Journal of Computational Physics **151,** 264–282 (1999)
Article ID jcph.1999.6199, available online at http://www.idealibrary.com on IDEAL®

OOMPAA—Object-Oriented Model for Probing Assemblages of Atoms

Gary A. Huber and J. Andrew McCammon

Department of Chemistry and Biochemistry, University of California, San Diego, La Jolla, California 92093-0365
E-mail: ghuber@chemcca10.ucsd.edu, Fax: 619-534-7042

Received July 7, 1998; revised December 24, 1998

An object-oriented library is presented for building molecular-modeling software. This library allows the user to treat individual components of molecules as C++ objects, and provides various templated lists and vector classes for manipulating these objects. Other utilities, such as minimizers and integrators, are continually being added to the body of code. Performance is a key consideration; the performance of simple benchmarks is comparable to that of hand-coded Fortran.

Key Words: object-oriented; molecular mechanics.

INTRODUCTION

Molecular simulation software is not simple, and most of it is written with computational efficiency in mind, rather than elegance of expression. This means that the source code is written in Fortran or C, with very strong coupling among the different components of the program. This is fine if the software will not change or be combined with other software; a computer program can then be treated as a monolithic black box. However, computer programs grow and change, and programs that are nicely written in Fortran tend to degrade over the years as generations of scientists and programmers add their own ideas. In addition, adding new functionality to such programs is a very laborious and tedious undertaking; thus, many promising ideas in the literature never make it into widely used code. Finally, two different software packages from two different research groups or companies might have different functions that one would want to combine; this is almost an impossible undertaking for many cases. Even when workers succeed in altering or combining such code, the results cannot always be trusted because of the introduction of bugs. This problem is not unique to the world of molecular modeling; it is known as the *software crisis* [4].

A promising way out of the software crisis in molecular modeling lies in *object-oriented programming*. Traditional procedural languages like Fortran are built around the subroutine, and the data are conceptually separate from the subroutine. This works well from an efficiency viewpoint, because that is how the computer sees the world, but it is not how

0021-9991/99 $30.00

humans see the world. Object-oriented languages, on the other hand, group related pieces of data together with the functions that act upon them; these are the objects. This results in more understandable code and a decoupling of unrelated code that allows the programmer to change one part of the program without affecting any other parts. The programmer can also create new types of objects from existing objects without changing the original object. In the past few years, interest has grown in devising heterogenous models for large molecules; such hybrids include combinations of quantum-mechanical [5, 10, 1, 2] methods and continuum descriptions of matter [12, 22, 9, 16] with molecular mechanics. Also, in order to improve configuration-space sampling and to make use of parallel computers, it is often desirable to use multiple copies of such models [17, 18, 8, 7]. Finally, for very large systems, it might be necessary to distribute just one copy over many processors [6]. Simulations combining several of the above features may require code of extreme complexity if written in C or Fortran.

Until recently, the main issue has been performance. C++ code did not compare in speed to equivalent Fortran code. However, this has changed in the past few years with C++ compilers that are now available. Numerically intensive C++ code can run as fast as Fortran code, while keeping the elegance of expression afforded by objects. In some cases, this results in more efficient algorithms, because related pieces of data are more likely to lie near each other in memory [27]. There are other elegantly formulated object-oriented languages such as Java and Eiffel, but none of them rival C++ in performance. Thus, it appears that C++ holds the future for scientific programming. Indeed, there already exists at least one widely used molecular simulation package that is written in C++, NAMD [3].

This situation is the inspiration for OOMPAA (Object-Oriented Model for Probing Assemblages of Atoms), which is a collection of C++ classes for constructing molecular-modeling software. OOMPAA has two major divisions: the *core* and the *additions*. The core includes the most basic objects that are used to describe molecules and is expected to remain very stable. The additions to OOMPAA include include C++ classes that act as machines to manipulate the data structures in the core; this section is constantly growing. Most of this article describes the core. OOMPAA strives to be very general. While it is sure to be useful in studying biomolecules, there is no obstacle to using it on other types of systems, and nothing in the core is unique to biomolecules. OOMPAA tries to insulate the user from the complexities of C++ (of which there are many). The user should only need to know what classes, objects, and templates are, and to understand the concept of pointers. Included are several scripts that automatically generate C++ code for classes with desired properties. OOMPAA is written with the assumption that the user has available a state-of-the-art optimizing C++ compiler that is nearly compliant with the draft ANSI/ISO standard. Assuming good compilers, OOMPAA is meant to be portable; it makes use of other portable class libraries, and all scripts are written in the language Python [25]. OOMPAA is free, users are encouraged to add classes to it or make their own code freely available, and OOMPAA makes use of other free C++ class libraries, such as VTK [21] for visualization.

OBJECTS

In computer science, an *object* is a collection of data that has functions defined on it. Often, an object in the computer will correspond to an object in the real world, with the data representing relevant information about the object's state and the functions representing possible actions taken by the object. In C++, the type of an object is known as its *class*.

All objects belonging to the same class have the same functions and the same types of data, but each object has its own copy of the data. First, the programmer creates a class, which describes the data and functions, and later in the computer program, objects of that particular class are created, manipulated, and destroyed. The closest thing to classes in the C language is *structures*, which are collections of data; Fortran 90 has *modules*, while Fortran 77 has nothing that resembles classes.

In most object-oriented languages, one can define *pointers* or *references* to objects. A pointer or a reference is merely a number that contains the starting location of an object's data in memory. C++ has both pointers *and* references; they are essentially the same thing but with slightly different syntax. Throughout this paper, the term *pointer* will refer generically to both C++ pointers and references unless the distinction must be made clear. Pointers are useful because one object can *refer* to another object or be associated with it, without needing to copy any data. When a pointer is created in C++, it must always point to objects of the same class, with one exception, given below.

Objects are very versatile. Objects can contain other objects or pointers to other objects, and functions can have objects or pointers to objects as arguments and return values. New classes can be created from existing classes by adding new functions and data; the old class is not changed, but the new class can be used wherever the old class is used. This is known as *inheritance*. Typically, the old class is known as the *parent*, or the *base class*, and the new class is known as the *child*. A pointer that points to an object of a particular class can also point to an object belonging to one of its child classes.

