Avoiding Algorithmic Obfuscation in a Message-Driven Parallel MD Code

James C. Phillips, Robert Brunner, Aritomo Shinozaki, Milind Bhandarkar, Neal Krawetz, Attila Gursoy, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten

Theoretical Biophysics Group, University of Illinois and Beckman Institute,
405 North Mathews Avenue, Urbana, IL 61801, USA

Abstract. Parallel molecular dynamics programs employing shared memory or replicated data architectures encounter problems scaling to large numbers of processors. Spatial decomposition schemes offer better performance in theory, but often suffer from complexity of implementation and difficulty in load balancing. In the program NAMD 2, we have addressed these issues with a hybrid decomposition scheme in which atoms are distributed among processors in regularly sized patches while the work involved in computing interactions between patches is decomposed into independently assignable compute objects. When needed, patches are represented on remote processors by proxies. The execution of compute objects takes place in a prioritized message-driven manner, allowing maximum overlap of work and communication without significant programmer effort. In order to avoid obfuscation of the simulation algorithm by the parallel framework, the algorithm associated with a patch is encapsulated by a single function executing in a separate thread. Output and calculations requiring globally reduced quantities are similarly isolated in a single thread executing on the master node. This combination of features allows us to make efficient use of large parallel machines and clusters of multiprocessor workstations while presenting minimal barriers to method development and implementation.

Introduction

This paper describes the design history of the program NAMD, developed by members of the Theoretical Biophysics Group at the University of Illinois starting in 1994. The intent is to give the reader a better understanding of the conflicting forces which shape the design of a parallel molecular dynamics code and to demonstrate the need for advanced features such as multiple threads and message-driven execution.

From a software design perspective, a molecular dynamics program carries out a very simple algorithm. The gradient of a potential energy function is calculated for all atoms in a system, yielding a force; this force is then employed by an integration algorithm to update the positions of the atoms for the next force evaluation. Aside from issues of reading data, generating output, and the actual integration algorithm there is only this basic cycle of force evaluation and integration which is carried out every timestep.

Molecular dynamics simulations run for millions of timesteps consuming months of computer time. It is the length of simulations which has led to the use of parallel computing in this field. It is the iterative nature of the molecular dynamics algorithm which produces the challenge, for although efficiently parallelizing independent force evaluations is trivial, the force evaluations for a sequence of timesteps must be individually parallel to realize a speedup. Also, even if force evaluation consumes the vast majority of computer time, it may be advantageous to perform the integration in parallel as well, increasing scalability according to Amdahl's law [1].

Parallelism increases programming complexity and with it the need for sound software engineering practices. This is especially true for programs designed for public use in an academic environment since the primary developers of such codes are often graduate students who tend to move on after obtaining their degrees. In addition, molecular dynamics is not a static field and the users of such software often propose new algorithms and techniques to be added to a working code. Thus, a complex program such as a parallel molecular dynamics code must be sufficiently well designed and documented that it can be maintained and enhanced by future generations of programmers. Those portions of the code which are most likely to be modified, such as the integration algorithm, must therefore be especially clear and modularly separated from the remaining code with well-documented interfaces.

The following sections cover the design goals, decisions, and outcomes of the first two major versions of NAMD and present directions for future development. It is assumed that the reader has been exposed to the basics of molecular dynamics [2–4] and parallel computing [5]. Additional information on NAMD is available electronically [6].

NAMD 1

NAMD [7] was born of frustration with the maintainability of previous locally developed parallel molecular dynamics codes. The primary goal of being able to hand the program down to the next generation of developers is reflected in the acronym NAMD: Not (just) Another Molecular Dynamics code. Specific design requirements for NAMD were to run in parallel on the group's then recently purchased workstation cluster [8] and to use the fast multipole algorithm [9] for efficient full electrostatics evaluation as implemented in DPMTA [10].

