PROTOMOL, an Object-Oriented Framework for Prototyping Novel Algorithms for Molecular Dynamics

THIERRY MATTHEY
Department of Informatics, University of Bergen, N-5020 Bergen, Norway
and
TREVOR CICKOVSKI, SCOTT HAMPTON, ALICE KO, QUN MA, MATTHEW NYERGES, TROY RAEDER, THOMAS SLABACH and JESÚS A. IZAGUIRRE
Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, United States

PROTOMOL is a high-performance framework in C++ for rapid prototyping of novel algorithms for molecular dynamics and related applications. Its flexibility is achieved primarily through the use of inheritance and design patterns (object-oriented programming). Performance is obtained by using templates that enable generation of efficient code for sections critical to performance (generic programming). The framework encapsulates important optimizations that can be used by developers, such as parallelism in the force computation. Its design is based on domain analysis of numerical integrators for molecular dynamics (MD) and of fast solvers for the force computation, particularly due to electrostatic interactions. Several new and efficient algorithms are implemented in PROTOMOL. Finally, it is shown that PROTOMOL's sequential performance is excellent when compared to a leading MD program, and that it scales well for moderate number of processors. Binaries and source codes for Windows, Linux, Solaris, IRIX, HP-UX, and AIX platforms are available under open source license at `http://protomol.sourceforge.net`.

Categories and Subject Descriptors: D.1.5 [**Software**]: Programming Techniques—*Object-Oriented Programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain Specific Architectures*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*; G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations—*Multistep and multivalue methods*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Algorithm Design and Analysis, Efficiency, Parallel and vector implementations, User interfaces*; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*Chemistry, Physics*; J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and Genetics*

General Terms: Object-Oriented Scientific Computing, Computer Simulations

Additional Key Words and Phrases: fast electrostatic methods, incremental parallelism, molecular dynamics, multi-grid, multiple time stepping integration, object-oriented framework.

This research is partially supported by a National Science Foundation Biocomplexity Grant IBN-0083653, a National Science Foundation CAREER Award ACI-0135195, the Center for Applied Mathematics at the University of Notre Dame, and partly by the Research Council of Norway and the Norwegian High Performance Computing Consortium (NOTUR). Corresponding authors' email addresses: `matthey@ii.uib.no`, `izaguirr@cse.nd.edu`. Qun Ma's current address: Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 08854; `qma@oak.njit.edu`.

1. INTRODUCTION

PROTOMOL is a framework for rapid development of efficient algorithms and applications in molecular modeling, particularly of biomolecules such as proteins and DNA. It supports state-of-the-art molecular dynamics (MD) and N-body algorithms which improve the speed and efficiency of these simulations. Because MD simulations are time consuming, taking from hours to weeks on modern workstations and supercomputers, the design of PROTOMOL aims at both flexibility and high performance.

Many robust and highly optimized programs exist for MD simulation of biomolecules [Brooks and Hodošček 1992; Vincent and Merz 1995; Kalé et al. 1999; Brünger 1992; Tuckerman et al. 2000]. However, these programs are not appropriate for algorithm development and training of new students and junior researchers due to their extreme complexity. Without an appropriate platform, novel algorithm development and training are difficult to perform. Also, few of the academic programs are easy to use. This intimidates potential users, who have to learn numerous implementation details of the simulation methods to be able to prepare all the input files and analyze the results. In particular, the more sophisticated algorithms have multiple parameters that interact in subtle ways. PROTOMOL automatically detects appropriate parameters for many of the algorithms implemented. Thus, algorithms successfully incorporated into this framework can be more easily incorporated into the most popular MD programs.

PROTOMOL provides components to handle input/output (I/O), graphical user interface (GUI) and visualization, as well as optimized algorithmic frameworks for easy incorporation and efficient implementation of new algorithms. An important optimization encapsulated in PROTOMOL is parallelism at a modest scale (for example, clusters or medium size symmetric multi-processors with typical size of 32 to 64 nodes), which addresses the needs of many MD users.

As proof of the flexibility of PROTOMOL we present examples of sophisticated MD and N-body algorithms implemented within the framework. The program is also efficient; its sequential performance is comparable to leading MD packages, and it scales to moderately sized clusters[1] or symmetric multi-processors. PROTOMOL has been successfully used as an instructional tool in university courses. It is open source, and binaries and source codes for Windows, Linux, Solaris, IRIX, HP-UX, and AIX platforms are available on its web page, `http://protomol.sourceforge.net`.

We show the design, implementation and evaluation of the framework, aiming to demonstrate how object-oriented and generic design help us achieve our twofold goal of flexibility and efficiency. We have presented partial reports on PROTOMOL in [Matthey 2002; Matthey and Izaguirre 2001; Matthey et al. 2003].

2. BACKGROUND

MD typically solves Newton's equations of motion:

$$m_i \frac{\mathrm{d}^2}{\mathrm{d}t^2} \vec{r}_i(t) = \vec{F}_i(t), \tag{1}$$

where the mass of the i^{th} atom is m_i, its atomic position at time t is $\vec{r}_i(t)$, and its instantaneous force is $\vec{F}_i(t)$.

[1]Clusters with reasonably fast interconnect, enabling general parallel computing.

```
MD Simulation:
(1) Pre-processing: Construct initial configuration of positions, velocities, and forces.
(2) loop 1 to number of steps
    (a) half kick: Update velocities (by a half time step $\delta t/2$)
    (b) drift: Update positions (by a full time step $\delta t$)
    (c) evaluate forces on each particle
    (d) half kick: Update velocities (by a half time step $\delta t/2$)
(3) Post-processing
```

Algorithm 1. Pseudo-code of an MD simulation.

The bulk of computation in an MD simulation is spent in evaluating the force $\vec{F}_i$ of Eq. (1), which for conservative forces is defined as the gradient of the potential energy, or more generally as

$$\vec{F}_i = -\nabla_i U(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) + \vec{F}_i^{\,\mathrm{extended}}, \tag{2}$$

where U is the potential energy, $\vec{F}_i^{\,\mathrm{extended}}$ an extended force (e.g., velocity-based friction) and N the total number of atoms in the system. Typically, the potential energy is given by

$$U = U^{\mathrm{bonded}} + U^{\mathrm{non-bonded}}, \tag{3}$$

$$U^{\mathrm{bonded}} = U^{\mathrm{bond}} + U^{\mathrm{angle}} + U^{\mathrm{dihedral}} + U^{\mathrm{improper}}, \tag{4}$$

$$U^{\mathrm{non-bonded}} = U^{\mathrm{electrostatic}} + U^{\mathrm{Lennard-Jones}}. \tag{5}$$

The bonded energy is a sum of $\mathcal{O}(N)$ terms that define the covalent bond interactions which model flexible molecules. The non-bonded energy is a sum of $\mathcal{O}(N^2)$ pair–wise terms over all atoms. $U^{\mathrm{electrostatic}}$ represents the well-known Coulomb potential and $U^{\mathrm{Lennard-Jones}}$ models a van der Waals attraction and a hard-core repulsion. Whereas the latter can usually be evaluated for a reduced range of distance, the long range effects of the electrostatic interactions are responsible for the stability of biomolecules, and are important in a variety of non-biological materials.

Newton's equations of motion are often integrated numerically by the leapfrog or Verlet method, which is single time stepping (STS), second-order accurate, time-reversible, and symplectic. Despite its low order of accuracy, it has excellent energy conservation properties and is computationally cheap [Skeel 1999; Verlet 1967]. The choice of an integrator that imposes different constraints on the equations of motion changes the ensemble in which sampling or dynamics is simulated. PROTOMOL allows users to choose from among several such ensembles, e.g., constant energy, constant temperature, and constant pressure. Algorithm 1 describes a complete MD simulation using leapfrog. Basically, it consists of the loop numerically solving the equations of motion, the evaluation of forces on each particle, and some additional pre- and post-processing.

2.1 Numerical integrators

When integrating Newton's equations of motion numerically, the fastest motions restrict the time step to satisfy stability conditions. For an MD simulation of solvated biological molecules, the fastest motion is around 10 femtoseconds (1 fs = 10^{-15} s) and the stability limit of leapfrog is 2.25 fs [Skeel 1999]. A typical step size for leapfrog method is 1 fs, and simulations may consist of millions of time steps.

```
One step of Impulse:
(1) half a long kick: update velocities using “slow” forces (by a half long-step $\Delta t/2$)
(2) vibration: propagate positions and velocities using “fast” forces (by a full long-step $\Delta t$),
    using for example, Verlet/leapfrog with short time step $\delta t \equiv \Delta t/k$
(3) evaluate slow forces on each particle
(4) half a long kick: update velocities using “slow” forces (by a half long-step $\Delta t/2$)
```

Algorithm 2. Pseudo-code for one step of Impulse.

```
One step of Targeted MOLLY:
(1) half a mollified kick: update velocities using mollified “slow” forces at time-averaged
    positions (by a half long-step $\Delta t/2$)
(2) vibration: propagate positions and velocities using “fast” forces and pair–wise Langevin
    damping (by a full long-step $\Delta t$)
(3) time averaging: do a time-averaging of positions using “fastest” forces, and evaluate
    “slow” forces using time averaged positions
(4) half a mollified kick: update velocities using mollified “slow” forces at time-averaged
    positions (by a half long-step $\Delta t/2$)
```

Algorithm 3. Pseudo-code for one step of Targeted MOLLY discretization.

MD systems consist of different force types and have different time scales of dynamics. Multiple time stepping (MTS) integrators address the different time scales by splitting the forces – if possible – by frequencies and incorporating them with appropriate time steps. Bonded forces typically are considered “fast”, while non-bonded forces have both “fast” and “slow” components. A common MTS integrator is Verlet-I [Grubmüller 1989]/r-RESPA [Tuckerman et al. 1992]/Impulse integrator (hereafter referred to as Impulse). One step of this algorithm is shown in Algorithm 2.

PROTOMOL contributes several new MTS algorithms. For example, the momentum-preserving Targeted MOLLY (TM) [Ma and Izaguirre 2003c; 2003a; 2003b], which allows use of time steps of 16 fs for the “slow” forces, and 2 fs for the “fast” forces. This represents a significant speedup over Verlet (which uses 2 fs for everything) and Impulse (which allows 4 fs for the slow force). The speedup comes about because the “slow” forces are most expensive to evaluate. Algorithm 3 gives one step of this integrator.