Objects also facilitate *data-hiding*, in which the internal workings of the object are hidden away from the outside world. It is possible to restrict access of data members and functions; if a data member is declared to be *private* (as opposed to *public*), only member functions of that particular class can gain direct access to those data. Indeed, it is considered poor programming practice for any data to be public; only functions should be public, as will be seen below.

In OOMPAA, perhaps the most basic class is the `Particle`. The OOMPAA Particle has very few features; the only data member is an unsigned long integer, the particle's *id* that can be used to represent the chemical type of the particle. For example, one value of the id might be used to represent an aliphatic carbon atom, while another value might represent an alcohol oxygen atom. The core of OOMPAA places no restrictions and makes no assumptions regarding the use of the id.

By itself, the bare `Particle` class is not very useful, but the user can create new particle-like classes with useful features by inheriting from the basic `Particle` class. Suppose that the user wants to perform a Monte Carlo simulation, in which particle positions are varied in a stochastic manner. Clearly, the particle needs data to describe its position. So, the user creates a class, called `MC_Particle`, which contains another object representing a 3-dimensional vector. The data are accessed indirectly through *accessor functions*. The following C++ code gives an outline of this new class (function implementations are not shown):

```
class MC_Particle: public Particle{
public:
  void Set_position( const Vector3< double> x);
  Vector3< double> position() const;
  void Add_to_position( const Vector3< double> dx);
private:
  Vector3< double> _position;
};
```

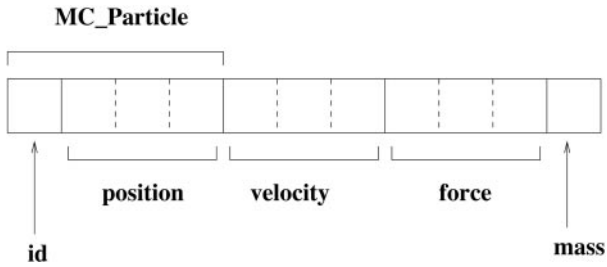

FIG. 1. Possible memory layout of a particle used in MD simulations.

The class describing the vector of 3 double-precision floating-point numbers is `Vector3< double>`; it has *template* syntax, which will be described below. The first function sets the position to x, the second function returns the particle's position, and the third function increments the position by dx. Fortunately, the user never needs to write such code; OOMPAA provides a script, `Create_particle`, to automatically generate C++ code for child classes having desired features. The user only needs to tell `Create_particle` that the new particle class must have a position described by a vector of the desired type, and all of the new C++ code is generated. (This is in accordance with the philosophy of shielding the user from the complexities of the C++ language.) Of course, the user can also set and retrieve the `MC_Particle` object's id which is inherited from the base `Particle` class.

Next, suppose that the user needs to perform a molecular dynamics simulation on the same system of particles. Now, a new particle class, `MD_Particle`, is needed, which has mass, velocity, and force, in addition to id and position. Using the `Create_particle` program, the user can specify that the velocity and force on the particle be treated like the position above, with 3-dimensional vectors storing the data. However, there are several different ways in which one might handle the mass. If all particles have the same mass, then the function `mass()` that returns the object's mass can simply return the same value for all objects. If memory is not in short supply, each object can carry around its own floating-point number representing its mass. Finally, the mass of the object can be computed by using the object's id number to look up the mass in a table. No matter how the retrieval of the mass is implemented, it is desirable that the interface between the `MD_Particle` object and the outside world, namely, the function `mass()`, not change. It is possible that the user might want to change the way in which the mass is implemented; if the interface stays the same, then all code that relies on it will still work. This would not be possible if the data representing the mass could be manipulated directly. The `Create_particle` script can generate code for all three possible implementations. For the case where the mass is stored explicitly, the layout of a `MD_Particle` object is illustrated in Fig. 1. The reader should note that neither of these classes derived from `Particle` is included in the core of OOMPAA; the authors do not presume to know which type of `MD_Particle` is most useful for the user.

TEMPLATES

A very useful code-reuse device in C++ is the technique of *templates*. Although templates have a very broad domain of usefulness, in the core of OOMPAA they are used mainly for the creation of objects that contain either other objects or pointers to other objects. Perhaps

the most simple example is that of the class `Vector3< double>` above, which contains three double-precision floating-point numbers. It is also possible to have a 3-dimensional vector of integers by creating objects of the class `Vector< int>`. Even though these are two different classes, they have the same code, which is written once in a very generic manner to accommodate all reasonable contained data types. In addition to templated classes, it is possible to have templated functions which can have different types of arguments and return values.

LISTS

In OOMPAA, one does not usually deal with individual particles; rather, one deals with *lists* of particles. Thus, OOMPAA provides a template class `List<>`, which represents variable-length lists of objects. So, if the user wants to create a list with 1000 `MC_Particle` objects described above, the code would be:

```
List< MC_Particle> particles( 1000);
```

When this code is executed, a list of 1000 new particles is created; it is then the user's role to put meaningful data into the particles themselves. One can access the member particles as if the list were a C++ array:

```
Vector3< double> x;
x = particles[100].position();
```

Here, the position of the 100th particle is placed into the vector `x`. (The "dot" notation used in the second line is used to denote the member function `position()` of a particular object `position[100]`.) Unlike simple C++ arrays, the OOMPAA `List` has member functions that can copy other lists, add and delete members, apply a given function to the members, create sub-lists, and carry out set operations such as unions and intersections. Templated lists are not a new idea; the C++ Standard Template Library [19] uses the same idea (in fact, the Oompaa list makes use of the STL `vector`). An important limitation is the fact that this list can contain only `MC_Particle` objects; this restriction is eased in the next section.

POINTER LISTS

The templated class `List` is actually two templated classes in one; it is possible to create lists that contain *pointers* to objects rather than the objects themselves. The type of list in the illustration above is called a *body list*, as opposed to a *pointer list*, because it contains the bodies of the objects. Body lists are obviously very important because they contain all of the information, but pointer lists are very flexible and useful for manipulating groups of atoms and for making new lists.

