Flexibility and Interoperability in a Parallel Biomolecular Dynamics Code*

Robert Brunner† Laxmikant Kalé‡ James Phillips§

Abstract

In simulations of biomolecular systems performance is critical, and parallel computing is an essential enabling technology. One million force-field evaluations are required to obtain a single nanosecond of simulated time. For large systems this may require a month or more of run time. In addition, the state of the art progresses at a steady pace, producing new and better integrators, long-range electrostatic methods, and simulation protocols. Experimenting with alternate novel strategies, and reusing new libraries with faster algorithms without having to implement an entirely new program present additional challenges. We have addressed these challenges in the program NAMD2, creating a software system which obtains excellent parallel speedups and serial performance on an increasing variety of distributed memory platforms while being flexible enough to incorporate new algorithms and externally developed packages. Owing to the Converse parallel runtime system, NAMD2 can even interoperate at the library level with parallel packages written in a variety of parallel languages and paradigms.

Applications in science and engineering are growing in complexity. It is clear that the vast amount of computational power available in near future will allow for much more accurate modeling of natural phenomena and engineering designs. In fact, it can be argued that our ability to develop software lags behind the hardware capabilities. How will the science and engineering applications of future be implemented? What technical issues must be resolved to facilitate increased productivity in implementations of such applications? How can algorithmic and hardware advances be incorporated rapidly in applications? We suggest some directions that may comprise solutions to these issues, and provide supporting data based on our recent experiences in implementing a parallel molecular dynamics program.

For increased productivity, it is essential that software components be reused to the extent possible. This is especially difficult for parallel software components, because of the need for inter-component interfaces to be parallel. We believe that an object-oriented design, especially one based on data driven parallel objects, provides the necessary infrastructure and methodology to achieve reuse of parallel software components. To provide the ability to incorporate new algorithms and extensions quickly, we advocate a flexible open-ended design, based on indirect "registration and call-back" mechanisms.

---

*This work was supported by the National Institutes of Health (NIH PHS 5 P41 RR05969-04) and the National Science Foundation (NSF/GCAG BIR 93-18159 and NSF BIR 94-23827 EQ).

†Department of Electrical and Computer Engineering and Beckman Institute, University of Illinois, Urbana, IL.

‡Department of Computer Science and Beckman Institute, University of Illinois, Urbana, IL.

§Department of Physics and Beckman Institute, University of Illinois, Urbana, IL. Supported by a Computational Science Graduate Fellowship from the United States Department of Energy.

Further, using user level threads, objects and the principle of information-hiding, the details of parallel programming can be separated from algorithmic aspects.

Parallel programs can be written using several different programming models: message passing, parallel loops, data-driven objects, multithreading, and any of the several experimental programming models. Each model may be well-suited for specific programming tasks. To achieve true reuse in this context, it is necessary to be able to integrate modules written using different models in a single application program. To do this without losing efficiency, the modules must be able to interleave their execution with each other, which creates conflicting demands on processor scheduling. We have developed a run-time framework named Converse that supports such multi-paradigm interoperability. Converse also provides portability, so languages implemented using Converse, and consequently applications written using Converse, run portably on a wide variety of parallel machines without loss of efficiency.

The following sections illustrate how the above ideas were used in the design of the molecular dynamics program NAMD2, which is a production-quality application being used in many scientific studies.

1 Parallel Molecular Dynamics

Molecular dynamics (MD) simulation is an important tool for studying the behavior of biochemical processes at scales too small for direct observation. The simulations reveal the behavior of an interesting molecular process by integrating the equations of motion of each atom over time. Simulations typically include between a few thousand and one hundred thousand atoms. Each atom exerts forces on its neighbors, due to several varieties of chemical bonds (*bond forces*), and Van der Waals' and Coulomb interactions (*nonbonded forces*). The properties of the force field require time steps of the order of one femtosecond, and since the simulations may run for a nanosecond or more, each simulation may require one million iterations.