The design of the integrators in PROTOMOL has the following goals: (1) To enable the user to compose arbitrary MTS integrators at runtime, by choosing the number of levels, forces and integrators; (2) To accommodate different integrators under a consistent interface; and (3) To allow pre-processing and post-processing steps for some integrators.

2.2 Boundary conditions

Boundary conditions specify how the molecules interact with their surroundings. Two commonly used boundary conditions are: [Allen and Tildesley 1987, pp. 24-32, 156ff.]:

(1) Vacuum: this models isolated systems. It is convenient for understanding the behavior of individual macromolecules. A spherical or cylindrical constraint force may be used to confine the system to a sphere or a cylinder, respectively.

(2) Periodic boundary conditions (PBC): the system experiences forces as if it was part of an infinitely large bulk. Periodic cells are used, and when evaluating forces multiple

images of particles are considered. The shape of the original cell may be cubic or hexagonal or any other number of space filling shapes.

2.3 Fast electrostatic algorithms

Bonded interactions can be evaluated in linear time. Non-bonded interactions are more computationally expensive. Most MD program optimizations happen here. A common optimization for non-bonded interactions is the use of cutoffs to limit the spatial domain of pair–wise interactions; switching functions bring energies and forces smoothly to zero at the cutoff point. Cutoff computation can be accelerated through the use of cell lists or pair lists, cf. [Frenkel and Smit 2002]. For strongly charged systems, however, the electrostatic interactions play a dominant role, e.g., in protein folding, ligand binding, and ion crystals. Several algorithms for fast electrostatic force evaluation are implemented in PROTOMOL. These include Ewald summation ($\mathcal{O}(N^{3/2})$ time complexity) [Ewald 1921; Fincham 1994], smooth Particle Mesh Ewald (PME) summation ($\mathcal{O}(N \log N)$) [Essmann et al. 1995], and a multi-grid (MG) summation ($\mathcal{O}(N)$)[Brandt and Lubrecht 1990; Skeel et al. 2002].

In the case of non periodic electrostatic forces, the problem is that of solving

$$U^{\text{electrostatic}}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j \neq i} \frac{q_i q_j}{\varepsilon_0 \left| \vec{r}_j - \vec{r}_i \right|}, \tag{6}$$

where the partial charge of the atom is q, and the dielectric coefficient is ε_0. This is the N-body problem.

In the case of periodic boundary conditions, the problem to be solved is

$$U^{\text{electrostatic}}(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{\vec{m} \in \mathbb{Z}^3}{}^{\prime} \frac{q_i q_j}{\varepsilon_0 \left| \vec{r}_j - \vec{r}_i + \vec{m}L \right|}, \tag{7}$$

where the sum is over all periodic cells with index $\vec{m}$, with self-interactions excluded, and L is the length of a periodic box cell with dimensions $L \times L \times L$. The primed sum excludes interactions for pairs that are in the exclusion list inside the original MD cell ($\vec{m} = 0$). This is a conditionally convergent sum, and a physically meaningful interpretation is given in [Leeuw et al. 1980]. The Ewald summation transforms this problem into the sum of two rapidly convergent sums, one in real space and the other in reciprocal or Fourier space.

The particle mesh Ewald (PME) method [Darden et al. 1993] splits the Ewald summation such that the real part is solved using cutoff in $\mathcal{O}(N)$, and interpolates the charges onto a mesh, thus allowing the use of FFT for the evaluation of the Fourier space part at a complexity of $\mathcal{O}(N \log N)$.

3. FRAMEWORK DESIGN

Four main requirements for PROTOMOL were addressed during the design of the framework. These reflect the submodules described in Algorithm 1.

(1) Allow end-users to compose multiple time stepping integrators dynamically. This allows them to *experiment* with different *integration schemes*. MTS methods require careful fine-tuning to get the full benefit of the technique.

(2) Allow developers to easily integrate and evaluate *novel force algorithm* schemes. For example, the force design allows the incorporation of mesh-based and MG methods,

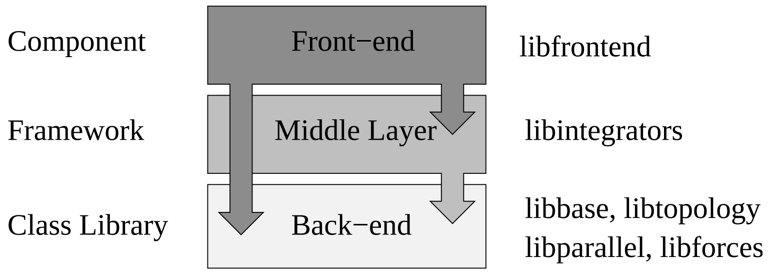


Fig. 1. Main layers of the component-based framework PROTOMOL.

or of new force kernels that compute the potential energy and its gradient.

(3) Develop an encapsulated parallelization approach, where sequential and parallel components coexist. This way, developers are not forced to consider the distributed nature of the software. Parallelism itself is based on range computation and a master-slave concept [Matthey and Izaguirre 2001].

(4) Provide facilities to compare accuracy, stability, and run-time efficiency of MD algorithms and methods. This allows users to examine various algorithms side by side, as well as determine a selection of optimal parameters.

For the design of the component-based framework PROTOMOL, three different modules were identified: front–end, middle–layer and back–end, see Fig. 1. The front-end provides *components* to compose and configure MD applications. The components are responsible for the actual MD simulation set up with its integration scheme and particle configuration. This layer is strongly decoupled from the rest to the extent that the front–end can be replaced by a scripting language. The middle layer is a *white-box framework* for numerical integration reflecting a general MTS design (see Section 3.2). The back-end is a *class library* carrying out the force computation and providing basic functionalities (see Section 3.3). It has a strong emphasis on run-time efficiency. PROTOMOL also implements interactive and steered MD interfaces to the visualization program VMD[2], cf. [Isralewitz et al. 2001].

A typical PROTOMOL user interacts with the front end, either through a console, a GUI, or an external visualization program. Depending on the user defined simulation protocol, the appropriate objects are generated and used for simulating the system. Possible output includes observables such as total energy (plots or files), trajectories of molecules (visualized or stored), and various other files. The user may use haptic interfaces to steer MD simulations. The flow of execution of a PROTOMOL simulation is illustrated in Fig. 2.

3.1 Front-end

The front-end mainly performs pre- and post-processing as in Algorithm 1. It supports a range of different I/O facilities for common file formats to construct the initial state and to perform output of interest. The simulation object contains positions, velocities, forces, energies, and molecular topology objects. The molecular topology defines the connectivity

[2] `http://www.ks.uiuc.edu/Research/vmd/`

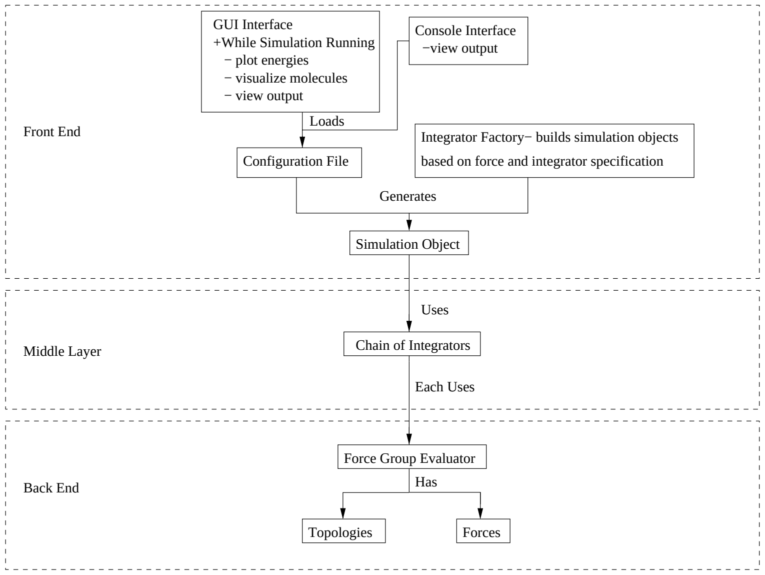


Fig. 2. Snapshot of the flow of execution in a typical MD simulation using PROTOMOL

and interaction strengths of atoms in a molecule. The topology is assumed to be static, e.g., covalent bonds of a molecule do not break during a simulation. Furthermore, the front-end is in charge of constructing the desired integrator scheme and the associated forces to solve Eq. (1). This is described in a configuration file using a language specific to integration and force evaluation. This language allows dynamic definition of arbitrary integration and force evaluation algorithms without modifying the code. A snapshot of a simple GUI application using PROTOMOL is shown in Fig. 3. Other important front end libraries are the integrator and force factories, which translate the user specification of integrator and forcing scheme into an executable object.

3.2 Time stepping integration

In order to allow user composition of high-performance MTS integrators a twofold solution is implemented: (i) an integrator definition language (IDL) that allows the user to compose arbitrary chains of MTS integrators and associate force evaluation algorithms at each level, and (ii) an integrator hierarchy that efficiently supports the IDL. Object composition and inheritance are used to provide reusable code, cf. [Gamma et al. 1995, p. 18]. Fig. 4 shows the abstract syntax for the IDL. Note that level 0 corresponds to the innermost integrator, which should always be an STS integrator. As an example, Program 1 shows the definition of a three level Impulse integrator in IDL. The actual frequency of evaluation of a force at a given level is recursively defined by the product of the current *cycle length* and the frequency of the next inner integrator. Thus, in this example, bond and angle forces are evaluated every 0.5 fs using leapfrog; dihedrals and impropers, and the short-range Coulomb and Lennard–Jones forces are evaluated every $4 \times 0.5 = 2$ fs; and finally the long range (reciprocal) part of the Coulomb force is evaluated every $2 \times 4 \times 0.5 = 4$ fs.