As an illustration, consider the following:

```
List< MC_Particle> mc_particles( 5);
List< MD_Particle> md_particles( 5);
List< MC_Particle *> all_particles = mc_particles + md_particles;
```

Two body lists, one of five `MC_Particle` objects and one of five `MD_Particle` objects, are created, and then a pointer list is created that represents the union of the two body lists. The pointer list is distinguished by the $*$ inside the template brackets. When the pointer

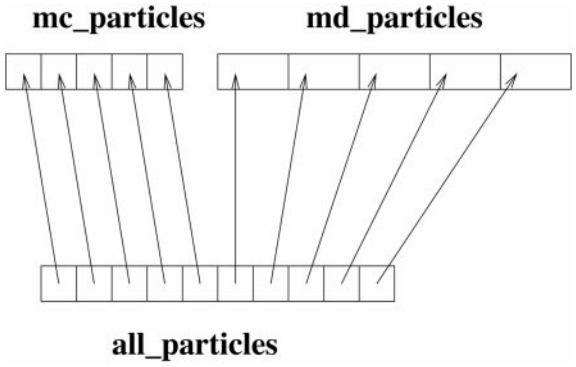

FIG. 2. Pointer list example.

list is created, no changes take place inside either body list, but the pointer list contains 10 pointers which point to each member of the two body lists (Fig. 2). Also, because the class `MD_Particle` is a child class of class `MC_Particle`, it is possible for a list of pointers to the parent class to contain pointers to the child class. This flexibility can be very useful; for example, if the user wants to carry out Monte Carlo steps on the `MD_Particle` objects as well as the `MC_Particle` objects, the pointer list can be passed to the function or object that performs such a computation. Moreover, the user can create an entirely new body list from this pointer list:

```
List< MC_Particle> all_particle_bodies = all_particles.body_list();
```

Now, the list `all_particle_bodies` contains 10 new `MC_Particle` objects that are copies of the original objects. The five objects that correspond to the original `MD_Particle` objects are *sliced* from the original, with the mass, velocity, and force data excluded.

Pointer lists can be useful for selecting out groups of particles for special treatment. As an illustration, suppose that the atoms in the active site of an enzyme merit some special treatment apart from the rest of the atoms. Given a function that returns `true` or `false` if an atom is or is not considered to be in the active site, this is easily done:

```
bool in_active_site( const MC_Particle &); // returns true if in
  active site
.
.
List< MD_Particle> all_atoms;
// now initialize list;
.
.
List< MD_Particle *> active_site_atoms = all_atoms.sub_list( in_
  active_site);
// select atoms that are not in active site
List< MD_Particle *> other_atoms = all_atoms - active_site_atoms;
```

It should be noted that the function `in_active_site` takes a reference to the *parent* class as an argument, assuming that only the position is necessary to determine membership in the active site. Still, a list of the objects belonging to the child class can make use of this

function, opening up the possibility of reusing Monte Carlo code in molecular dynamics code. Indeed, pointer lists contain a larger set of functions than do the body lists, because of their flexibility. Although rearranging lists is done infrequently enough in typical simulations that it is unlikely to become a bottleneck, care has been taken to ensure that no list operation's execution time has a scaling worse than $N \log N$, where N is the number of items.

The = operator, when applied to `List` classes in OOMPAA, follows the convention of *copy-by-reference*. This is the default behavior of objects in several other object-oriented languages such as Java and Python, but not in C++. For example, in the code

```
List< MD_Particle> list1( 100), list2( 200);
list2 = list1;
```

what happens is that `list2` and `list1` now refer to the same underlying object containing 100 particles, while the 200 particles in the original `list2` are deleted. Each underlying list object keeps track of the number of names referring to it, and when the last name is deleted or assigned to something else, the underlying object deletes itself. This differs completely from the container objects in the Standard Template Library, in which the objects on the right side of the = operator are copied to the container on the left. The copy-by-reference convention increases the convenience of complicated list manipulations.

OOMPAA includes convenient C-style macros for looping through members of lists and pairs of members. For example:

```
List< MD_Particle> list( 10000);
...
FOR_ITEMS_IN_LIST( MD_Particle, list, atom)
  Do_something( atom);
END_ITEMS_IN_LIST;
```

In the above code, the function `Do_something` is applied to each member of `list`. Here is a more complex example that loops through all pairs of particles in a list to compute the Coulombic energy of interaction:

```
List< MD_Charged_Particle> list( 10000);
// initialize atoms
...
double energy = 0.0;
FOR_PAIRS_IN_LIST_OUTER( MD_Charged_Particle, list, atom1)
  double q1 = atom1.charge();
  FOR_PAIRS_IN_LIST_INNER( MD_Charged_Particle, list, atom2)
    double q2 = atom2.charge();
    Vector3< double> r = atom1.position() - atom2.position();
    double R = norm( r);
    energy += q1*q2/R;
END_PAIRS_IN_LIST;
```

Good C++ compilers can optimize the above code to get performance that is comparable to equivalent Fortran code. The macros work with both pointer lists and body lists. In some cases, perhaps depending on personal programming style, it might be desirable to use *iterators* [19] instead of macros to loop through a sequence of objects. Iterators are more flexible, but macros might be more readable to those coming from a Fortran-based

background. Future versions of OOMPAA will include iterators corresponding to the looping macros.

CHEMICAL STRUCTURES

Another kind of templated container, the *chemical structure*, contains a fixed, small number of pointers to other objects. It also contains an id integer just like the one in `Particle` objects. In OOMPAA, the name of the class is `Structure`. It has *two* template parameters: the first denotes the type of object pointed to, and the second is an *integer* that denotes the number of pointers. Integers can be template parameters; this fixes the number of pointers during compilation, allowing more efficient code. One very important application of chemical structures is the representation of bonds between atoms; the id number can denote the type of bond, such as double carbon–carbon or single carbon–nitrogen. Likewise, chemical structures with three pointers can represent bond angles, and quartets of pointers can represent torsion angles. One disadvantage of templates is that the names can become unwieldy; in such cases, it can be useful to use `typedef` to condense the names:

```
typedef Structure< MD_Particle, 2> Bond;
List< Bond> bonds;
List< Bond *> active_site_bonds;
```

As seen above, `Structure` objects can be stored in body lists and referenced by pointer lists. When either type of list contains a chemical structure, additional list functions are available to select `Structure` objects with certain properties or to create lists of `Particle` objects referenced by the `Structure` objects.