Two implementation decisions could be made immediately. First, DPMTA is based on the PVM message-passing library [11] and therefore it was necessary to base NAMD on PVM as well. All communication done by NAMD, however, would use an intermediate interface to allow communications to be easily retargeted to MPI [12] or other standards, and to simplify later implementation of communication optimizations such as combining messages destined for the same processor. Second, after much debate C++ was selected

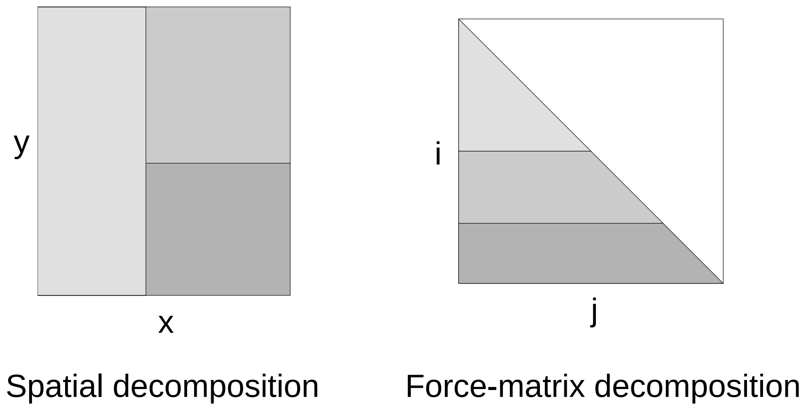


Fig. 1. Nonbonded force evaluation may be distributed among processors according to atomic coordinates, as in spatial decomposition (left), or according to the indices of the interacting atoms, as in force-matrix decomposition (right). Shades of gray indicate processors to which interactions are assigned.

as the development language. This was based on the desire to use an object-oriented design and on prior good experiences in developing the visualization program VMD [13]. There was concern that existing C++ compilers were not uniformly mature and hence to ensure portability across platforms exotic features (at the time) such as templates would be avoided in NAMD. In order to avoid possible performance problems [14] time-critical sections of code like force evaluation were reduced to plain C, many functions were inlined, and virtual functions were avoided.

Parallel molecular dynamics codes are distinguished by their methods of dividing the force evaluation workload among the processors (or *nodes*). The force evaluation is naturally divided into bonded terms, approximating the effects of covalent bonds and involving up to four nearby atoms, and pairwise nonbonded terms, which account for the electrostatic, dispersive, and electronic repulsion interactions between atoms that are not covalently bonded. The nonbonded forces involve interactions between all pairs of particles in the system and hence require time proportional to the square of the number of atoms. Even when neglected outside of a cutoff, nonbonded force evaluations represent the vast majority of work involved in a molecular dynamics simulation.

Methods of decomposing the nonbonded force evaluation fall into two classes, *spatial decomposition* [15] in which atoms and their interactions are divided among processors based on their coordinates, and *force-matrix decomposition* [16] in which the calculation of the interaction between a pair of atoms is assigned to a processor without considering the location of either atom (Fig. 1). Spatial decomposition scales better to large numbers of

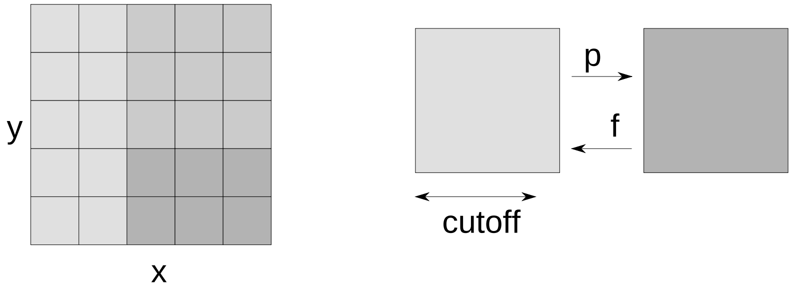


Fig. 2. Patches divide the simulation space into a regular grid of cubes, each larger than the nonbonded cutoff. Interactions between atoms belonging to neighboring patches are calculated by one of the patches which receives a positions message (p) and returns a force message (f). Shades of gray indicate processors to which patches are assigned.

processors because it takes locality of communication into account, while force-matrix decomposition is easier to implement and load-balance.