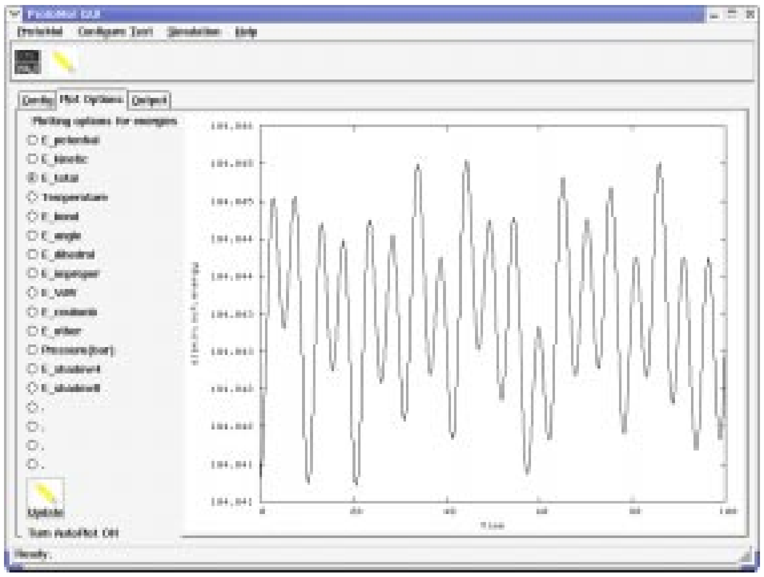


Fig. 3. Snapshot of a PROTOMOL GUI application. Displayed is the total energy for a simulation of decalanine.

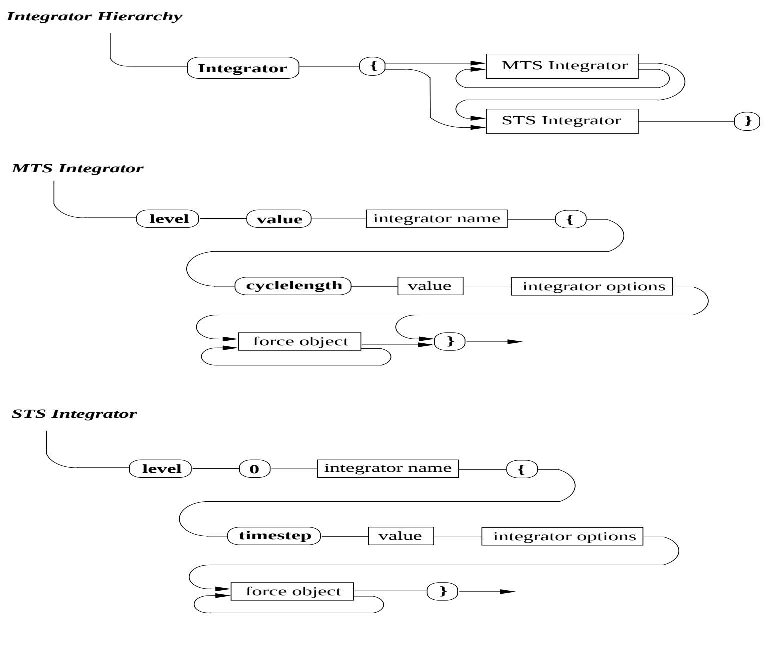


Fig. 4. Syntax chart for PROTOMOL's integrator definition language (IDL).

```
Integrator {
  level 2 Impulse { # Long-range electrostatics
    cyclelength 2
    force Coulomb -algorithm Ewald -reciprocal
  }
  level 1 Impulse { # Medium-range forces
    cyclelength 4
    force Coulomb -algorithm Ewald -real -correction
    force Improper, Dihedral
    force Lennard-Jones -algorithm Cutoff -switchingFunction C2
  }
  level 0 Leapfrog { # Shortest-range forces
    timestep 0.5 # (fs)
    force Bond, Angle
  }
}
```

Program 1. Three level Impulse MTS using PROTOMOL's integrator definition language.

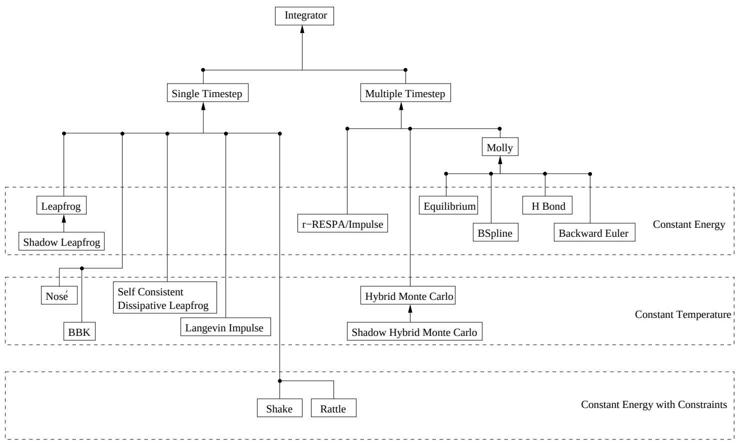
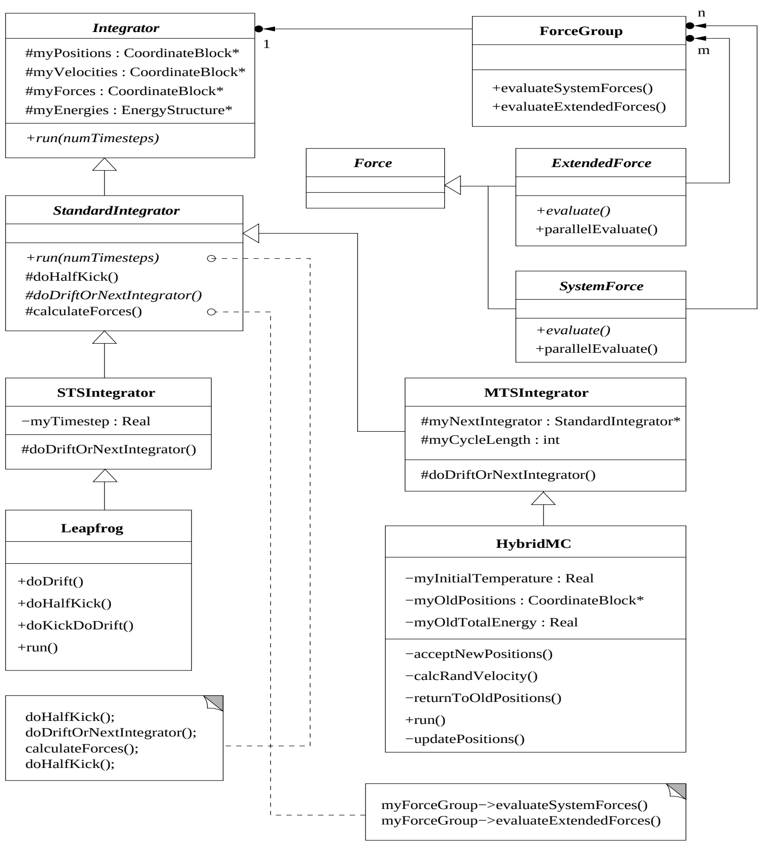


Fig. 5. Partial view of the hierarchy of numerical integrators in PROTOMOL.

By using the IDL in PROTOMOL, a user can select the integrator and forces to be used at each level. The integrator hierarchy that is supported by PROTOMOL is shown schematically in Fig.5. An integrator can be single time stepping (STS) or MTS. Different STS integrators may be used to define different ensembles and constraints. For example, constrained NVE dynamics can be defined by using the STS integrator SHAKE [Ryckaert et al. 1977] or RATTLE [Andersen 1983]. One can sample from the NVT ensemble by using the Nosé-Hoover, BBK [Brünger et al. 1982], Langevin Impulse [Skeel and Izaguirre 2002], or self-consistent leapfrog [Pagonabarraga et al. 1998] integrator [Nosé 1984; Martyna et al. 1992]. Also, different MTS integrators have different stability and accuracy properties. In order of increasing stability, one can cite Impulse, MOLLY, the hybrid Monte Carlo (HMC) method and the shadow HMC (SHMC). For example, SHMC achieves an order of magnitude speedup over HMC, cf. [Hampton and Izaguirre 2004]. Finally, each integrator has a set of forces to be evaluated at each level, called a *force group* in our framework. This gives the user great flexibility in partitioning forces across different levels.

PROTOMOL integrators can easily be extended to multiple levels, cf. [Biesiadecki and



```
doHalfKick();
doDriftOrNextIntegrator();
calculateForces();
doHalfKick();
```

```
myForceGroup−>evaluateSystemForces()
myForceGroup−>evaluateExtendedForces()
```

Fig. 6. Collaboration diagram between the integrator object and the force objects. The diagram illustrates the integrator hierarchy with two concrete integrator classes representing MTS and STS integrators. The force hierarchy is pruned at the top level.

Skeel 1993]. The behavior of Algorithms 1–3, Verlet/leapfrog, Impulse and MOLLY, can be abstracted using this form:

(1) `doHalfKick()`

(2) `doDriftOrNextIntegrator()`

(3) `calculateForces()`

(4) `doHalfKick()`.

The function `calculateForces()` evaluates each force in the *force group*. The function `doHalfKick()` updates the velocities by half a time step.