GROUPS

A third kind of templated container is the *group*. A group is like a chemical structure in that it refers to other objects rather than containing them, it has a fixed length, and it has an id number. Unlike a chemical structure, it does not contain an individual pointer for each referenced object; instead, it contains one pointer that points to an object or a pointer in a `List` object and an integer than denotes the number of objects referenced in the list beyond the first one referenced (Fig. 3). Like the class `List`, groups come in two flavors. A *body group* points directly to objects in a body list, and a *pointer group* points to other pointers in a pointer list, thus indirectly referencing the objects. Each object referenced by a group is given a unique name and is accessed by appropriate functions. Because groups are more complex than chemical structures, the script `Create_group` is used to generate the C++ code for a group class; it uses an input file that gives the names for the referenced objects. Group classes do not have an integer template parameter, it is unnecessary because the C++ code is generated by the script. Body groups and pointer groups have similar behavior, but with pointer groups, it is possible to “delete” object references by setting the pointer in the corresponding pointer list to a null value. Finally, one can store and reference groups using the `List` class; as with chemical structures, additional functions become available for manipulating groups.

One use for groups is to represent amino acids. A base class, `Amino_Acid`, contains references to the atoms that all 20 of the usual amino acids have in common. The group

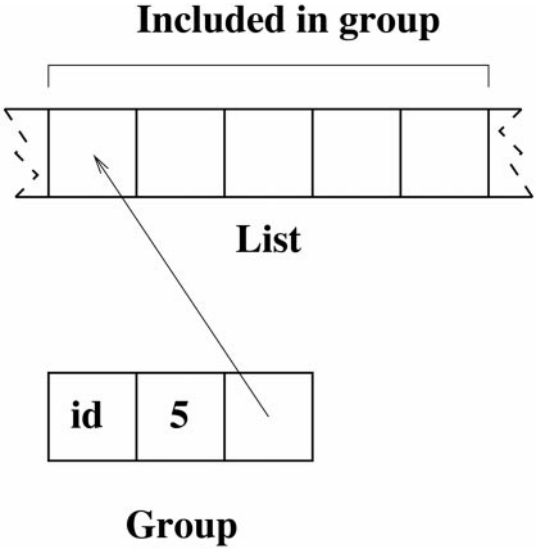

FIG. 3. Memory layout of a group and its data.

classes representing the actual amino acids inherit from `Amino_Acid`, adding references to their particular atoms. Similar groups can be generated to reference the bonds, bond angles, and torsion angles of the amino acids. The atoms of a protein molecule can be represented as shown in Fig. 4. All of the atoms are contained in one body list, and all of the amino acid group objects are contained in twenty different body lists. Each of the amino acid body lists contains groups of a specific type; for example, all alanine groups are contained in one body list. Finally, a pointer list of plain `Amino_Acid` groups points to all of the amino acid groups, in the same order in which they appear in the protein. Using this scheme, one could, with a few lines of code, create a list of atoms belonging to all amino acid residues whose

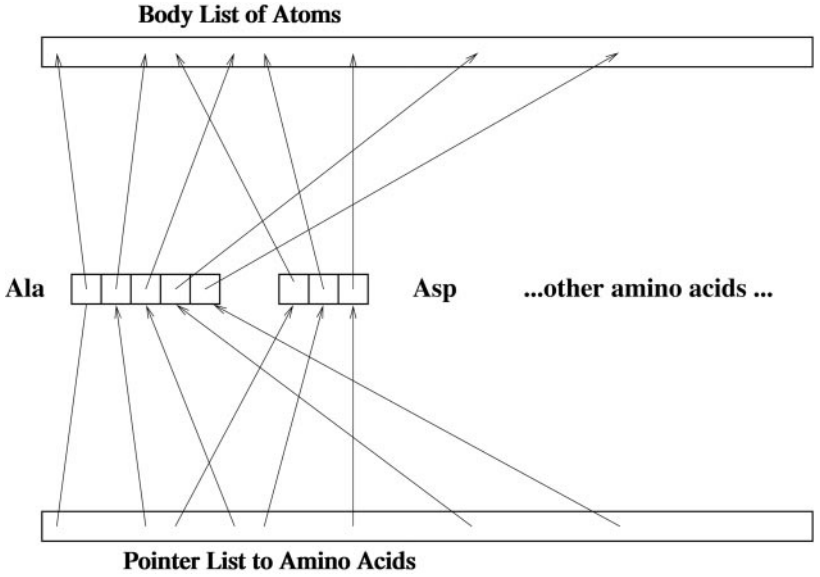

FIG. 4. Possible organization of protein molecule data.

α carbons lie inside the active site:

```
bool in_active_site( const MC_Particle &); // returns true if in
  active site
typedef List< Amino_Acid< MD_Particle> *> AA_List;
AA_List residues;
// set up residues
...
List< MD_Particle *> alpha_carbons =
  residues.selected_members( Amino_Acid::Ca);
List< MD_Particle *> alpha_carbons_in_site =
  alpha_carbons.sub_list( in_active_site);
AA_List residues_in_site =
  residues.groups_with_members( alpha_carbons_in_site);
List< MD_Particle *> atoms_near_site = residues_in_site.all_members();
```

VECTORS

In just about every scientific computing application, it is desirable to treat large collections of numbers as vectors, and to use these vectors in mathematical expressions within the code. One of the main impediments to the use of C++ in scientific computing has been the absence of standard vectors and matrices. When someone creates a class library for a particular area of computation, very often the author includes his own vector and matrix classes. It is true that the C++ Standard Template Library has a `vector` class, but this class does not have many facilities for numerical computing. Thus, this can lead to serious incompatibilities between different class libraries. Another obstacle is related to the operator overloading of C++. Consider the following example:

```
Vector< double> a( 10000), b( 10000), c( 10000);
// initialize vectors
....
c = a + b;
```