Although MD simulations may require thousands of hours of computation, each integration step requires only seconds to compute, so parallel simulation demands efficient distribution of this small block of work among processors. The most expensive part of the computation is the calculation of the forces between atoms. Atoms are usually bonded to a small number of nearby atoms, so the bond force calculation for a system of N atoms is $O(N)$. Nonbonded forces involve all pairs of atoms, so exact computation of the non-bonded forces is $O(N^2)$. Fortunately, the non-bonded forces decay rapidly with increasing separation, so many simulations ignore atoms pairs separated by more than some selected *cutoff radius*, reducing the complexity of non-bonded computation to $O(N)$. When the long-range forces are not negligible, a simulation can take advantage of the slow variation of forces due to distant atoms to perform the full-electrostatic calculation less frequently, using a method called *multiple timestepping* [4].

The Theoretical Biophysics group has been working on production-quality molecular dynamics programs since 1994. Unlike previous parallel MD programs, which were simply parallelized versions of sequential programs, NAMD[1] was conceived from the outset to run efficiently in parallel. It was also designed to be modifiable by experienced users other than the developers. The first version of the program, NAMD 1[13], was a PVM-based parallel program which ran on networks of workstations and several supercomputers. NAMD 1 was implemented in C++, but used few object-oriented features of the language. Several

[1] NAMD is an acronym for Not (just) Another MD program.

researchers ran successful simulations with the program [12]. NAMD 1 proved efficient at utilizing small parallel machines (i.e. fewer than 16 processors), affirming our belief that C++ could produce fast-running scientific code. Unfortunately, NAMD 1 did not scale well to larger machines, for the simulation sizes of interest. Also, although NAMD 1 was well documented, and easily modifiable in some ways (such as adding new types of forces), modifications requiring interprocessor communication proved complicated. Development of NAMD2 [9, 15, 8] started in mid-1996. The main goals were to improve the parallel performance, and increase modifiability. It retained most of the I/O routines of NAMD 1, but replaced the parallel simulation core with a new, object-oriented design, implemented on top of the Converse runtime system [10]. The beta version of NAMD2 is now available to the public (http://www.ks.uiuc.edu/Research/namd/), and several local users have already performed simulations using NAMD2 [16].

2 NAMD2 Design Overview

Parallel MD codes typically use one of two schemes to decompose the computation. One technique, force decomposition, takes a list of atom pairs between which forces arise and distributes the list among processors. The advantage of this scheme, besides being straightforward to implement, is that since each force computation requires approximately equal computation, good load balance is easily achieved by just dividing the pairlist evenly. The disadvantage is that, since a pair may get sent to any processor, after integration the updated positions need to be sent to all the other processors. This limits the parallel efficiency of the method, although it can perform well for as many as a few dozen processors. The other popular scheme, which we selected for NAMD 1, is spatial decomposition. The simulation space is divided into regions and all atoms in a region are distributed to one owning processor. Since forces result from atoms that are close together, a processor can compute the forces on its atoms with only the positions of atoms in the adjacent regions. One disadvantage of spatial decomposition is that the density of atoms in regions varies across the simulation space, and also changes during the course of the simulation, complicating load balancing. In NAMD 1, space is divided into cubic regions called *patches*. The patch size is selected to be slightly larger than the cutoff radius, so that, for any atom in a particular patch, all atoms within its cutoff radius lie within the same patch or one of the 26 neighboring patches. This distance is also sufficient so all bonded atoms lie within neighboring patches, insuring that the number of messages is independent of the number of processors.

NAMD2 uses a hybrid of the force and spatial decomposition methods. User experience with NAMD 1 demonstrated a problem with our implementation of spatial decomposition. Typical simulations would result in a few hundred patches of widely varying sizes, which could not be efficiently distributed over more than a few dozen processors. One solution is to use smaller patches, but smaller patches result in increased communication which hurts performance on smaller machines. Instead, NAMD2 keeps the same patch size as NAMD 1, but treats the forces between atoms in each pair of adjacent patches as a separate piece of work which may be run on any processor. Therefore, NAMD has 13 times more pieces of work (for a particular patch, half of the 26 neighbor patches), resulting in better performance on large parallel machines.