The integrator class hierarchy is designed using inheritance. Fig. 6 shows a close-up of the integrator hierarchy. The collaboration between integrators and forces is as follows:

(1) At the base of this hierarchy there is an abstract integrator class. It specifies that every integrator has to provide a `run()` method. This method runs some number of MD steps. This is an instance of a virtual or dynamically bound function. Using virtual

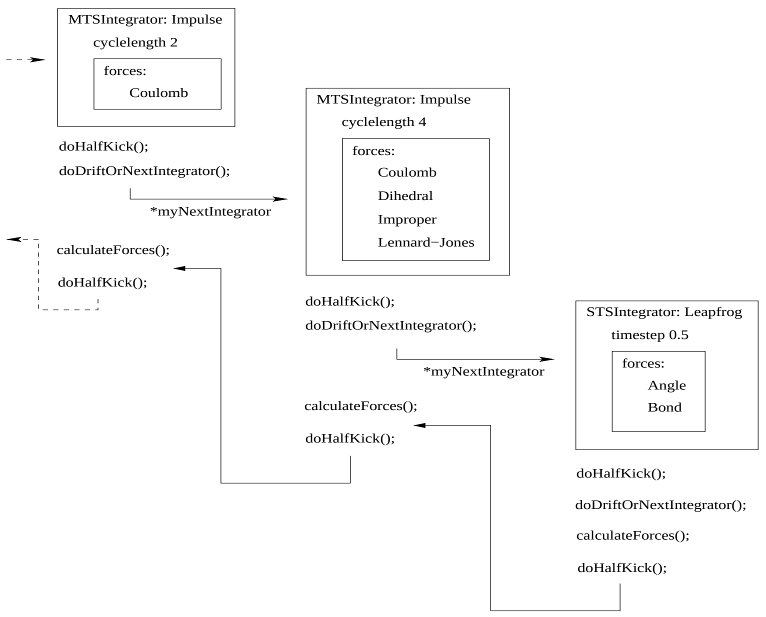


Fig. 7. Chain of integrator objects implementing multiple time stepping schemes. MTS differs from STS in that it calls recursively the next inner integrator before evaluating its forces. The recursion is terminated by an STS integrator.

functions, PROTOMOL can define operations that apply to any integrator, and at run-time the appropriate, specialized, `run()` method for specific integrators is invoked. The beauty of virtual functions is that they allow existing code to be extended without modifications.

(2) Integrators based on splitting of the update to velocities are called `StandardIntegrators`. All integrators discussed in this paper fall into this category, and their `run()` methods call default methods `doHalfKick()` and `calculateForces()`. However, they call a virtual function `doDriftOrNext Integrator()` which at run-time performs the appropriate action.

(3) The function `doDriftOrNextIntegrator()` is the key to the abstraction. For an STS integrator, it updates the positions for a full step (cf. Algorithm 1), whereas for an MTS integrator such as Impulse, it executes the next level of integration. Fig. 6 shows that MTS integrators have a pointer to the next integrator in the chain (`*myNextIntegrator`).

(4) Integrators such as leapfrog inherit much functionality from the hierarchy, but may redefine routines, and add data needed for their functionality. For example, the hybrid Monte Carlo integrator in the same figure adds a set of old positions and energy, which is restored if the MD integration is rejected.

At run time, an integrator definition in IDL is interpreted, and an "integrator factory" sets up the correct chain of integrators defined by the user. An example of the links and flow of the 3 level MTS integrator set up in Program 2 is shown in Fig. 7. The level 2 integrator starts the execution cycle. Because this integrator is MTS, it calls the next

```
Integrator {
  level 1 BSplineMOLLY { # Long-range electrostatics
    cyclelength 8
    force Coulomb -algorithm Ewald -reciprocal
  }
  level 0 SCDLeapfrog { # Fast varying forces
    timestep 2.0 # [fs]
    gamma 4.0
    temperature 300
    force Bond, Angle, Improper, Dihedral
    force Coulomb -algorithm Ewald -real -correction
    force LennardJones -algorithm Cutoff -switchingFunction C2
                        -cutoff 6.5 -switchon 4.0
  }
}
```

Program 2. A two-level Targeted MOLLY integrator hierarchy using IDL.

integrator instead of updating the positions itself. Likewise, the level 1 integrator also calls the next integrator. The level 0 integrator is leapfrog and it ends the chain by calling `doDrift()`, which updates the positions. Only bond and angle forces are computed at this level, which is executed 4 times. Once execution at this level is finished, the level 1 integrator is completed. Level 1 has 4 forces to evaluate. When the cycle finishes at this level, the process is repeated because the cycle length defined in level 2 is 2. Then, the single force of level 2 is computed and a full integration cycle completes. The forces to be used by an integrator object are encapsulated in `ForceGroup`, making it possible to perform some pre- and post-processing, e.g., to distribute the force computation among different processors.

In summary, there are two ways of extending the framework: The first is using the IDL, whereby the user can define new combinations of existing components. This is illustrated in Program 2, which implements the aforementioned two-level Targeted MOLLY integration. The second way of extending the framework is to actually add a class to the integrator inheritance hierarchy in PROTOMOL, which is simplified by the functionalities that already exist in the framework. An example of extension of the integrator hierarchy is in Section 4.1.4.

3.3 Force computation

The design of forces, particularly non-bonded forces, tries to maximize performance. There is a base class specifying the interface for every force evaluation. Five requirements (or customizable options, policies) for the evaluation of pair–wise interactions of non-bonded forces are proposed:

(R1) Algorithms to find or define the pairs or n-tuples to be evaluated from the set of all particles, e.g., cutoff, Ewald, or all pair computations. Closely associated with this algorithm is the algorithm to manage cells (R3), and the kernel function (R4).

(R2) Types of boundary conditions defining how to compute distances in the system, and how to retrieve actual positions, e.g., periodic or vacuum.

(R3) Cell managers to retrieve efficiently the spatial information of each particle. Cells can be cubic, hexagonal, or any space filling shape. This has $\mathcal{O}(1)$ complexity.

(R4) Kernels that compute a potential energy term, its gradient (the force), and optionally a matrix of second derivatives (the Hessian). A kernel operates on either a pair of atoms or an n-tuple defined by (R1).

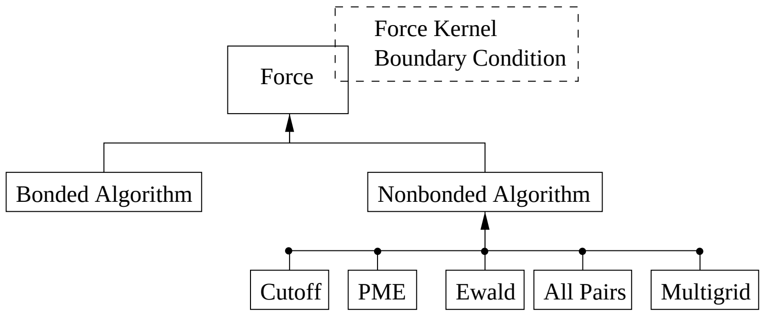


Fig. 8. Hierarchy of force computation algorithms. There are several non-bonded algorithms available, and some of these can be parameterized to work with different kernels (e.g., cutoff works with Coulomb or Lennard–Jones) and boundary conditions (e.g., periodic or vacuum).

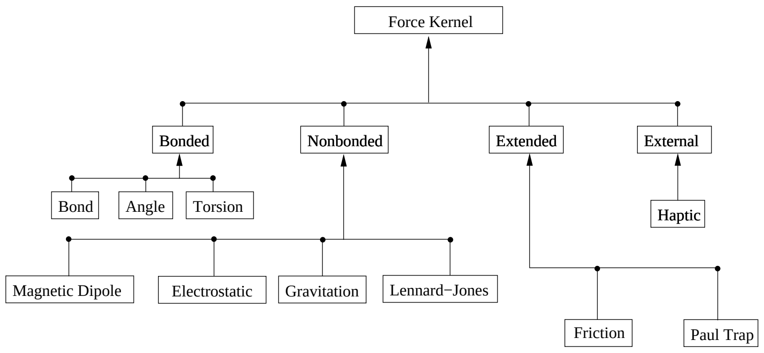


Fig. 9. Hierarchy of force kernels. These functions compute potential energy, its gradient (the force), and optionally a matrix of second derivatives (the Hessian).

(R5) Components to modify the potentials and forces, i.e., switching functions.

The evaluation of non-bonded forces performs the following steps: do a distance testing, check for exclusions, call a non-bonded kernel, apply a switching function (if specified), and then apply the chain rule to the potential energy and forces computed. It is possible to reuse some of the evaluation algorithms (cutoff or all pairs) for almost any pair–wise non-bonded force kernel. This is illustrated in Fig. 8. The forces are parameterized on the boundary condition and the kernel, and they require an algorithm to evaluate them, either bonded or non-bonded. Examples of kernels available are given in Fig. 9: bond, angle, dihedral and improper for bonded interactions; electrostatic, Lennard–Jones, gravitation, and magnetic dipole for non-bonded; and extended forces such as friction and Paul trap attraction, which are helpful in a variety of non biological applications. There is also an external force that is the input from a haptic device, which can be used to steer the MD simulation. For most kernels, methods for computing the corresponding Hessian or matrix of second derivatives are available. This is useful in some advanced numerical integrators and other applications. An example of adding a new kernel is in Section 4.1.1. An example of adding a new non-bonded force computation algorithm called multigrid summation is in Section 4.1.3.

The great advantage of this generic design is maintenance — performance improvement or bug fixes to a module of the generic non-bonded force computation apply to all non-bonded forces automatically. For example, by changing the order of cutoff and exclusion tests to perform the most frequent case first, performance improvement of up to 40% is obtained. To have all non-bonded forces benefit from this optimization, it is sufficient to change the generic method that evaluates one atom–pair interaction.

3.3.1 *Force interface.* Templates and inheritance are combined in the Policy or Strategy pattern [Alexandrescu 2001],[Gamma et al. 1995, pp. 315-323]. This pattern promotes the idea of varying the behavior of a class independent of its context. It is well-suited to break up many behaviors with multiple conditions and it decreases the number of conditional statements. In particular, this pattern is used so that the algorithm to select the n-tuples (R1) – *host class* – is decoupled from the other four requirements (R2–R5) – *policy classes*. This allows the simultaneous evaluation of different force kernels with the same algorithm.

Templates or generic classes enable *parameterization* in ways not supported by regular classes, cf. [Stroustrup 1997; Veldhuizen 1998]. Template parameters can be seen as place holders that the user will fill out when instantiating an object or defining a type. The code of template classes is created at compile time depending on user-defined types or values. For example, in the framework design the boundary conditions are type template parameters (periodic or vacuum) and the non-bonded force definition takes a Boolean parameter indicating whether a switching function is used or not.