NAMD implemented spatial decomposition and addressed the load balancing issue by dividing the simulation space into a large number of cubes called *patches* (Fig. 2). A patch serves three purposes. First, it is a region of space larger than the cutoff distance for nonbonded force evaluation, and can therefore function in a cell list or linked-list method [17] to accelerate distance checking for nonbonded interactions. Second, a patch is a unit of parallelizable work which can be reassigned to balance load among processors—each node possessing several patches. Finally, a patch is a *message-driven object* that receives atomic coordinates from some of its neighboring patches, calculates interactions, and returns forces while sending coordinates to and receiving forces from its other neighbors.

Message-driven execution [18] is a parallel processing technique in which communication latency is hidden by overlapping computation and communication. This is achieved by executing computations specified by the messages as they arrive instead of in a fixed serial order. Every coordinate message that arrives contains data that allows some subset of the force evaluation to be carried out, primarily nonbonded interactions between atoms of the patch which sent the message and those of the patch which receives it. Messages are prioritized such that those which generate off-node communication (such as position messages from off-node patches) are processed before messages between patches on the same node. (Actually, the main message loop in NAMD attempted to receive each of the several types of messages in order of priority, providing only roughly prioritized message execution.)

NAMD was implemented in an object-oriented fashion (Fig. 3). Patches, the encapsulated communication subsystem, the molecular structure, and various output methods were objects. Every patch owned specialized objects

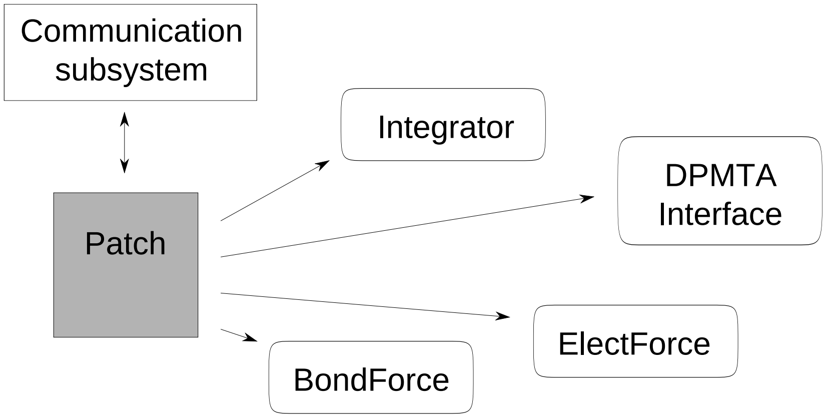


Fig. 3. NAMD 1 employs a modular, object-oriented design in which patches communicate via an encapsulated communication subsystem. Every patch owns an integrator and a complete set of force objects for bonded (BondForce), nonbonded (ElectForce), and full electrostatic (DPMTA) calculations.

responsible for integration, the several types of force calculations, and the interface to the DPMTA full electrostatics package. This made the system modular in that new forces or integration methods could be added with minimal modification of existing code.

Once it entered production mode, the strengths and weaknesses of the NAMD design could be determined. C++, message-driven execution, and the concept of patches had each proven their utility and the program performed well on small numbers of processors. There were also some problems. Load balancing was hampered because most of the work was concentrated in a few patches near the center of the system (simulations lacked periodic boundary conditions). A patch with multiple neighbors on the same node would send several identical messages to that node; the workaround for this unnecessarily complicated the communication system. Finally, it was found that a patch-centric flow of control created a mixing of the essentially serial simulation algorithm with the parallel logic for responding to incoming messages, obfuscating both and requiring an understanding of the message structure in order to make trivial modifications to the iterative loop. For these reasons, it was decided that a major redesign was necessary and work began on NAMD 2.