Patch	HomePatch	
	ProxyPatch	
Compute	ComputePatch	ComputeNonbondedSelf ‡
	ComputePatchPair	ComputeNonbondedPair ‡
	ComputeHomeTuples †	ComputeBonds
		ComputeAngles
		ComputeDihedrals
		ComputeImpropers
		ComputeNonbondedExcl ‡
	ComputeHomePatches	ComputeDPMTA
		ComputeDPME

†Template base class.
‡Secondary inheritance from ComputeNonbondedUtil.

FIG. 1. *The Patch and Compute class hierarchies of NAMD2.*

3 Object Oriented Design for Flexibility

In NAMD2, every piece of work involved in the force computation is encapsulated by a *compute object* which is responsible for a particular type of interaction between atoms in a fixed set of patches. Compute object execution is dynamically scheduled as patch data becomes available. Both patches and compute objects may be distributed among nodes to achieve load balance and minimize communication. *Proxies* of patches are created on processors where the compute objects need the data from a remote patch. Thus, someone developing a new type of force object needs to focus only on what the computation object does, and treats the proxies identically to locally available data. The class hierarchies responsible for force evaluation in NAMD2 are illustrated in Fig. 1 and described in detail below. The object-oriented design of NAMD2 has allowed several extensions, such as computational steering and long-range electrostatics, to be implemented by simply adding a few derived classes from appropriate existing base classes.

3.1 Patch and Compute Abstract Base Classes

All interactions between patches and compute objects are implemented in the Patch and Compute abstract base classes. On instantiation, each Compute is given a list of patches from which it requires coordinates in order to perform a part of the force calculation. The Compute then finds and registers itself with each required Patch, setting up the following callback mechanism. During each timestep, the Patch notifies all Computes registered with it when the necessary coordinates for force evaluation are available. When a Compute has been notified by all of the patches it depends on, the Compute adds itself to a prioritized work queue, making use of a virtual member function to determine its priority. After waiting in the work queue, another virtual member function of Compute is called to carry out the force calculation. The Compute returns forces to each Patch on which it depends, triggering the Patch to move on to the next timestep when forces have been returned by each dependedent Compute.

3.2 Home and Proxy Patches

Two classes are derived from Patch in order to provide transparent off-node communication of coordinates and forces. A single instance of HomePatch exists for every patch in the spatial decomposition; these are distributed among the nodes of the parallel machine. HomePatch is responsible for integrating atomic coordinates and velocities between force evaluations and migrating atoms between patches. In order that a Compute requiring a particular patch can exist on a different node, HomePatch also sends coordinates to and receives forces from instances of ProxyPatch which may exist on any other node in the machine. The ProxyPatch algorithm is very simple, alternating in state between waiting to receive coordinates from the HomePatch and waiting to return forces to the HomePatch after the dependent Compute objects on the same node have run.

3.3 Second Level Compute Abstract Base Classes

Between the Compute base class and the leaf classes which actually perform the force evaluation lies a layer of classes which implement common patterns of Patch dependency. These classes include ComputePatch, which depends on a single patch, and ComputePatchPair, which depends on two patches. More complicated examples are ComputeHomePatches, which combines data from every HomePatch on the node, and ComputeHomeTuples, which is specialized for dealing with forces calculated from small fixed sets of nearby atoms. The primary benefit of this second layer of abstraction, in addition to a reduction in duplicate code, is that new force field terms can be added to NAMD2 by simply identifying the appropriate dependency pattern, deriving from the corresponding base class, and adding the new class to the initialization routines; no knowledge of the underlying parallel structure is needed. This has allowed researchers with limited knowledge of the NAMD2 design to implement steering forces and boundary potentials.

3.4 Compute Leaf Classes

With communication hidden within Patch and depedency patterns implemented in the Compute base classes, the final implementation of the actual force calculation becomes straightforward, resembling that of a serial program. ComputeHomeTuples is actually a template, isolating the nearly identical implementation of the various classes needed for bonded forces (ComputeBonds, ComputeAngles, ComputeDihedrals, and ComputeImpropers) as well as efficient implementation of exclusion of the nonbonded forces between pairs of atoms (ComputeNonbondedExcl). The remaining nonbonded force calculations, normally the bulk of the required calculation, are divided into intra-patch interactions (ComputeNonbondedSelf) and inter-patch interactions (ComputeNonbondedPair). These forces are typically truncated beyond a certain radius such that interactions only need to be calculated between atoms in neighboring patches. If long-range forces must be calculated as well, more efficient algorithms are used (ComputeDPMTA [17] or ComputeDPME [3]) which are implemented in a one-call-per-procesor manner by deriving from ComputeHomePatches.