To illustrate how the different requirements interact, let us take a close look at how three force types are defined in Program 3. The first uses a MG algorithm (R1) for fast computation of the `Coulomb` force kernel (R4) with C^2-continuous switching function. The second uses a cutoff algorithm (R1) for computation of the `LennardJones` force kernel (another example of R4), which is modified by a C^1-continuous switching function (R5). The last one simultaneously evaluates the Lennard–Jones and Coulomb force kernels. For all forces, the interaction pairs are managed by a cubic cell manager `CM` (R3) and periodic boundary conditions `PBC` (R2). All forces have a common interface, a virtual function called `evaluate(),` which does the evaluation of the force contributions based on its parameterization and policy choices.

```
class NonbondedMultiGridForce<PBC,CM,Hermite,CoulombForce::C2>;
class NonbondedCutoffForce<CM,OneAtomPair<PBC,C1,LennardJones> >;
class NonbondedCutoffForce<CM,TwoAtomPairs<PBC,C1,LennardJones,C2,CoulombForce> >;
```

Program 3. Some examples of non-bonded force declarations

3.3.2 *Force object creation.* The definition of a force maps into a unique identification string. Fig. 10 illustrates the force definition language for PROTOMOL users. A force definition starts always with the keyword `force` followed by the name of a force and *force options*. Examples of these are interpolation schemes, friction coefficients, etc.. Other forces are defined only by one keyword, e.g., angle forces. The keywords **compare** and **time** are for comparison purposes (see Section 3.5).

As Fig. 11 illustrates, a just-in-time compiler is needed that can transform the user definition into a real force object. The requirements of object creation are satisfied by the

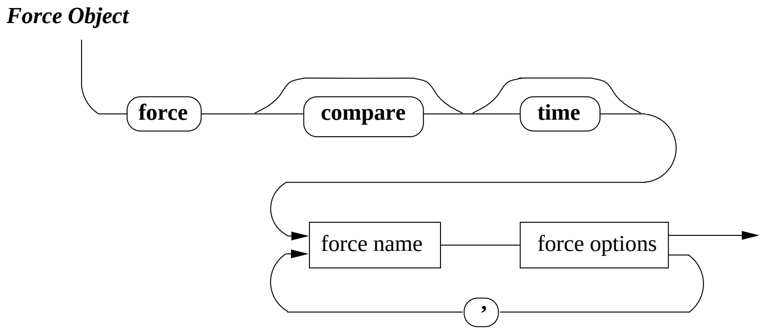


Fig. 10. Syntax chart for PROTOMOL's force definition language.

Abstract Factory [Gamma et al. 1995, pp. 87-95] and the Prototype [Gamma et al. 1995, pp. 117-126] patterns. The Abstract Factory pattern delegates the object creation, and the Prototype pattern allows dynamic configuration. The factory is in charge of converting the user-specified force into an object that has been properly setup to do computation. The factory creates replicas of "prototypes" that have been registered by the developer. This restricts the factory to create only supported objects, since not all combinations of R1-R5 make sense or are supported at a given stage of development.

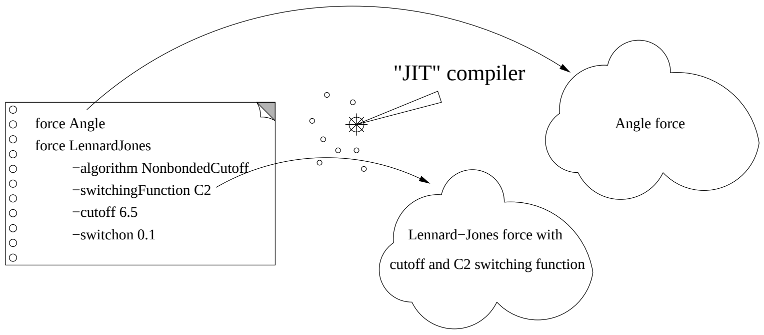


Fig. 11. Correspondence between the input definition and the actual force object.

3.4 Parallelization

PROTOMOL follows an incremental parallelization approach [Matthey and Izaguirre 2001] to encapsulate the parallelization details and provide mechanisms to postpone the development of a parallel implementation. Incremental parallelization relies on common force interfaces. Methods called `evaluate()` and `parallelEvaluate()`, perform the evaluation of the force contributions based on its parameterization and policy choices. The method `evaluate()` is a pure virtual function (required interface for all forces) and expects always a sequential implementation. In contrast, `parallelEvaluate()` is an optional interface for parallel implementation. In case no parallel implementation is provided, it will call the sequential method `evaluate()`. This mechanism allows the developer to postpone a parallel implementation, even in a parallel environment.

The parallelization itself is based on a force decomposition [Plimpton and Hendrickson 1996] with master-slave or with static work distribution. In case of master-slave decomposition, the slaves are assigned on demand a range (work-command) by the master, which represents a collection of terms of the sum of all interactions which make up $U(\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N)$. The splitting of each parallel force is defined and implemented individually, such that each force provides splitting (or distribution) information for the master based on its own characteristics and the actual user input. Each force defines the number of ranges the force is divided in. The work defined by the range should be as big as possible to reduce communication, but small enough to balance out the total work for one time step. The framework assumes that each range of a single force represents computational work of the same order. In some cases, a complete force type is assigned to one slave, i.e., when no parallel implementation is provided. After calculating all interactions (completion of step 2c in Algorithm 1) the force and energy contributions are globally updated by the framework. The implementation is based on MPI with some data replication. The approach reflects to some extent the idea of multi-threading or OpenMP [OpenMP-forum 1997], but MPI allows control of data locality and more coarse-grain parallelism. The approach may not scale well for a very large number of processors. Nevertheless, it has performed well for a moderate number of processors, i.e., up to 32. An example of parallelization of a force kernel is in Section 4.1.2.

We are working on parameterization for the parallelization in order to make the framework more scalable. The goal is to keep parallelism encapsulated so that algorithm developers do not need to consider these details at once. The first parameter is the *load distribution* scheme (e.g., master-slave, static). The second is the *data management* scheme, which allows for data distribution according to a spatial decomposition or some other sorting criterion. The final parameter is the type of *global communication mechanism* to be used, such as point-to-point or broadcast, which can be chosen based on automatic analysis of execution run-times.

3.5 Performance and accuracy monitoring

In order to make it easier to implement and evaluate new algorithms, ProtoMol supports fairly generic performance and accuracy monitoring. To perform experimental analysis of algorithms, two of the most important operational principles are reasonably efficient implementations and comparability [Johnson 2002].

To ensure efficient implementation of all algorithms to be compared, the design and implementation of ProtoMol was done after a thorough domain analysis of whole families of algorithms for MD. The analysis pinpoints commonalities and specific differences among algorithms. The commonalities are exploited through the use of genericity and inheritance, which results in efficient implementations for all algorithms.

To ease the evaluation and comparison of different force algorithms, comparability functionalities were added to the framework. At present, pairs of forces can be compared to determine energy and force errors of new force methods. The comparison is performed on-the-fly, such that the reference force does not affect the current simulation. Program 4 shows an example of use of performance monitoring to compare the fast electrostatics method PME using two grid sizes, such that the more accurate one serves as an accuracy estimator. This is useful to validate and verify a simulation. Another example of use would

```
force compare time force Coulomb
    -algorithm PME -real -reciprocal -correction
    -cutoff 6.5 -gridsize 10 10 10   # Coarser grid used for the MD simulation
force compare time force Coulomb
    -algorithm PME -real -reciprocal -correction
    -cutoff 6.5 -gridsize 20 20 20   # Finer grid used as reference
force time LennardJones -algorithm Cutoff -switchingFunction C1 -cutoff 8.0
```

Program 4. Examples of accuracy and timing comparison.

be to compare two different algorithms, such as PME and Ewald, or MG and all–pair evaluation. The first algorithm is the one being measured and the second is the reference algorithm. Comparisons can be nested to evaluate accuracy and run-time performance simultaneously.

4. EVALUATION OF THE FRAMEWORK

Many MD and N-body algorithms have been implemented in PROTOMOL, which has more than 45,000 lines of code and nearly 200 reusable classes. Optimized versions, including parallel ones, have been relatively easy to incorporate by exploiting the common services available in the framework. An evaluation of the extensibility of PROTOMOL, its performance, and examples of scientific and educational applications, are presented here.

4.1 Extension of the framework

We give a few examples here: (i) addition of a new force kernel; (ii) parallelization of an existing force kernel; (iii) addition of a new non-bonded force algorithm; and (iv) addition of a new MTS integrator.

4.1.1 *Adding a new force kernel.* Coulomb crystal systems are defined by ions with an additional magnetic trap [Tosuji et al. 2002]. A commonly used trap is the Paul Trap attraction, which is given by

$$U^{\mathrm{PaulTrap}}(\vec{r}_i) = \sum_{i=1}^{N} \left(\frac{1}{2} m_i \omega_{xy}^2 (x_i^2 + y_i^2) + \frac{1}{2} m_i \omega_z^2 z_i^2 \right), \qquad \vec{r}_i = (x_i, y_i, z_i)^\top. \tag{8}$$

Here, $\vec{r}_i$ is the coordinate of particle i, and ω_{xy} and ω_z parameterize the Paul Trap attraction.

Program 5 shows the implementation of this force. Lines 1–2 declare the class `PaulTrapSystemForce` as a derived class from `SystemForce`. The class then redefines the method `evaluate` to implement Eq. (8) in lines 5–18. Note that the class is parameterized on the boundary conditions. The boundary condition class computes distances and positions. After obtaining the mass m_i in line 9, the coordinates for the current atom, `pos`, are obtained in line 10. Then the force and energy contributions are calculated (lines 11–15). The potential energy contribution is accumulated in line 17. The function `getParameters` can be overloaded for any force function or integrator. It defines locally what parameters should be read from the configuration files or GUI menus to properly define this force type. For example, `omegaXY` (lines 21, 22) corresponds to ω_{xy} in Eq. (8).