NAMD 2

NAMD 2 added several new design goals. First, parallel performance needed to be increased through more parallelism and better load balancing. Second, communication efficiency needed to be improved without adding application-

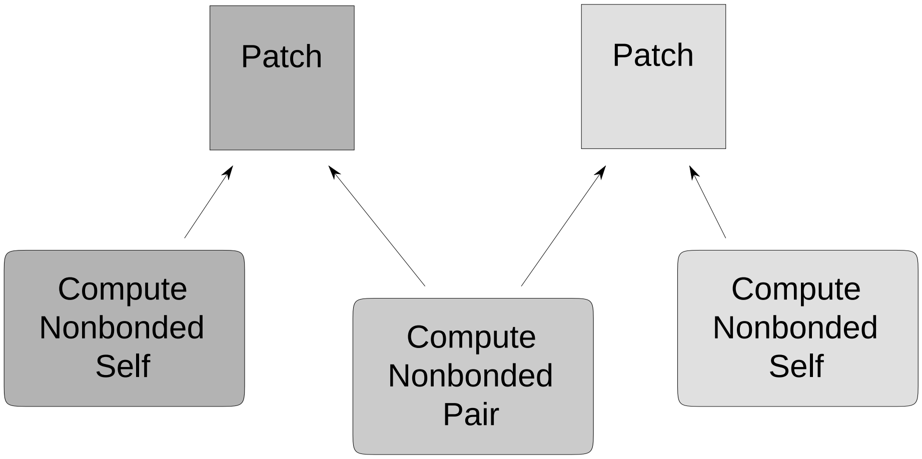


Fig. 4. In NAMD 2 forces are calculated not by force objects owned by individual patches, but rather by independent compute objects which depend on one or more patches for atomic coordinates. As suggested by shading in this illustration, a compute object need not reside on the same node as the patches upon which it depends.

specific code to the communication subsystem. Third, the simulation algorithm's outer loop should be made explicit and parallel logic in this section of code eliminated. Finally, the design needed to be able to take advantage of the eventual availability of kernel-level threads on a newly-acquired cluster of symmetric multiprocessor shared-memory workstations; a node would be able to control several processors in a common memory space.

NAMD 2 did not use PVM as its parallel communication protocol, switching instead to the Charm++/Converse system developed locally by the group of L. V. Kalé. While NAMD 1 simulated message-driven execution in PVM, Charm++ [19] provides direct support for NAMD's message-driven object paradigm and provides tools for analyzing the performance of parallel programs. (A Charm++ version of NAMD 1 was also implemented but maintaining both versions required too much manpower.) Converse [20] is an underlying communications layer which is portable to most parallel machines and features the ability to let multiple parallel languages coexist in a single code. This later feature allowed us to continue using the PVM-based DPTMA package [21]. Converse also incorporates multiple threads into its messaging system, the utility of which is described below. NAMD 2 also made aggressive use of C++ templates in order to provide efficient yet safe and convenient container classes and employed a more thoroughly object-oriented design.

In order to improve parallelism and load balancing, a hybrid force-spatial decomposition scheme was adopted in NAMD 2. Rather than decomposing