3.5 Multiple Inheritance of Nonbonded Functionality

The large amount of work associated with nonbonded calculations makes the efficiency of this inner loop critical for overall performance. This challenge is greater due to the variety of runtime options associated with the implementation of and continuing experimentation with multiple timestepping in NAMD2. The normal $1/r$ electrostatic potential is smoothly truncated with one of several functional forms and during some timesteps subtracted from a non-truncated potential calculated elsewhere. The efficient exclusion of nonbonded interactions between bonded atoms also adds to the complexity, since it is generally more efficient to subtract off a correction between all pairs of excluded atoms except when this would introduce roundoff errors.

It is useful in this situation for the three classes responsible for nonbonded forces (ComputeNonbondedSelf, ComputeNonbondedPair, and ComputeNonbondedExcl) to share a common implementation of the force calculation routine. This has been achieved through multiple inheritance from the ComputeNonbondedUtil class. In addition to all necessary parameters, the C preprocessor has been used to generate from a single source file a set of force evaluation functions specialized for every possible combination of options. This approach reduces the number of branches and function calls in the inner loop and allows the compiler to produce code optimized for each situation. In addition, the majority of the code, being common to all situations, exists in a single location for easy tuning and modification. The one drawback of this approach is the limited expressivity of the C preprocessor; in the future the same result may be achieved more elegantly with templates.

4 Parallel Interoperability through the Converse Parallel Runtime

Parallel software components are significantly harder to reuse than sequential ones. The traditional method for importing a parallel library is to make sure that it is invoked at the same time on all the processors. Then control is fully transferred to the library until it finishes its task. Although this method makes reuse possible, it is not an efficient way of combining library modules in many situations. For example, two different modules that a program needs to invoke may be mutually independent. In this case, it is theoretically possible to overlap the idle times in one module (arising either because of load balance or critical paths or just communication latencies) with useful computations from the other module. However, the traditional library invocation method does not allow for this possibility, at least not easily. (It is possible to use wild-card receives, and other mechanisms to alleviate this problem and regain efficiency; however such techniques sacrifice clarity and modularity. See [6] for a detailed discussion of this issue).

In the Department of Computer Science at the University of Illinois, we have developed techniques based on data-driven objects over the past decade. The Charm system [11], and its predecessor called the Chare Kernel support multiple objects per processor with asynchronous method invocation. The method invocations are sent as messages to the destination processors, where they are placed in a scheduler queue. The scheduler invokes methods from this queue, completing one before moving on to be next. This data-driven execution allows concurrent modules to interleave the execution: if an object from one module is waiting for some data, another object from a different module, for which a message (i.e. remote method invocation) has already been received can continue execution.

User level threads also provide this capability of interleaving execution, and so do several other novel parallel languages/libraries such as Cid, Cilk, Nexus, CC++, etc. [14, 1, 5, 2]. The importance and utility of such new paradigms can be expected to increase as parallel programming becomes ubiquitous, and is applied to many irregularly-structured problems which are not amenable to efficient solution via traditional paradigms.

The question then may seem to be: which one of these new languages should one use for future application. However, these new languages often offer complimentary, non-overlapping, advantages. It may be better to use different languages (parallel programming paradigms) for different modules, so as to provide a better match to the algorithm implemented by the module. Subjective tastes will also dictate choice of languages for individual modules. Also, it is not clear which one of these languages, if any, will succeed and be popular enough to be well supported.