4.1.2 *Parallelizing an existing force kernel.* In order to parallelize a given force kernel, `parallelEvaluate()` has to be overwritten. Furthermore, a splitting of the force computation has to be defined. For example, a velocity dependent friction could be implemented

```
template<class TBoundaryConditions>
class PaulTrapSystemForce : public SystemForce
{
 public: // From class SystemForce
 virtual void evaluate(...){
  const TBoundaryConditions &boundary = ...;
   Real e = 0.0;
   for(int i=0;i<topo->atoms.size();i++){
     Real c = topo->atomTypes[topo->atoms[i].type].mass;
     Coordinates pos(boundary.basisPosition((*positions)[i]));
     Coordinates f(c*myOmegaXY*myOmegaXY*pos.x,c*myOmegaXY*myOmegaXY*pos.y,
                   c*myOmegaZ*myOmegaZ*pos.z);
     (*forces)[i] -= f;
     e += 0.5*c*(myOmegaXY*myOmegaXY*(pos.x*pos.x+pos.y*pos.y)+
                 myOmegaZ*myOmegaZ*pos.z*pos.z);
   }
   energies->otherEnergy += e;
 }
public: // From class Force
  virtual void getParameters(vector<ParameterType>& parameters) const{
   parameters.push_back(ParameterType("-omegaXY",VarValType::REAL,
                                      VarVal(myOmegaXY)));
   parameters.push_back(ParameterType("-omegaZ",VarValType::REAL,
                                      VarVal(myOmegaZ)));
 }
protected: // New methods
 virtual Force* doMake(string&, const vector<VarVal>&) const{
   Real omegaXY = values[0].getReal();
   Real omegaZ = values[1].getReal();
   return new PaulTrapSystemForce(omegaXY,omegaZ);
 }
...
};
```

Program 5. Implementation of a Paul Trap attraction.

```
01  virtual evaluate(...){...} // sequential implementation
02  virtual parallelEvaluate(...){
...
03   int n =  positions->size();
04   int blocks = ...; // number of blocks
05   for(int i = 0;i<blocks;i++){
06      if(topo->parallel->next()){
07        int to = (n*(i+1))/blocks;
08        if(to > n) to = n;
09          int from = (n*i)/blocks;
10          for(int j=from;j<to;j++){
11             (*forces)[j] += (*velocities)[j]*myF;
12        } } } }
```

Program 6. Parallelization of a velocity dependent friction.

as described in Program 6. In this program, `n` is the number of particles, and `blocks` defines the number of blocks, typically the number of processors. Whenever a slave enters `parallelEvaluate()`, `next()` in line 6 will be true if the i^{th} block has not been processed by any other slave.

4.1.3 *Adding a new non-bonded force algorithm.* MG summation has been used to solve the N-body problem by [Sandak 2001; Skeel et al. 2002]. MG scales as $\mathcal{O}(N)$ by imposing a hierarchical separation of spatial scales. The pair–wise interactions are *split* into a local and a smooth part. The local part consists of short-range interactions, which are computed directly. The smooth part represents the slowly varying energy contributions, approximated with fewer terms – a technique known as *coarsening*. MG uses interpolation

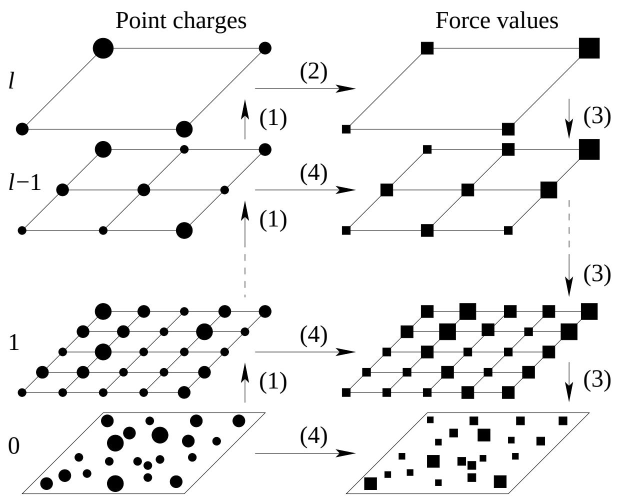


Fig. 12. The multilevel scheme of the MG algorithm. (1) Aggregate to coarser grids; (2) Compute potential energy and force induced by the coarsest grid; (3) Interpolate potential energy and force values from coarser grids; (4) Local corrections.

```
main:
(1) inverse interpolate particle charges to the finest charge grid(1)
(2) call **multiscale**(maxLevel, level 1)
(3) interpolate finest force grid(1) to the particles
(4) correct particle force and potential energy
(5) accumulate forces and total energy

multiscale(maxLevel, level k):
(1) if maxLevel = k then
    (a) compute force values on coarsest grid(maxLevel)
(2) otherwise
    (a) inverse interpolate charge grid(k) to coarser charge grid(k+1)
    (b) call multiscale(maxLevel, k+1)
    (c) interpolate coarser force grid(k+1) to force grid(k)
    (d) correct force grid(k)
```

Algorithm 4. Pseudo-code of a recursive MG scheme with V-cycle.

onto a grid for both the charges and the force to represent its smooth – *coarse* – part. The splitting and coarsening are applied recursively and define a grid hierarchy (Fig. 12). For electrostatics, the *kernel* is defined by $G(r) = r^{-1}$ and $r = ||\vec{y} - \vec{x}||$. $G(r)$ is obviously not bounded for small r. The interpolation imposes smoothness to bound its interpolation error. Splitting functions produce *smoothed kernels* G^k_{smooth} for grid level $k \in \{1, 2, \ldots, l\}$. Corrections to the force and potential energy are required when the modified, smoothed kernel is used instead of the exact one.

The MG scheme can be described as in Algorithm 4, defining a *V-cycle*. It has the same interface as the cutoff or all-pairs algorithm. The order of the interpolation schemes can

```
Hybrid Monte Carlo (HMC)
Given positions $X$ and $\beta := 1/(k_\mathrm{B}T)$, where $k_\mathrm{B}$ is Boltzmann's constant and $T$ is the temperature:
(1) MC Step:
    Generate new random momenta $P$ from a Gaussian distribution
(2) MD Step:
    Given time step $\delta t$ and trajectory length $L$:
    (a) Compute energy $\mathcal{H}_0(X, P)$
    (b) Run MD algorithm for $L/\delta t$ steps
    (c) Compute new energy $\mathcal{H}_1(X', P')$
    (d) Compute change in energy $\delta\mathcal{H} = \mathcal{H}_1 - \mathcal{H}_0$
    (e) Choose a uniform random number, $U$, between $[0, 1]$
    (f) Accept new positions if $U < \exp(-\beta\delta\mathcal{H})$
    (g) If new positions $X'$ are rejected, restore old positions $(X' := X)$.
(3) Sampling Step:
    Compute observable $A(x = X')$.
```

Algorithm 5. One step of Hybrid Monte Carlo (HMC).

be chosen from among several generic interpolation routines, which operate from particles to grid, grid to particles, and from grid to grid. Internally, the algorithm keeps a MG structure that contains the hierarchy of grids. The interpolation from particles to grid and the grid data structures are reused by PME. MG is very competitive, and under periodic boundary conditions is five times faster than PME for systems with 8,000 or more atoms. It has already enabled material science simulations with millions of atoms that would be otherwise intractable, cf. [Matthey 2002, pp. 49 ff.],[Matthey et al. 2003].

4.1.4 *Adding a new MTS integrator.* The Hybrid Monte Carlo (HMC) method consists of short MD trajectories interleaved with a random update of momenta of all particles and a Metropolis Monte Carlo (MC) acceptance criterion. The MC step makes HMC an exact, rigorous method for sampling, whereas MD alone introduces a bias that depends on the finite step size. The HMC method is described in Algorithm 5. The implementation of HMC within PROTOMOL was challenging. The MD step of HMC is applicable to the hierarchy, but how would the MC step apply? Given that HMC requires a STS method, HMC was implemented as a MTS integrator. Fig. 6 shows how `HybridMC` derives from `MTSIntegrator`. The MC step is handled by the `run()` method and a few auxiliary functions. Because the updated positions of an HMC step are conditionally accepted, `HybridMC` requires a unique set of methods to save and restore positions and energies if needed. Aside from these small differences, `HybridMC` is just like any other MTS integrator. Although HMC is not traditionally considered a MTS method, the versatility and robustness of the integrator hierarchy allow HMC to fit right in.

4.2 Performance evaluation

Here we show evidence that the object-oriented and generic design of this scientific application need not sacrifice performance. PROTOMOL was compared against NAMD 2 [Kalé et al. 1999], a leading MD package, because they have similar basic functionalities and both are implemented in C++. Both sequential and parallel performances are compared. In order to make the comparison fair, the same simulation protocol is used. Both configu-

Table I. Sequential comparison of PROTOMOL vs. NAMD 2 for one time step in average on a Pentium III, 1.26 GHz running Linux RedHat.

Test case		# atoms N	PROTOMOL [s]	NAMD 2 [s]	Ratio
Water	vacuum	423	0.01082	0.0 1215	0.89
	periodic		0.01558	0.01488	1.047
BPTI	vacuum	14,281	0.7826	0.889	0.88
	periodic		1.3855	1.216	1.139
ApoA1	vacuum	92,224	5.6221	6.537	0.86
	periodic		9.2586	10.056	0.92

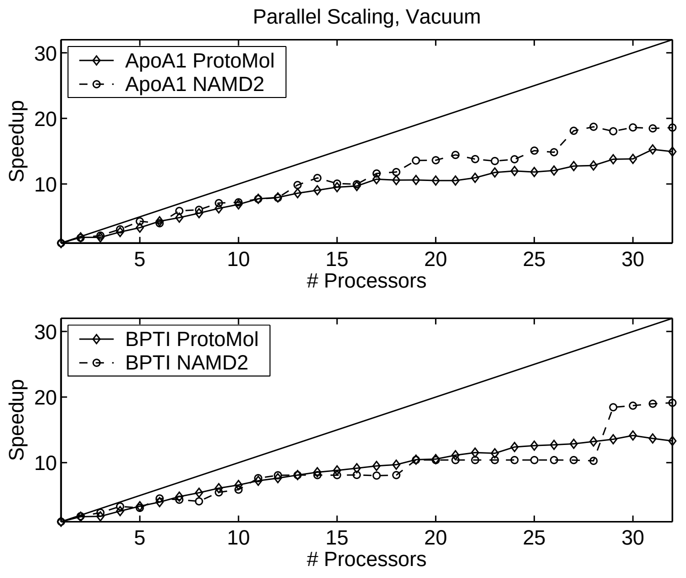


Fig. 13. Parallel scalability for vacuum on an Origin2000. BPTI and ApoA1 are solvated protein systems with 14281 and 92224 atoms, respectively.

rations use leapfrog with time step 1 fs, a non-bonded force cutoff of 10 Å, and Lennard–Jones and Coulomb forces with C^1-continuous and *shift* switching functions, respectively. For the structural and initial simulation state, identical input was used. PROTOMOL was configured with a cell size of 5 Å for the water case and $3\frac{1}{3}$ Å for the two other cases in order to exploit the cache optimally. All runs were performed on a dedicated node of a Linux cluster with 1.26 GHz Pentium III processors. From Table I, one can conclude that PROTOMOL performs well and has a slightly better (sequential) scaling than NAMD 2. Fig. 13 illustrates the parallel scalability of PROTOMOL and NAMD 2. The simulation setup for all test cases was the same as for the sequential comparison, besides a cell size of 5 Å for all PROTOMOL runs. All runs were performed on an Origin2000 with 195MHz R10000 processors. NAMD 2 scales better for large systems with many processors or nodes. PROTOMOL, however, shows smooth scaling.