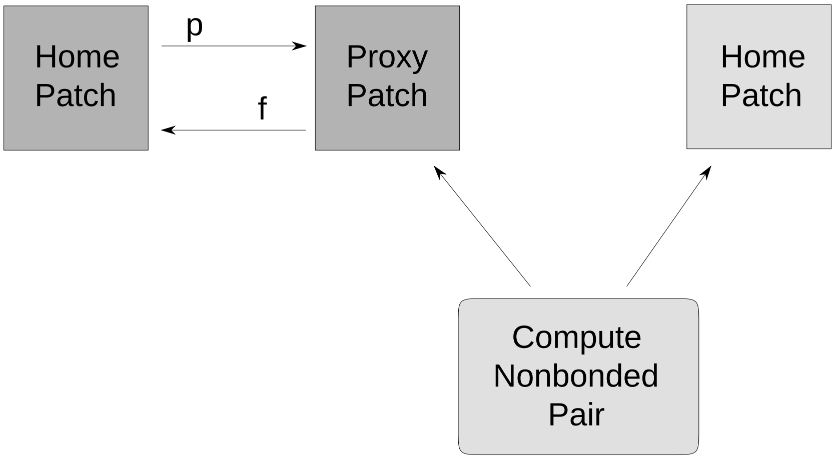
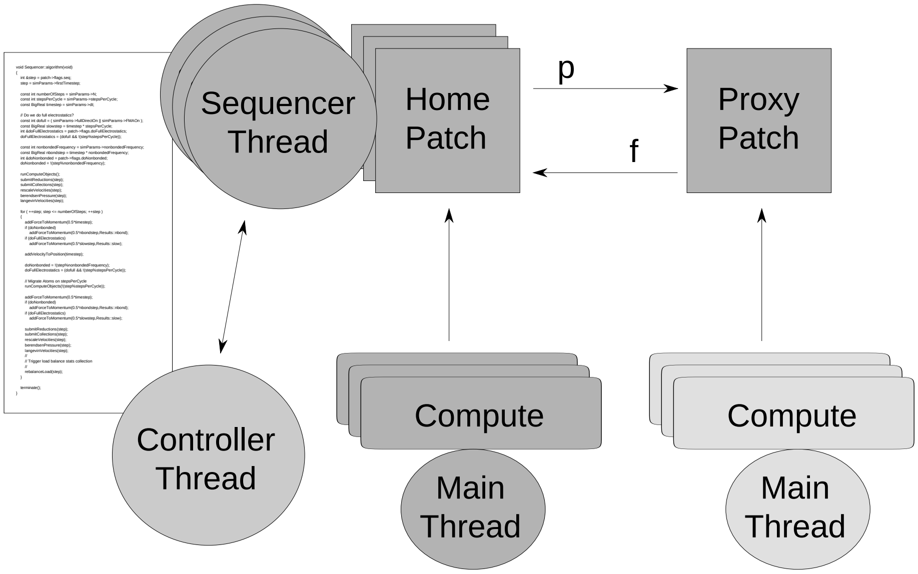


Fig. 5. Compute objects requiring off-node patches do not engage in off-node communication but rather interact with local proxy patches. When force evaluations are required the home patch sends positions messages (p) to its proxies and receives force messages (f) containing the results of off-node calculations. The proxy patch in this illustration exists on the same node as the compute object but represents the off-node home patch with which it communicates.

the nonbonded computation into regions of space or pairwise atomic interactions, the basic unit of work was chosen to be interactions between atoms in regions of space. This was represented in the object-oriented design of NAMD 2 by moving responsibility for calculating forces from objects owned by a patch to more general *compute objects* that were responsible only for nonbonded interactions between atoms in a pair of patches, or within a single patch (Fig. 4).

Moving responsibility for the force computation away from the patches required a move away from pure message-driven execution to *dependency-driven execution* in which patches control the data (atomic coordinates) needed for compute objects to execute. A compute object, upon creation, registers this dependency with those patches from which it needs data. The patch then triggers force calculation by notifying its dependent compute objects when the next timestep's data is available. Once a compute object has received notification from all of the patches it depends on, it is placed in a prioritized queue for eventual execution.