A solution to these problems is to support concurrent interoperability among such languages. The Converse runtime framework supports such interoperability. With Converse [10], one can integrate modules written using threads, data driven objects etc., and allow them to interleave their execution. Converse accomplishes this by using a single general-purpose scheduler to schedule multi-lingual entities. Unblocked user-level threads, messages for data-driven objects, and other language specific entities are encoded as "generalized messages" with a common partial structure, and entered into the scheduler queue. Each "message" carries with it a reference to a handler function, which the scheduler invokes. The handler function is language specific, and may do such things as resuming a user-level thread or invoking a method on an object.

In addition to supporting interoperability, Converse is intended to support quick development of portable runtime libraries for new languages. For this purpose, Converse provides several commonly needed runtime components. Using these, for example, a threaded message passing library that supports nx-like message passing calls from user-level threads was put together in an afternoon with only about 100 lines of code. Converse currently supports over a dozen mini-languages and libraries.

5 Reuse of Parallel Libraries

Although many simulations do not require full-range electrostatics, some simulations do not exhibit realistic behavior unless time-consuming full-range electrostatic forces are computed. Calculating full electrostatics efficiently is a significant programming task. We decided that rather than implement our own scheme, we would use the DPMTA library from Duke University [17]. DPMTA is an efficient parallel implementation of a full electrostatics algorithm, written in C and employing PVM for communications. The multilingual capabilities of Converse, especially its PVM communication component, allow us to use the unmodified DPMTA library without native PVM. We wrote a ComputeDPMTA object, which is simply an object-oriented wrapper object to collect all the atom positions, pass them all at once to DPMTA, and return the resulting forces to the proper patches when DPMTA finishes. While DPMTA is running, it receives its PVM messages normally. Meanwhile, Converse buffers any outstanding messages destined for other modules, until DPMTA completes its calculation, at which point the other modules resume computation. Although Converse could allow DPMTA to overlap its execution with message-driven code, this would require running DPMTA in a separate thread. Since DPMTA consumes a large amount of memory, we opted to run it in the main thread and forego the potential benefits of overlapped execution.

6 Sequential Expression of Algorithms

The outer-loop molecular dynamics algorithm is normally expressed in an iterative, sequential manner: evaluate force-field, update positions and velocities, and repeat. Since one of the goals of NAMD2 is to allow users to easily modify the program, it is essential that this loop be expressed in a serial manner. This is in conflict, however, with the desire to operate in an adaptive and asynchronous manner to achieve maximum parallel efficiency.

Fortunately, typical integrators are *locally sequential* in the sense that the motion of each atom is determined only by the forces calculated for that atom. Hence, the atoms residing on each patch can be integrated independently once all force for a given timestep have arrived. For a simple algorithm this may be implemented as a function which is called between force evaluations. However, startup and shutdown of the algorithm must be expressed as special cases and the code must be read with the understanding that integrator execution will alternate with force field evaluation via some hidden mechanism.

However, this solution becomes less satisfactory when global elements are introduced into the integrator algorithm. For example, maintaining a constant temperature during the simulation requires that a rescaling factor be applied to all atomic velocities at regular intervals. Since this factor varies with the temperature of the system, the kinetic energy of the entire system must be available on all patches before integration may proceed. As the number of optional data required for integration to proceed increases, the source code becomes decreasingly legible and bears little resemblance to the serial formulation of the algorithm. Furthermore, additional functions may be needed for the global aspects of the algorithm.

The problem has been addressed in NAMD2 by taking advantage of the availability of threads in the Converse system. A sequencer thread is associated with every HomePatch and in this thread a single integration function runs from initialization to termination. Global calculations and output are handled by a single controller thread running a complementary algorithm on one node. Although two functions are necessary in this approach, the serial nature of the integration algorithm is immediately evident.

Communication between the local sequencers and global controllers occurs transparently though reduction and broadcast subsystems. When the controller reaches a point where a global quantity, such as temperature, is required for either output or the integration algorithm, a call is made to the reduction system requesting the data. If the data is unavailable the controller thread is suspended other processing continues until the data arrives. Similarly, the sequencer threads may be suspended pending the arrival of broadcasts from the controller.