Fig. 14 show the parallel speedup and scalability of the full electrostatics MG implemented in PROTOMOL applied on Coulomb Crystal systems [Hasse and Avilov 1991;

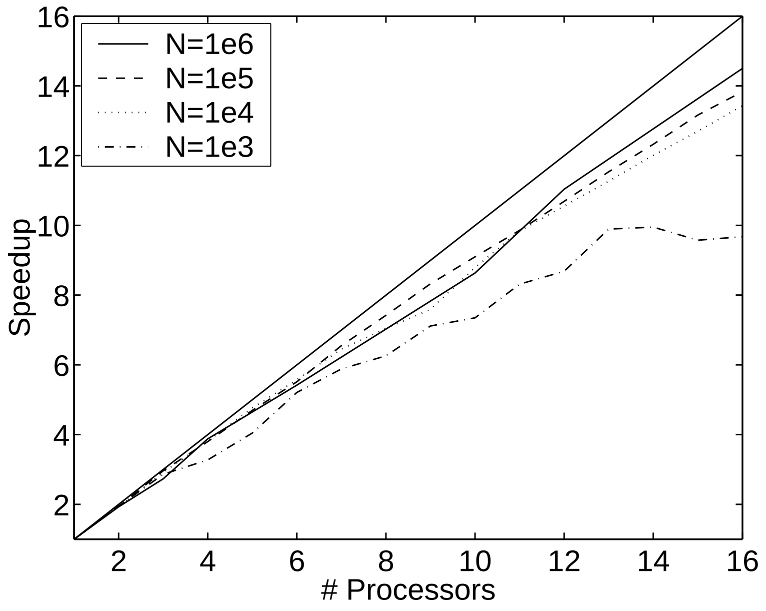


Fig. 14. Parallel speedup of MG electrostatic solver applied on Coulomb Crystal systems with relative force error of order 10^{-5} or less; performed on an IBM p690 Regatta Turbo.

Matthey et al. 2003], which are defined by a computationally dominating electrostatic part and an electric field with linear work complexity. The simulations were performed on an IBM p690 Regatta Turbo. Note that the sequential speedup for $N = 10^6$ is of order 10^2 or more compared to the direct method, and for lower accuracy a speedup of order 10^3 was observed.

4.3 Education and outreach

PROTOMOL is used in several applications, including simulation of ionic crystals, magnetic dipoles, and large biological molecules. It has also been used in courses on scientific computing simulations at Notre Dame and Bergen (for example, CSE 598[3], Computational Biology at University of Notre Dame, USA, and IM 200[4], Ion Crystallization and Dynamics, at University of Bergen, Norway).

5. DISCUSSION

PROTOMOL allows rapid prototyping of fast electrostatics algorithms and integration schemes while achieving high performance. In this section, we discuss how we achieved this.

5.1 High–performance framework and rapid prototyping

The main question addressed in this paper is *what it takes to make a high–performance platform that allows for rapid development of new algorithms,* particularly for MD in-

[3]`http://www.cse.nd.edu/courses/cse598k/www/`
[4]`http://realfag.uib.no/fag/2002v/IM200`

tegrators and N-body solvers. There are excellent examples of highly successful rapid prototyping environments for scientific and engineering software: for example, Mathematica,[5] MATLAB,[6] and FEMLAB.[7] However, due to the interpreted nature of these environments, it is not possible to solve challenging MD applications in them. Nonetheless, these environments are built on top of highly optimized libraries that are typically written in an efficient language such as Fortran or C. A more traditional solution for high performance software is to use directly these libraries, and perhaps write new algorithms into them. In these libraries, such as LAPACK [Anderson et al. 1999] for linear algebra, there are many versions of functions to perform the same task. The interfaces are also very large, because many parameters need to be specified (ten or more are not unusual). If the library supports parallelism, the interface of these functions will grow, and the difficulties of using or extending them are significant, cf. ScaLAPACK [Blackford et al. 1997].

Another way of writing scientific software is suggested here: *encapsulate commonalities* of families of algorithms using inheritance and templates. Inheritance allows for shared behavior and specialization, such as in the integrator and force hierarchies presented here. Templates, or "code skeletons", enable highly efficient code generation at compile time. Parallelism is encapsulated: developers of new algorithms can benefit from existing parallelism, but do not have to worry about it from the beginning. Whenever possible, users are allowed to define their own methods and protocols without having to recompile the code. This is often done with scripting language interfaces, such as in SPASM [Beazley and Lomdahl 1996]; in this work we have preferred *domain specific languages*, because they are easier to learn and can easily be part of a visual environment through a GUI. Another important component is the ability to monitor performance of the algorithms, in terms of accuracy, stability, and time. This facilitates comparison of algorithms and automatic tuning of the parameters for complex algorithms, and greatly enhances usability of the code. All these elements have been integrated into PROTOMOL.

Inheritance and the reuse of design patterns allow for common interfaces and optimizations, both at the algorithmic level (such as was done here for integrators and forces), and at the architectural level (for example, by designing classes to be "make-able" by factories, one can greatly simplify the complex process of object creation and destruction). Maintenance is greatly improved, as well as the modularity of the code. A disadvantage of this approach is the relative speed of C++ compiled code, which can be slower by a factor of two to Fortran code. However, compiler technology for C++ is continually improving. Also, PROTOMOL's implementation can use object-oriented languages that tend to produce more efficient code, such as Eiffel[8]. Another disadvantage is that there is a higher learning curve to start adding code into the framework, because of the need to master its overall design. However, after the initial effort, code development in PROTOMOL is easier than using traditional methods, and more productive in the long run.

Evaluation of the framework has shown that it is simple to add new force kernels, N-body algorithms, and integrators, both using the IDL and by adding classes. Proof of its suitability as a research and development platform are the variety of new methods implemented. These methods provide orders of magnitude speedups over state-of-the-art

[5]`http://documents.worlfram.com/v4/index34.html`
[6]`http://mathworks.com/access/helpdesk/help/techdoc/matlab.shtml`
[7]`http://femlab.com`
[8]`http://www.eiffel.com`

MD and N-body technology. PROTOMOL's sequential performance is excellent, and its parallel performance is adequate for moderate size clusters of commodity computers.

The main remaining challenge is further customization and optimization of parallelism, to both achieve high scalability and respect encapsulation. Spatial decomposition, dynamic load balancing, and more complex task scheduling favor scalability, but a naive implementation of these sophisticated techniques would force developers to be aware of these issues from the start in detriment of the rapid prototyping goals of the framework. The data structures and the algorithms need to be decoupled further. This can be achieved by using more generic algorithms, such as in the C++ standard library. We are implementing XML[9] interfaces for all data transfers, to make PROTOMOL "grid friendly". There are also many integrators and methods for analysis of MD that should be incorporated. Because PROTOMOL is open source, some users have already started contributing methods. We might also wrap methods implemented in other MD programs, see below. Finally, performance monitoring and run-time selection of optimal algorithms and parameters need to be incorporated more thoroughly into PROTOMOL. For example the stand-alone tool MDSIMAID recommends optimal parameters for combinations of fast electrostatic algorithm and STS or MTS integrators for a given simulation using PROTOMOL [Ko 2002].

5.2 Related work in design of scientific software

An MD program designed with modern object oriented software engineering techniques is NAMD 2. It addresses several of the requirements identified in this paper and has been an inspiration in the design of PROTOMOL. NAMD 2 is written in C++ and the parallel language Charm++, and has excellent performance[10] through the use of an efficient data and work decomposition and active messages. However, many optimizations are not encapsulated enough: For example, different numerical integrators are hardwired into one sequence. Multiple tests determine what kind of equations of motion one is using, for example NVE or NVT dynamics, and force algorithm, such as all pairs or cutoff electrostatics. This makes it extremely difficult to add new integrators. The force computation is written using compilation macros, whereas PROTOMOL uses generic classes (templates). These have type checking and are a more reliable way of writing software. Most importantly, in NAMD 2 developers need to deal with active messages, which hide the control flow, and the distributed nature of the data decomposition. PROTOMOL's design hides these optimizations at the deepest level.

Another MD suite of programs written in Fortran 77 as a platform for algorithm development and parameterization is TINKER.[11] It is a series of programs that provide many tools for MD simulation and analysis. PROTOMOL has more integration and N-body algorithms available, whereas TINKER has more force field support, including a polarizable force field for proteins. It is possible to interface modules between this program (and several others) and PROTOMOL.

PROTOMOL was influenced by BALL [Boghossian et al. 1999], a library that abstracts the data structures used in molecular modeling; OOMPAA [Huber and McCammon 1999], which emphasizes the use of generic algorithms; and BOOGA [Streit 1997], an object-oriented framework for visualization. Compared to BALL, PROTOMOL provides algorith-

[9] `http://www.w3.org/XML`
[10] NAMD 2 is the recipient of a 2002 Gordon Bell Award, for scaling to thousands of processors.
[11] http://dasher.wustl.edu/tinker/

mic facilities for integrators and N-body solvers not available there. They have interesting abstractions, such as force field, that would be helpful in PROTOMOL. PROTOMOL would also benefit from generic algorithms in the style of OOMPAA.

ACKNOWLEDGMENTS

Many students have contributed to PROTOMOL. Special thanks to Jeremiah Willcock who first designed the non-bonded force evaluation structure as a final project for a graduate computer science course taught by J. A. Izaguirre. Simon Kanaan carefully proofread this manuscript. A list of all contributors can be found in the PROTOMOL web page: `http://protomol.sourceforge.net`.