In most obvious schemes, the + operator is *overloaded*, or redefined, to return a `Vector` object, and the = operator is overloaded to accept a `Vector` object. Then, a temporary vector is created which holds the sum of `a` and `b`, and it then is copied to `c`. This greatly inhibits performance, especially on vector supercomputers. However, the recently developed technique of *expression templates* [26], together with good optimizing compilers [20], has completely changed this situation. Instead of returning a whole vector, the + operator returns a small, temporary object, called an *expression object*, that points to the vectors `a` and `b`. The = operator accepts this small object, hands it back an integer index, and the small expression object computes the sum of the elements of `a` and `b` denoted by the index. The true performance boost is realized when the compiler is sophisticated enough to eliminate the small object altogether and generate code that is equivalent to the following C code:

```
double a[10000], b[10000], c[10000];
int i;
....
for (i=0; i<10000; i++)
  a[i] = b[i] + c[i];
```

Furthermore, the expression template technique works on arbitrarily complicated expressions.

This technique, in addition to many others, is a part of the Blitz++ array class library of Veldhuizen [28]. The Blitz++ arrays comprise the primary vectors of OOMPAA. Blitz++ distinguishes large, variable-length arrays that are allocated on the heap from small, fixed-length arrays that are allocated on the stack. At the time of this writing, the small vector classes of Blitz++ are still under construction, so OOMPAA has included its own templated `Vector3` class, seen in several examples above. Although it is somewhat risky to tie a class library to one vector implementation, it is the authors' judgment that the Blitz++ arrays will eventually become the standard for C++ scientific computing. OOMPAA does not use the bare Blitz++ arrays, but it uses its own templated `Vector` class that inherits from the Blitz++ array and adds more features to allow interaction with *general vectors*, discussed below.

GENERAL VECTORS

Often it is convenient to represent certain components of the items in a list collectively as one vector. For example, one might want to treat the positions of the atoms in a list as one vector, the velocities as another, and the accelerations as a third. This is done by the `General_Vector` class, which takes two template parameters. The first parameter is a place-holder class that selects out the appropriate accessor function of the item, and the second parameter is the type of list. The `General_Vector` class acts as a "wrapper" around the list, causing it to appear as a vector to the rest of the code. Assume that the class `MD_Particle` has a function `acceleration()` that computes the acceleration from the mass and the force and returns a `Vector3< double>` object. Then, the implementation of a velocity-Verlet algorithm on a list of particles might look like this:

```
List< MD_Particle> atoms( 10000);
General_Vector< v::position, List< MD_Particle> > x( atoms);
General_Vector< v::velocity, List< MD_Particle> > v( atoms);
General_Vector< v::acceleration, List< MD_Particle> > a( atoms);
...
v += a*0.5*dt;
x += v*dt;
Compute_forces();
v += a*0.5*dt;
```

This code can be optimized by the compiler to be equivalent to hand-coded C or Fortran. The place-holder classes reside in the namespace `v` to avoid name clashes, and they can be automatically generated by the script `Create_general_vector`. The expression template method is implemented for general vectors, and they can interact fully with OOMPAA `Vector` objects. Another templated class, the `Composite_Vector`, allows one to stack two or more different `Vector` or `General_Vector` classes end-to-end to make a new vector class.

PARAMETERS

OOMPAA's treatment of parameters, such as those found in molecular mechanics computations, represents an attempt to satisfy three requirements. First, the parameters must be rapidly accessible to the computer. Second, it should be very easy for the user to change

parameters and introduce new parameters. Third, the presence of the parameters should be clear within the code itself; a hasty glance at the code should suffice to reveal the parameters.

In OOMPAA, parameters are stored in a templated `Parameter_List` object. This class is templated with respect to the C++ type of of the parameter; one can have not only parameters that are floating point numbers, but also vectors, tensors, integers, etc. The `Parameter_List` acts like a large array, where the parameter values are indexed by the type of parameter (e.g., charge, mass, bond-stretch spring constant) and the type of chemical structure (e.g., aliphatic carbon, alcohol oxygen, carbon–carbon double bond), as given by its id number. Many parameters depend only on one object (like charge). Others depend on two objects (like the Lennard-Jones interaction parameters); in this case, the parameter value would be indexed by the parameter type and both of the atom types.

When a `Parameter_List` object is created, it is initially empty. The `Parameter_List` is informed of a parameter type by giving it a string representing a name for the parameter. The `Parameter_List` then assigns a unique unsigned integer to the parameter; then, this integer can be obtained from the `Parameter_List` and used to access parameter values. A chemical object type is placed into the `Parameter_List` in the same manner. When a name (in the form of a string) is given to the `Parameter_List`, it registers a unique index. The `Parameter_List` class provides a function for reading parameters and their names directly from a file with a very simple format.

In order to enter the parameter value of interest, the `Parameter_List` object is given two integers and a parameter value. One integer represents the type of parameter, and the other represents the type of chemical object. The parameter value is entered into the `Parameter_List` object under those two indices. In a similar manner, the parameter value can be retrieved during a simulation by giving the the `Parameter_List` object the same two integers. (For the case where the parameter depends on two chemical objects, the parameter list would be given three integers.) During a simulation, the parameter name string and the object name string should not be used to retrieve the parameter. Looking up entries in a table using strings takes much longer than using an integer index. The string is merely a convenience in the event that the `Parameter_List` object is used in several subroutines. At the beginning of each subroutine, the relevant integer indices can be extracted using an easily remembered name.