Although this thread-based approach represents a significant improvement in modifiability, there are two drawbacks to the use of threads in this manner. First, additional memory must be allocated to provide stack space for each thread. Although wasteful, this additional memory usage is spread across all processors in the system so its impact is reduced by running in parallel. (This memory usage can be reduced by using the Structured Dagger notation [7].) Second, it is somewhat confusing that the integration algorithm is divided between the controller and sequencer functions. Additional work is needed to eliminate these shortcomings. One interesting direction for experimentation being the use of a scripting language which can be parsed at run-time to generate both a partitioning of the algorithm into local and global components with automatic handling of interdependencies between them.

7 Performance Impact of New Methods

Our experience with NAMD 1 and NAMD2 demonstrates that programmers can implement efficient computational algorithms in C++, if a few precautions are observed. In MD simulation, as in most scientific codes, a small fraction of the code is responsible for the majority of the computation time, so only these sections require careful scrutiny. Most of the program can use the full expressiveness with C++, with efficiency only of secondary concern. As with any computer language, function calls are an overhead to be avoided in inner loops of code. The C++ `inline` keyword would suggest a way around this

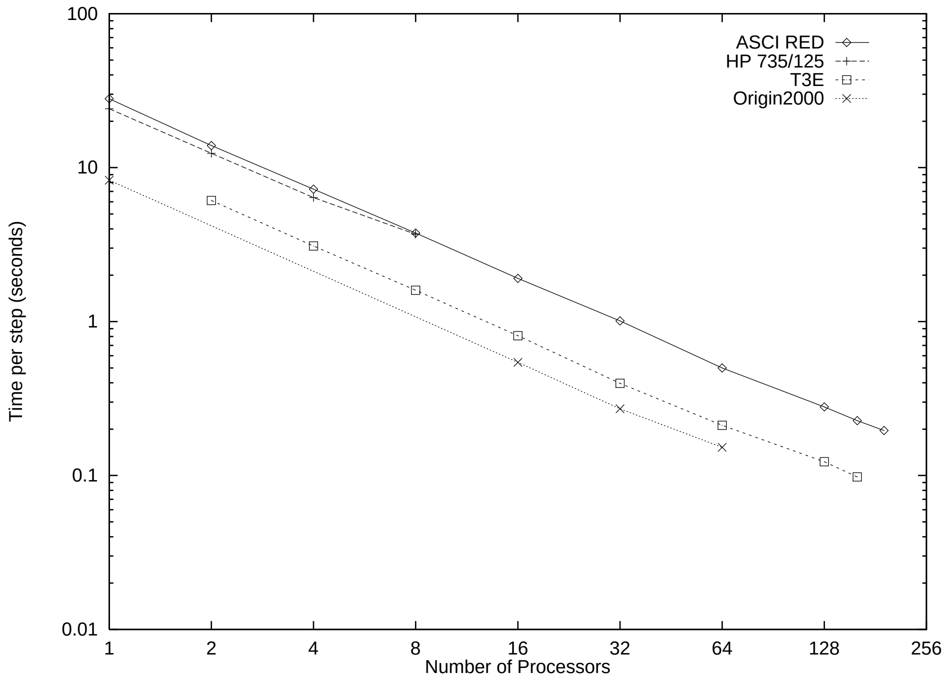

FIG. 2. *Execution time (seconds) per timestep for ER-ERE (36,573 atoms, 12Å cutoff) on several parallel machines.*

problem, but our experience suggests that support for `inline` is compiler-dependent. Some compilers will not inline functions which call other inlined functions. Also, some compilers will not inline functions containing complex control structures or multiple return points. Usually, compilers do not alert the programmer when the `inline` keyword is ignored. In addition to function-call overhead, C++ introduces the overhead of virtual function calls. Virtual functions require an additional pointer lookup to determine the function address. Fortunately, the overhead is comparable to that imposed by a normal function call, and since we avoid function calls in time-critical parts of the code, virtual function calls are not a computational burden. We believe that current C++ compilers produce code which is nearly as efficient as FORTRAN for mathematical operations on simple data types such as scalars and arrays, and the richness of C++ allows us to implement more complex data structures whose algorithmic benefits outweigh the low-level optimization benefits afforded by FORTRAN compilers. The chief disadvantage of C++, particularly when employing templates, is the increase in compilation time. This does not affect the user, but does slightly impede the program development process.

