positions are updated. If particles have moved to new grid points, the appropriate data structure updates must then be made. If the particles have moved a lot since the last time they were sorted, the particles are sorted again. Third, the particle masses are accumulated in their corresponding points for the next iteration step.

5 ADAPTIVE GRID AND MULTI-THREADED EXECUTION

pC++ is designed to support parallel programming styles that are not well supported by Fortran90 and HPF. Problems with complex, adaptive data structures are naturally suited for an object-oriented language like C++. Though it is true that Fortran90 supports many advanced data-structure features, they are not as flexible as those in C++ and have not been integrated into HPF.

In this section, we describe the basic design of pC++ collection that will be used in adaptive grid computations. The idea is to allow a mesh to be refined in regions where more resolution is needed, for example, near a congestion of galaxies or stars. We modify the `Mesh` collection element class so that a grid cell can represent a new finer grid if the cell is in a region where extra resolution is required. The modified `MeshElement` now called `MeshCell` is give below:

```
class MeshCell{
 double gravPotential;
 double massDensity;
 int isRefined;  // 1 if the cell is refined
 Mesh<MeshCell> *refinedRegion;
};
```

The resulting data structure, as illustrated for a two dimensional problem in Figure 1 below, is a dynamical structure. Multi-threaded execution plays an important role in this type of parallel programming. Because new collections are created within elements of a collection, we must be able to invoke parallel operations within parallel sections. This type of nested parallelism requires a multi-threaded model of execution.

While it is still too early to evaluate this approach to parallel adaptive structures, we feel that the potential for success is high. The reader should note that there are several other approaches to adaptive parallel grid computation [17, 1, 7].

6 CONCLUSION

In this paper we have show how pC++ is used to design and implement object-oriented software for N-body and adaptive mesh refinement simulations. As a part of the NSF GC^3 effort, pC++ research will focus on problems related to adaptive computations on parallel machines. pC++ 1.0 is now available by anonymous ftp from Indiana University (ftp.cica.indiana.edu in pub/sage). Currently, pC++ runs on the Thinking

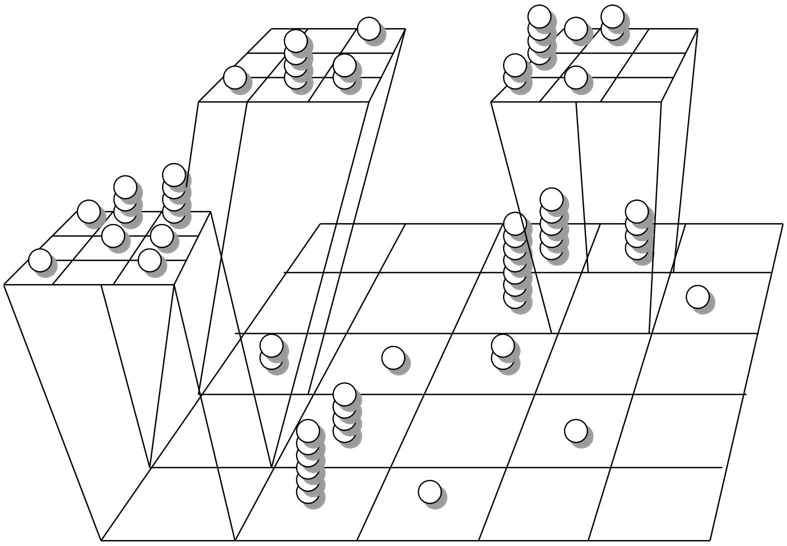

Figure 1: *Adaptive Mesh Refinement. If more resolution is required, the grid cell is refined to form a new mesh structure.*

Machines CM-5, the Intel Paragon, the BBN TC2000, the Kendall Square Research KSR-1, the IBM SP-1, Parallel Virtual Machines (PVM) for workstation clusters, and the Sequent Symmetry. A version for Cray T3D will also soon be available.

pC++2.0 is currently under construction. The key new features in pC++2.0 are multi-threads and global pointers. Multi-threaded execution is important for parallel adaptive structures. Global pointers are an essential requirement for the N-body simulations describe in this paper.

References

[1] S. Bhatt, M. Chen, C-Y Lin, and P. Liu. Object-oriented support for adaptive methods on parallel machines. In *Object Oriented Numerics (OON-SKI)*, pages 266–267, 1993.

[2] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing 93*. IEEE Computer Society, 1993.

[3] R. Chandra, A. Gupta, and J. Hennessy. *COOL: a Language for Parallel Programming*, pages 126–148. The MIT Press, 1990.

[4] K. M. Chandy and C. F. Kesselman. Compositional C++: Compositional parallel programming. In *Fourth Workshop on Parallel Computing and Compilers*. Springer-Verlag, 1992.

[5] K. M. Chandy and C. F. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. MIT Press, 1993.

[6] T. H. Cormen, C. E. Leiserson, and R.L.Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[7] W. Crutchfield and M. Welcome. Object oriented implementation of adaptive mesh refinement algorithms. In *Object Oriented Numerics (OON-SKI)*, pages 108–122, 1993.

[8] J. Dongarra, R. Pozzo, and D. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Object Oriented Numerics (OON-SKI)*, pages 257–264, 1993.

[9] R. Ferrell and E. Bertschinger. Particle-mesh methods on the connection machine. *International Journal of Modern Physics C*, 1993.

[10] D. Gannon, F. Bodin, P. Beckman, S. Yang, and S. Narayana. Distributed pC++: Basic ideas for an object parallel language. In *Object Oriented Numerics (OON-SKI)*, pages 1–24, 1993.

[11] D. Gannon and J. K. Lee. On using object oriented parallel programming to build distributed algebraic abstractions. In *Conpar-Vap*, pages 769–774. Springer Verlag, 1992.

[12] D. Gannon, S. X. Yang, and P. Beckman. *User Guide for a Portable Parallel C++ Programming System, pC++*. Computer Science Department, Indiana University, available from moose.cs.indiana.edu:/pub/sage by ftp, 1994.

[13] A. S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report 91 07, University of Virginia, 1991.

[14] L. Hernquist and J. P. Ostriker. A self-consistent field method for galactic dynamics. *The Astrophysical Journal*, 386:375–397, 1992.

[15] J. Larus, B. Richards, and G. Viswanathan. C**: A large-grain object oriented data-parallel programming language. Technical Report 1126, Computer Science Department, University of Wisconsin, 1992.

[16] J. K. Lee and D. Gannon. Object oriented parallel programming: Experiments and results. In *Supercomputing 91 (Albuquerque, Nov.)*, pages 273–282. IEEE Computer Society and ACM, 1991.

[17] M. Lemke and D. Quinlan. P++, a parallel C++ array class library for architecture-independent development of numerical software. In *Object Oriented Numerics (OON-SKI)*, pages 268–269, 1993.