REFERENCES

ALEXANDRESCU, A. 2001. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley, Reading, Massachusetts.

ALLEN, M. P. AND TILDESLEY, D. J. 1987. *Computer Simulation of Liquids*. Clarendon Press, Oxford, New York. Reprinted in paperback in 1989 with corrections.

ANDERSEN, H. C. 1983. Rattle: A 'velocity' version of the Shake algorithm for molecular dynamics calculations. *J. Comput. Phys. 52*, 24–34.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, Third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

BEAZLEY, D. M. AND LOMDAHL, P. S. 1996. Lightweight computational steering of very large scale molecular dynamics simulations. In *Proceedings of Supercomputing '96*.

BIESIADECKI, J. J. AND SKEEL, R. D. 1993. Dangers of multiple-time-step methods. *J. Comput. Phys. 109,* 2, 318–328.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

BOGHOSSIAN, N. P., KOHLBACHER, O., AND LENHOF, H.-P. 1999. BALL: Biochemical Algorithms Library. In *3rd Int. Workshop on Algorithm Engineering (WAE-99),* J. S. Vitter and C. D. Zaroliagis, Eds. Lecture Notes in Computer Science, vol. 1668. Springer-Verlag, London, 330–344.

BRANDT, A. AND LUBRECHT, A. A. 1990. Multilevel matrix multiplication and fast solution of integral equations. *J. Comput. Phys. 90*, 348–370.

BROOKS, B. R. AND HODOŠČEK, M. 1992. Parallelization of CHARMM for MIMD machines. *Chemical Design Automation News 7*, 16–22.

BRÜNGER, A., BROOKS, C. B., AND KARPLUS, M. 1982. Stochastic boundary conditions for molecular dynamics simulations of ST2 water. *Chem. Phys. Lett. 105*, 495–500.

BRÜNGER, A. T. 1992. *X-PLOR, Version 3.1: A System for X-ray Crystallography and NMR*. The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University.

DARDEN, T. A., YORK, D. M., AND PEDERSEN, L. G. 1993. Particle mesh Ewald. An $N \cdot \log(N)$ method for Ewald sums in large systems. *J. Chem. Phys. 98*, 10089–10092.

ESSMANN, U., PERERA, L., AND BERKOWITZ, M. L. 1995. A smooth particle mesh Ewald method. *J. Chem. Phys. 103,* 19, 8577–8593.

EWALD, P. 1921. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Ann. Phys. 64*, 253–287.

FINCHAM, D. 1994. Optimisation of the Ewald sum for large systems. *Mol. Sim. 13*, 1–9.

FRENKEL, D. AND SMIT, B. 2002. *Understanding Molecular Simulation, Second Edition*. Academic Press, San Diego.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts.

GRUBMÜLLER, H. 1989. Dynamiksimulation sehr großer Makromoleküle auf einem Parallelrechner. M.S. thesis, Physik-Dept. der Tech. Univ. München, Munich.

HAMPTON, S. AND IZAGUIRRE, J. A. 2004. Improved sampling for biological molecules using Shadow Hybrid Monte Carlo. Accepted in *International Conference on Computational Science* (ICCS 2004), Poland.

HASSE, R. H. AND AVILOV, V. V. 1991. Structure and Mandelung energy of spherical Coulomb crystals. *Phys. Rev. A 44,* 7, 4506–4515.

HUBER, G. A. AND MCCAMMON, J. A. 1999. OOMPAA—Object-oriented model for probing assemblages of atoms. *J. Comput. Phys 151,* 1, 264–282.

ISRALEWITZ, B., GAO, M., AND SCHULTEN, K. 2001. Steered molecular dynamics and mechanical functions of proteins. *Curr. Opinion Struct. Biol. 11*, 224–230.

JOHNSON, D. S. 2002. A theoretician's guide to the experimental analysis of algorithms. In *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges,* M. Goldwasser, D. S. Johnson, and C. C. McGeoch, Eds. American Mathematical Society, Providence, RI, 215–250.

KALÉ, L., SKEEL, R., BHANDARKAR, M., BRUNNER, R., GURSOY, A., KRAWETZ, N., PHILLIPS, J., SHINOZAKI, A., VARADARAJAN, K., AND SCHULTEN, K. 1999. NAMD2: Greater scalability for parallel molecular dynamics. *J. Comput. Phys. 151*, 283–312.

KO, A. 2002. MDSimAid: An automatic recommender for optimization of fast electrostatic algorithms for molecular simulations. M.S. thesis, University of Notre Dame, Notre Dame, Indiana, USA.

LEEUW, S. W. D., PERRAM, J. W., AND SMITH, E. R. 1980. Simulation of electrostatic systems in periodic boundary conditions. I. lattice sums and dielectric constants. *Proc. R. Soc. Lond. A 373*, 27–56.

MA, Q. AND IZAGUIRRE, J. A. 2003a. Long time step molecular dynamics using targeted langevin stabilization. In *Proceedings of the ACM Symposium on Applied Computing, Melbourne, FL*. ACM, New York, 178–182.

MA, Q. AND IZAGUIRRE, J. A. 2003b. Targeted langevin stabilization of molecular dynamics. In *Proc. of the* SIAM Conference on Computational Science and Engineering (CSE'03)*(CD-ROM)*. SIAM, San Diego, CA.

MA, Q. AND IZAGUIRRE, J. A. 2003c. Targeted mollified impulse — a multiscale stochastic integrator for long molecular dynamics simulations. *SIAM Multiscale Model. Simul. 2,* 1, 1–21.

MARTYNA, G. J., KLEIN, M. L., AND TUCKERMAN, M. E. 1992. Nosé-Hoover chains: The canonical ensemble via continuous dynamics. *J. Chem. Phys. 97,* 4, 2635–2643.

MATTHEY, T. 2002. Framework design, parallelization and force computation in molecular dynamics. Ph.D. thesis, University of Bergen, Bergen, Norway.

MATTHEY, T., HANSEN, J. P., AND DREWSEN, M. 2003. Coulomb bi-crystals of species with identical charge-to-mass ratios. *Phys. Rev. Lett. 91,* 16, 165001.

MATTHEY, T. AND IZAGUIRRE, J. A. 2001. ProtoMol: A molecular dynamics framework with incremental parallelization. In *Proc. of the Tenth SIAM Conf. on Parallel Processing for Scientific Computing (PP01)*. Proceedings in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia.

MATTHEY, T., KO, A., AND IZAGUIRRE, J. A. 2003. ProtoMol, an object-oriented framework for algorithmic development. In *Computational Science—ICCS 2003, International Conference, Melbourne, Australia and St. Petersburg, Russia*. Springer-Verlag, Berlin, 50–59. Lecture Notes Comput. Sci. 2659.

NOSÉ, S. 1984. A unified formulation of the constant temperature molecular dynamics methods. *J. Chem. Phys. 81,* 1, 511–519.

OPENMP-FORUM. 1997. OpenMP: A proposed industry standard API for shared memory programming. technical report. `http://www.openmp.org`.

PAGONABARRAGA, I., HAGEN, M. H. J., AND FRENKEL, D. 1998. Self-consistent dissipative particle dynamics algorithm. *Europhys. Lett. 42,* 4, 377–382.

PLIMPTON, S. AND HENDRICKSON, B. 1996. A new parallel method for molecular dynamics simulation of macromolecular systems. *J. Comp. Chem. 17,* 3, 326.

RYCKAERT, J., CICCOTTI, G., AND BERENDSEN, H. 1977. Numerical integration of the Cartesian equation of motion of a system with constraints: molecular dynamics of n-alkanes. *J. Chem. Phys. 23*, 327–341.

SANDAK, B. 2001. Multiscale fast summation of long-range charge and dipolar interactions. *J. Comp. Chem. 22,* 7, 717–731.

SKEEL, R. D. 1999. Integration schemes for molecular dynamics and related applications. In *The Graduate Student's Guide to Numerical Analysis,* M. Ainsworth, J. Levesley, and M. Marletta, Eds. Springer Ser. Comput. Math., vol. 26. Springer-Verlag, Berlin, 119–176.

SKEEL, R. D. AND IZAGUIRRE, J. A. 2002. An impulse integrator for Langevin dynamics. *Mol. Phys. 100,* 24, 3885–3891.

SKEEL, R. D., TEZCAN, I., AND HARDY, D. J. 2002. Multiple grid methods for classical molecular dynamics. *J. Comp. Chem. 23,* 6, 673–684.

STREIT, C. 1997. BOOGA, ein komponentenframework für grafikanwendungen. Ph.D. thesis, Institute of Computer Science and Applied Mathematics, University of Berne.

STROUSTRUP, B. 1997. *The C++ Programming Language*, Third ed. Addison-Wesley, Reading, Massachusetts.

TOSUJI, H., KISHIMOTO, T., TOTSUJI, C., AND TSURUTA, K. 2002. Competition between two forms of ordering in finite Coulomb clusters. *Phys. Rev. Lett. 88,* 12.

TUCKERMAN, M., BERNE, B. J., AND MARTYNA, G. J. 1992. Reversible multiple time scale molecular dynamics. *J. Chem. Phys. 97*, 3, 1990–2001.

TUCKERMAN, M. E., YARNE, D., SAMUELSON, S. O., HUGHES, A. L., AND MARTYNA, G. J. 2000. Exploiting multiple levels of parallelism in molecular dynamics based calculations via modern techniques and software paradigms on distributed memory computers. *Comput. Phys. Commun. 128,* 333–376.

VELDHUIZEN, T. 1998. Blitz++: The library that thinks it is a compiler. Conference presentation, Extreme! Computing Laboratory, Indiana University Computer Science Department. Sept.

VERLET, L. 1967. Computer 'experiments' on condensed fluids I. Thermodynamical properties of Lennard–Jones molecules. *Phys. Rev. 159,* 98–103.

VINCENT, J. AND MERZ, K. M. 1995. A highly portable parallel implementation of AMBER using the Message Passing Interface standard. *J. Comp. Chem. 11*, 1420–1427.

Received MM YY; MM YY; accepted MM YY