Load balancing can then be achieved in NAMD 2 by moving compute objects and patches between nodes. But what if a compute object and a patch it depends on are on different nodes? Compute objects individually communicating with off-node patches would generate a huge amount of redundant communication. Therefore, patches are represented on other nodes by *proxy*



```
void Sequencer::algorithm(void)
{
    int &step = patch->flags.seq;
    step = simParams->firstTimestep;

    const int numberOfSteps = simParams->N;
    const int stepsPerCycle = simParams->stepsPerCycle;
    const BigReal timestep = simParams->dt;

    // Do we do full electrostatics?
    const int dofull = ( simParams->fullDirectOn || simParams->FMAOn );
    const BigReal slowstep = timestep * stepsPerCycle;
    int &doFullElectrostatics = patch->flags.doFullElectrostatics;
    doFullElectrostatics = (dofull && !(step%stepsPerCycle));

    const int nonbondedFrequency = simParams->nonbondedFrequency;
    const BigReal nbondstep = timestep * nonbondedFrequency;
    int &doNonbonded = patch->flags.doNonbonded;
    doNonbonded = !(step%nonbondedFrequency);

    runComputeObjects();
    submitReductions(step);
    submitCollections(step);
    rescaleVelocities(step);
    berendsenPressure(step);
    langevinVelocities(step);

    for ( ++step; step <= numberOfSteps; ++step )
    {
        addForceToMomentum(0.5*timestep);
        if (doNonbonded)
            addForceToMomentum(0.5*nbondstep,Results::nbond);
        if (doFullElectrostatics)
            addForceToMomentum(0.5*slowstep,Results::slow);

        addVelocityToPosition(timestep);

        doNonbonded = !(step%nonbondedFrequency);
        doFullElectrostatics = (dofull && !(step%stepsPerCycle));

        // Migrate Atoms on stepsPerCycle
        runComputeObjects(!(step%stepsPerCycle));

        addForceToMomentum(0.5*timestep);
        if (doNonbonded)
            addForceToMomentum(0.5*nbondstep,Results::nbond);
        if (doFullElectrostatics)
            addForceToMomentum(0.5*slowstep,Results::slow);

        submitReductions(step);
        submitCollections(step);
        rescaleVelocities(step);
        berendsenPressure(step);
        langevinVelocities(step);
        //
        // Trigger load balance stats collection
        //
        rebalanceLoad(step);
    }

    terminate();
}
```

Fig. 6. Multiple threads in NAMD 2 allow the integration algorithm to be expressed sequentially as a single function. This function, shown illegibly at left, runs in sequencer threads associated with home patches. A similar function running in a controller thread on the master node communicates with the sequencers to deal with output and global calculations. Compute objects execute in the larger stack space of each node's main thread.

patches, which implement the same interface as *home patches* for dealing with compute objects and handling dependencies but receive coordinates from and send forces to their respective home patches rather than performing integration themselves (Fig. 5). Thus data is replicated on those nodes where it is needed with a minimum of communication while no off-node communication is done by compute objects.

The logic associated with the patch has been greatly simplified by separating compute objects and limiting communication to patches and proxies, but one additional step is needed to fully separate sequential molecular dynamics algorithm from the complex logic of a message-driven parallel code. A *sequencer thread* is associated with every patch. This thread runs a single function which contains an explicit loop over all of the timesteps in the simulation (Fig. 6). In this way, the integration algorithm can be inspected in a single section of code closely resembling the outer loop of a serial molecular dynamics program. All of the parallel logic is hidden inside of a force evaluation function called by the sequencer that simply propagates coordinates to proxies and notifies all registered dependent compute objects that coordinates are available for calculating forces before suspending the sequencer thread. The thread is later awakened when all dependent compute objects and proxies have deposited their forces. A similar *controller thread* on the master node coordinates energy output and global aspects of the integration algorithm such as calculating velocity rescaling factors. Thread suspension is also used to wait for unavailable data such as energies needed for output in the case of the controller or forces needed for integration in the case of the sequencer.

Future Plans

As noted above, one of the goals of NAMD 2 is to take advantage of clusters of symmetric multiprocessor workstations and other non-uniform memory access platforms. This can be achieved in the current design by allowing multiple compute objects to run concurrently on different processors via kernel-level threads. Because compute objects interact in a controlled manner with patches, access controls need only be applied to a small number of structures such as force and energy accumulators. A shared memory environment will therefore contribute almost no parallel overhead and generate communication equal to that of a single-processor node.