The benefits of the object-oriented design, made possible because of C++ and Converse/Charm++, are demonstrated in Fig. 2. The sequential performance of NAMD2 compares favorably with other MD simulations, but parallel efficiency is where the biggest benefits are achieved. An object-based design allow us to divide the work into many pieces, which better handle communication latency, and giving each processor equal load is easier when the work is finely divided. We have demonstrated speedups of 180 on 224 processors, for a 92,000 atom simulation running on the ASCI Red computer.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, in Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, Santa Barbara, California, July 1995, pp. 207–216. MIT.

[2] K. Chandy and C. Kesselman, *CC++: A Declarative Concurrent Object-oriented Programming Notation*, in Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner, and A. Yonezawa, eds., MIT Press, 1993, pp. 281–313. ISBN 0-272-01139-5.

[3] T. Darden, D. York, and L. Pedersen, *Particle mesh Ewald. an N·log(N) method for ewald sums in large systems*, J. Chem. Phys., 98 (1993), pp. 10089–10092.

[4] T. Forester and W. Smith, *On multiple time-step algorithms and the Ewald sum*, Mol. Sim., 13 (1994), pp. 195–204.

[5] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, *Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems*, Tech. Rep. ANL/MCS-TM-189, Argonne National Laboratory, May 1994.

[6] A. Gursoy, *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*, PhD thesis, University of Illinois at Urbana-Champaign, June 1994. Also, Technical Report UIUCDCS-R-94-1852.

[7] L. V. Kalé and M. Bhandarkar, *Structured Dagger: A Coordination Language for Message-Driven Programming*, in Proceedings of Second International Euro-Par Conference, vol. 1123-1124 of Lecture Notes in Computer Science, September 1996, pp. 646–653.

[8] L. V. Kale, M. Bhandarkar, and R. Brunner, *Load balancing in parallel molecular dynamics*, in Fifth International Symposium on Solving Irregularly Structured Problems in Parallel, August 1998.

[9] L. V. Kalé, M. Bhandarkar, R. Brunner, N. Krawetz, J. Phillips, and A. Shinozaki, *NAMD: A case study in multilingual parallel programming*, in Proc. 10th International Workshop on Languages and Compilers for Parallel Computing, Minneapolis, Minnesota, August 1997.

[10] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, *Converse: An Interoperable Framework for Parallel Programming*, in Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii, April 1996, pp. 212–217.

[11] L. V. Kalé and S. Krishnan, *Charm++: Parallel Programming with Message-Driven Objects*, in Parallel Programming using C++, G. V. Wilson and P. Lu, eds., MIT Press, 1996, pp. 175–213.

[12] D. Kosztin, T. C. Bishop, and K. Schulten, *Binding of the estrogen receptor to DNA: The role of waters*, Biophys. J., 73 (1997), pp. 557–570.

[13] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kalé, R. D. Skeel, and K. Schulten, *NAMD – A parallel, object-oriented molecular dynamics program*, J. Supercomputing App., 10 (1996), pp. 251–268.

[14] R. S. Nikhil, *Parallel Symbolic Computing in Cid*, in Parallel Symbolic Languages and Systems, 1995.

[15] J. C. Phillips, R. Brunner, A. Shinozaki, M. Bhandarkar, N. Krawetz, L. Kalé, R. D. Skeel, and K. Schulten, *Avoiding algorithmic obfuscation in a message-driven parallel MD code*, in Algorithms for Macromolecular Modelling, P. Deuflhard, J. Hermans, B. Leimkuhler, A. Mark, R. D. Skeel, and S. Reich, eds., Lecture Notes in Computational Science and Engineering, Springer-Verlag. In press.

[16] J. C. Phillips, W. Wriggers, Z. Li, A. Jonas, and K. Schulten, *Predicting the structure of apolipoprotein A-I in reconstituted high density lipoprotein disks*, Biophys. J., 73 (1997), pp. 2337–2346.

[17] W. Rankin and J. Board, *A portable distributed implementation of the parallel multipole tree algorithm*, IEEE Symposium on High Performance Distributed Computing, (1995). [Duke University Technical Report 95-002].