An illustration of this process is shown in Fig. 5. The chemical object of interest is a single bond between two ordinary carbons, and the parameter of interest is the spring constant for the bond stretching motion. In Step 1, the bond type is entered under the name "`C–C Single Bond`," and the resulting index is placed into the variable `i_CCS`. In Step 2, all of the carbon–carbon single bonds in the simulation have their id's set to `i_CCS`. In Step 3, the parameter type is entered under the name `Bond Stretch` and the resulting index is placed into `i_BS`. In Step 4, the value of the parameter is stored, using the two indices from above. Finally, during the simulation, in Step 5, the parameter value is retrieved using the two indices. Chances are that Step 5 occurs within a loop that goes through all bonds; the index representing the bond type is extracted from the `id()` function of each bond.

```
typedef Structure< Atom, 2> Bond;
List< Bond> bonds;
// initialize bonds
Parameter_List< double> bond_parameters;
// Step 1
```

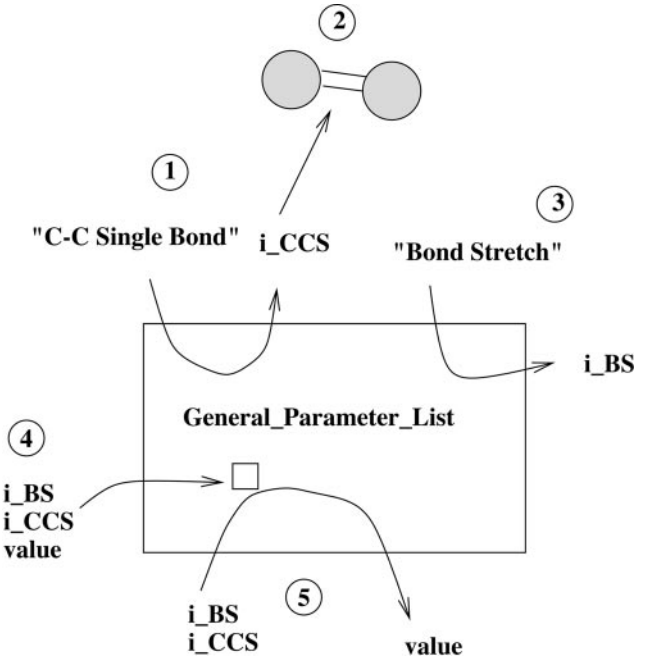

FIG. 5. Illustration of parameter list use.

```
size_t i_CCS = bond_parameters.Add_type ("C-C Single Bond");
// Step 2
// Figure out which bonds are C-C and set their id's to i_CCS
...
// Step 3
size_t i_BS = bond_parameters.Add_parameter( "Bond Stretch");
// Step 4
double C_C_single_bond_stretch = ...;
bond_parameters.Set_parameter( C_C_single_bond_stretch, i_BS, i_CCS);
// Repeat above steps for other parameters and bond types
...
// Step 5
FOR_ITEMS_IN_LIST( Bond< Atom>, single_bonds, bond)
  // retrieve bond-stretch parameter
  real k_bond = bond_parameters( i_BS, bond.id());
...
END_ITEMS_IN_LIST;
// or it can be done this way; which might be more efficient.
// access a vector containing all bond-stretch parameters
const std::vector< real>& bond_stretch_parameters =
  bond_parameters.parameter_vector( i_BS);
FOR_ITEMS_IN_LIST( Bond< Atom>, single_bonds, bond)
  real k_bond = bond_stretch_parameters[ bond.id()];
  ...
END_ITEMS_IN_LIST;
```

As can be seen, the parameters are closely coupled to the id numbers of the particles, chemical structures, and groups. For this reason, objects that read in Protein Database Files in preparation of a molecular-mechanical protein simulation will also need to read in the appropriate parameter files at the same time.

Ensemble Lists

In many applications, one might simultaneously perform simulations on several copies of the physical system of interest. These applications might range from simple cases of several one-copy simulations being run at the same time, to simulations where the different copies actually physically interact with each other. Examples of potential applications include the weighted-ensemble methods [17, 18] and the reaction pathway methods [8, 7]. These copies can be contained in an object of the templated class `Ensemble_List`. At the time of this writing, OOMPAA has only a serial implementation of the `Ensemble_List`, but future plans include a parallel version, built on top of the standard MPI (Message-Passing Interface) [23]. Even though the `Ensemble_List` is a container of objects, in much the same way as the OOMPAA `List` and the Standard Template Library `vector`, it has additional functions that collectively manipulate its members. Its syntax is designed so that when the parallel version is available, it will be possible to port existing code to parallel computers with minimal change. The parallel version uses the technique of *data parallelism* [24], in which the same commands are issued on each processor, but with different local data.

As a simple example, consider an ensemble list of 100 user-defined `Molecule` objects. The molecules are created, issued a command, and queried about their status, and some are deleted.

```
size_t n = 100;
Ensemble_List< Molecule> molecules( n);
// Commands to the parallel version to set the MPI_Communicator,
// customize placement of molecule objects on different processors
...
FOR_ITEMS_IN_ENSEMBLE( Molecule, molecules, molecule, i)
  molecule.Move();
END_ITEMS_IN_ENSEMBLE;
Vector< double> energy( n);
GET_ENSEMBLE_INFO( Molecule, molecules, molecule, i, energy)
  energy[i] = molecule.energy();
END_ENSEMBLE_INFO
double average_energy = energy.sum()/n;
FOR_ITEMS_IN_ENSEMBLE( Molecule, molecules, molecule, i)
  if (energy[i] > 3.0*average_energy)
    molecules.Remove_item( i);
END_ITEMS_IN_ENSEMBLE;
```

In the parallel version, the `Molecule` objects are apportioned among the different processors, but the vector `energy` is duplicated on each processor. Inside `FOR_ITEMS_IN_ENSEMBLE` loops, each processor applies the `Move` command only to its own objects, or deletes only its own objects. After the deletions occur, the `Molecule` objects are redistributed across the processors if the loads become too unbalanced. Inside the `GET_ENSEMBLE_INFO`

loop, each processor places the values from its `Molecule` objects into its copy of the `energy` array, and then exchanges its array contents with all of the other arrays on the other processors before the loop is exited. Some of the computation is duplicated across processors, but the assumption is made that the member functions of the `Molecule` objects take most of the processor time.

OTHER CORE FEATURES

OOMPAA handles physical units in a unified manner. First, it assumes *fundamental units* of angstroms for length, atomic mass units for mass, picoseconds for time, Kelvin for temperature, and the unit charge for electrical charge. Next, it provides a `Unit` class for building new physical units and two functions for converting to and from the fundamental units. Finally, it provides many built-in physical units and physical constants.

OOMPAA deals with the problem of passing functions as arguments to other functions in a consistent manner. In C++, there are three basic types of functions: unbound functions, such as those in C and Fortran, non-constant member functions of a class, and constant member functions of a class. Even if templates are used, it is not possible to pass, as an argument, a function of one kind to a function that expects a function of another kind. Thus, the programmer is faced with the prospect of writing three versions of the same function. Based on software by Hickey [13], OOMPAA includes templated classes that "wrap" the different types of function information into the same type of object, thus allowing the same code to serve for all three cases. Although the casual user who passes functions to functions does not need to be aware of this method, it removes a major obstacle to code development within OOMPAA.

ADDITIONS

In scientific modeling, there are two general uses for objects. The first use is to represent the objects being simulated; this is the main emphasis of OOMPAA's core. The second use of objects is that of "machines," or tools for carrying out a task. Several of these tools are already included in the additions to OOMPAA, and many more will be added in the future. Already included are functions for computing bond and torsion angles and their derivatives as functions of three and four positions, respectively. There are multivariate minimizer classes which can be applied to any function and generalized vector. There are different types of numerical integrators, including multiple-time step integrators, which can take any set of generalized vectors as inputs. OOMPAA has a *cell list*, which groups particles into cubic cells to facilitate the computation of pair-wise short-ranged forces. OOMPAA has classes for performing weighted-ensemble dynamics and annealing. There are classes for viewing lists of particles that make use of classes from the Visualization Toolkit (VTK), which itself is object-oriented, portable, and free.

OOMPAA also has several classes for performing continuum electrostatic calculations, which has been used successfully for eliminating atomic degrees of freedom in simulations of proteins. Included is a grid class for storing the input and results of finite-difference computations, electrostatic solver classes that sit on top of multigrid code written in C by Holst [15, 14], and classes that wrap lists of particles to make them appear to be solid bodies with specified dielectric values and charge densities.

PERFORMANCE

As mentioned above, performance is no longer a reason to avoid the use of C++ in scientific programming. Several advances in compiler technology have brought this about. First of all, good compilers do a thorough job of inlining functions, or placing the function body directly into the calling code. Inline functions can even call other inline functions. This is particularly important when using the accessor functions discussed above.

Second, good compilers can get rid of small, temporary objects and replace what remains with equivalent, Fortran-style code. The expression template technique depends heavily on this optimization as well as inlining; once the compiler strips out the expression objects, it can recognize the equivalence to a Fortran do-loop. Other small objects, such as the `Vector3` templated objects, can take advantage of small-object optimization. For this reason, `Vector3` arguments to functions in OOMPAA are usually passed by value rather than by reference; this coding practice encourages optimization.

Finally, several C++ compilers offer the `restrict` keyword to overcome the *aliasing* problem. This qualifier can be applied to a pointer, and constitutes a promise to the compiler not to use any other unrelated pointer to reference the same data as that referenced by the restricted pointer. The key advantage formerly held by Fortran is that there is only one way to refer to a memory location. C and C++, on the other hand, can refer to the same location many times by using pointers, so compilers must make much broader assumptions that inhibit optimization and vectorization. For example, a C function that adds two vectors together and places the result in a third has no way of knowing, during compilation, whether either of the input vectors overlaps with the output vector, so such a function might not take full advantage of a vector processor. However, using restricted pointers to point to the vectors' data alleviates this problem. OOMPAA's lists use restricted pointers, particularly in the looping macros. The Blitz++ vectors use restricted pointers, as well.

As a non-trivial comparison of OOMPAA with hand-coded Fortran, consider the computation of the Lennard-Jones energy of a large group of particles. One way to proceed is to loop through every pair of particles and sum up the Lennard-Jones terms:

```
List< MC_Particle> atoms(62500);
// place into FCC lattice
...
real energy = 0.0;
FOR_PAIRS_IN_LIST( MC_Particle, atoms, atom1, atom2)
  const Vector3< real> d = atom1.position() - atom2.position();
  real r2 = 1.0/dot( d,d);
  real r6 = r2*r2*r2;
  real r12 = r6*r6;
  real lj = r12 - 2.0*r6;
  energy += lj;
END_PAIRS_IN_LIST;
```

Another way is to use a cell list [11], which discretizes space into an array of cubes and assigns each particle to a cell. The pairwise energy term is assumed to be zero beyond a certain distance, and the width of each cell is the same as this cutoff radius. Thus, pairwise interactions for a particle need only be computed with the other particles in its own cell and the particles in the immediately neighboring cells. Each cell contains pointers to its member particles. The code below creates the cell list in the form of a `Cell_Collection` object and

tells it the desired spacing. From the list of particles, the cell list figures out how many cells are needed and where they are placed, and how many particles can be referenced by each cell. Then, the cell list allocates memory for pointers, and assigns the particles to its cells. Finally, the macro FOR_PAIRS_IN_NEIGHBORING_CELLS sequentially references each cell, in each cell it sequentially references each particle, and for each particle it sequentially references all pairs formed from (1) the current particle, and (2) each other particle that is in the same cell, and the immediately neighboring cells ahead of the current cell. No pairs are double-counted.

```
List< MD_Particle> atoms(864000);
// place into FCC lattice
...
Cell_Collection< MC_Particle> cells;
cells.Set_spacing( 2.5);
cells.Set_corners( atoms);
size_t nc = cells.required_cell_size( atoms);
cells.Set_n_in_cell( nc);
cells.Allocate_cells();
cells.Add_items( atoms);
real energy = 0.0;
FOR_PAIRS_IN_NEIGHBORING_CELLS( MC_Particle, cells, atom1, atom2)
  const Vector3< real> d = atom1.position() - atom2.position();
  real r2 = 1.0/dot( d,d);
  real r6 = r2*r2*r2;
  real r12 = r6*r6;
  real lj = r12 - 2.0*r6;
  energy += lj;
END_PAIRS_IN_NEIGHBORING_CELLS;
```

Of course, in a real simulation, this can be combined with a *Verlet list* for greater efficiency. In neither example was any attempt made to improve the performance by changing the order of pair traversals in order to account for the data cache. Future versions of OOMPAA may have this feature built into the macros, with no change in syntax.

For both cases, Fortran programs were written to carry out the same computations. The memory layout of the data was made to resemble, as closely as possible, that of the OOMPAA implementation. The pairwise summation used 62,500 particles, and the cell list summation used 864,000 particles. The computations were performed on a Silicon Graphics Indigo 2 R10000 running the Irix 6.2 operating system. The Fortran code was compiled using the native SGI Fortran 77 compiler, and the C++ code was compiled using the KAI 3.2d compiler from Kuck and Associates. Full optimization was used for all cases. The timing results are summarized in Fig. 6; the C++ code has a speed of execution that is at

	Fortran	C++
Pair Summation	56.4	57.5
Cell List	29.2	32.3

FIG. 6. Benchmark execution times in seconds.

least 90% of the speed of the Fortran code. It should be noted that this level of performance is unlikely to be reached by most other C++ compilers at this time.

CONCLUSIONS

Using object-oriented programming techniques, it is now possible to write molecular simulation code that is both flexible and rapidly executing. C++ class libraries such as OOMPAA will be very useful for writing more complex molecular simulations. OOMPAA source code can be found at the web site `http://chemcca10.ucsd.edu/~oompaa/`.

ACKNOWLEDGMENTS

This work has been supported in part by the NIH, the NSF, and the MetaCenter Program of the NSF Supercomputer Centers. G.A.H. is the recipient of NIGMS Postdoctoral Fellowship 1-F32-GM12862-01.

REFERENCES

1. J. Aqvist and A. Warshel, Simulation of enzyme reactions using valence bond force fields and other hybrid quantum/classical approaches, *Chem. Rev.* **93**, 2523 (1993).
2. P. Bala, P. Grochowski, B. Lesyng, and J. A. McCammon, Quantum-classical molecular dynamics and its computer implementation, *Comput. Chem.* **19**, 155 (1995).
3. M. Bhandarkar, R. Brunner, A. Dalke, A. Gursoy, W. Humphrey, N. Krawetz, M. Nelson, J. Phillips, and A. Shinozaki, URL http://www.ks.uiuc.edu/Research/namd/Namd2.html, 1998.
4. G. Booch, *Object-Oriented Analysis and Design*, 2nd ed. (Addison–Wesley, Reading, MA, 1994).
5. R. Car and M. Parrinello, Unified approach for molecular dynamics and density-functional theory, *Phys. Rev. Lett.* **55**, 2471 (1985).
6. T. W. Clark, R. V. Hanxleden, J. A. McCammon, and L. R. Scott, Parallelization using spatial decomposition for molecular dynamics, in *Scalable High Performance Computing Conference*, IEEE Comput. Soc., Los Alamitos, CA, 1994), pp. 95–102.
7. C. Dellago, P. G. Bolhuis, and D. Chandler, Efficient transition path sampling: Application to Lennard-Jones cluster rearrangements, *J. Chem. Phys.* **108**, 9236 (1998).
8. R. Elber, A. Roitberg, C. Simmerling, R. Goldstein, H. Li, G. Verkhivker, C. Keasar, J. Zhang, and A. Ulitsky, MOIL: A program for simulations of macromolecules, *Comput. Phys. Commun.* **91**, 159 (1995).
9. A. H. Elcock, M. J. Potter, and J. A. McCammon, Application of Poisson–Boltzmann solvation forces to macromolecular simulations, in *Computer Simulation of Biomolecular Systems*, Vol. 3, edited by W. F. van Gunsteren, P. K. Weiner, and A. J. Wilkinson (Kluwer Academic, Dordrecht, 1997, pp. 244–261.
10. M. J. Field, P. A. Bash, and M. Karplus, A combined quantum mechanical and molecular mechanical potential for molecular dynamics simulations, *J. Comput. Chem.* **11**, 700 (1990).
11. D. Frenkel and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications* (Academic Press, San Diego, 1996).
12. M. K. Gilson, M. E. Davis, B. A. Luty, and J. A. McCammon, Computation of electrostatic forces on solvated molecules using the Poisson–Boltzmann equation, *J. Phys. Chem.* **97**, 3591 (1993).
13. R. Hickey, Callbacks in C++ using template functors, *C++ Report* **7**, 42 (1995).
14. M. Holst, URL http://sdna3.ucsd.edu/mholst/codes/codes.html, 1998.
15. M. Holst and F. Saied, Numerical solution of the nonlinear Poisson–Boltzmann equation: Developing more robust and efficient methods, *J. Comput. Chem.* **16**, 337 (1995).
16. D. Horvath, D. van Belle, G. Lippens, and S. J. Wodak, Development and parametrization of continuum solvent models. I. Models based on the boundary element method, *J. Chem. Phys.* **104**, 6679 (1996).
17. G. A. Huber and S. Kim, Weighted-ensemble Brownian dynamics simulations for protein association reactions, *Biophys. J.* **70**, 97 (1996).

18. G. A. Huber and J. A. McCammon, Weighted-ensemble simulated annealing: Faster optimization on hierarchical energy surfaces, *Phys. Rev. E* **55**, 4822 (1997).
19. M. Nelson, *C++ Programmer's Guide to the Standard Template Library* (IDB Books Worldwide, Foster City, CA, 1995.
20. A. Robison, C++ gets faster for scientific computing, *Comput. Phys.* **10**, 458 (1996).
21. W. Schroeder, K. Martin, and B. Lorensen, URL http://www.kitware.com/vtk.html, 1998.
22. J. L. Smart, T. J. Marrone, and J. A. McCammon, *J. Comput. Chem.* (1997).
23. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference* (MIT Press, Cambridge, MA, 1997).
24. S. R. Kohn and S. B. Baden, A parallel software infrastructure for structured adaptive mesh methods, in *Proceedings, Supercomputing '95, San Diego, 1995*.
25. G. van Rossum, URL http://www.python.org, 1998.
26. T. Veldhuizen, Expression templates, *C++ Report* **7**, 26 (1995).
27. T. Veldhuizen, URL http://monet.uwaterloo.ca/blitz/benchmarks/, 1998.
28. T. Veldhuizen, URL http:/monet.uwaterloo.ca/blitz/, 1998.