Although the current multithreaded implementation of sequencers works well and provides a clearly visible algorithm, threads have several drawbacks. Extra memory is required for multiple stacks, there is overhead from context-switching between threads, and a running sequencer cannot migrate between processors along with its patch. These problems will be addressed by using the Structured Dagger coordination language [22], which enables programmers to specify partial order between entry methods of an object. Using constructs such as overlap, forall, and when-blocks, one can easily express dependencies between entry methods of an object while letting the system do the buffering, bookkeeping, etc. required for the specified flow of control.

Finally, the ultimate in algorithmic flexibility can be achieved by the addition of a scripting language interface to NAMD. Such an interface, most likely based on Tcl [23], will allow the end user to modify the simulation algorithm without recompiling and to implement multi-stage simulation protocols in a single script. By adopting an existing scripting and extension language such as Tcl, Perl or Python [24], the end user will avoid learning a special-purpose language and enjoy the benefits of a well-designed and fully featured programming environment. The success of the Tcl interface in VMD [13], the Theoretical Biophysics Group's biomolecular visualization package, makes this line of development almost inevitable.

Acknowledgements

The primary developers of NAMD 1 were M. Nelson, W. Humphrey, A. Gursoy, A. Dalke and R. Brunner. The primary developers of NAMD 2 were

J. Phillips, A. Shinozaki, R. Brunner, N. Krawetz, M. Bhandarkar and A. Gursoy. NAMD development was performed at the National Institutes of Health Resource for Concurrent Biological Computing under the supervision of principal investigators L.V. Kalé, R. Skeel, and K. Schulten. This work was supported by the National Institutes of Health (NIH PHS 5 P41 RR05969-04 and NIH HL 16059) and the National Science Foundation (NSF/GCAG BIR 93-18159 and NSF BIR 94-23827 EQ). JCP was supported by a Computational Science Graduate Fellowship from the United States Department of Energy.

References

1. Amdahl, G. M.: Validity of the single processor approach to achieve large scale computing capabilities. *In* Proc. AFIPS spring computer conf. vol. 30. AFIPS Press, Reston, Virginia, 1967.
2. Allen, M. P., Tildesley, D. J.: Computer Simulation of Liquids. Oxford University Press, New York, 1987.
3. Brooks III, C. L., Karplus, M., Pettitt, B. M.: Proteins: A Theoretical Perspective of Dynamics, Structure and Thermodynamics. Advances in Chemical Physics, vol. LXXI. John Wiley & Sons, New York, 1988.
4. McCammon, J. A., Harvey, S. C.: Dynamics of Proteins and Nucleic Acids. Cambridge University Press, Cambridge, 1987.
5. Almasi, G. S., Gottlieb, A.: Highly Parallel Computing. 2nd edn. Benjamin/Cummings, Redwood City, California, 1994.
6. Theoretical Biophysics Group. http://www.ks.uiuc.edu/.
7. Nelson, M., Humphrey, W., Gursoy, A., Dalke, A., Kalé, L., Skeel, R. D., Schulten, K.: NAMD— A parallel, object-oriented molecular dynamics program. J. Supercomputing App. **10** (1996) 251–268.
8. Lin, M., Hsieh, J., Du, D. H. C., Thomas, J. P., MacDonald, J. A.: Distributed network computing over local ATM networks. *In* Proceedings of Supercomputing '94. IEEE Computer Society Press, Los Alamitos, California, 1994.
9. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulation. J. Comp. Phys. **73** (1987) 325–348.
10. Rankin, W., Board, J.: A portable distributed implementation of the parallel multipole tree algorithm. IEEE Symposium on High Performance Distributed Computing. Duke University Technical Report 95-002.
11. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts, 1994.
12. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge, Massachusetts, 1995.
13. Humphrey, W. F., Dalke, A., Schulten, K.: VMD – Visual molecular dynamics. J. Mol. Graphics. **14** (1996) 33–38.
14. Haney, S. W.: Is C++ fast enough for scientific computing? Computers in Physics. **8** (1994) 690–694.
15. Clark, T., Hanxleden, R., McCammon, J., Scott, L.: Parallelizing molecular dynamics using spatial decomposition. *In* Proceedings of the scalable high performance computing conference, May 23-25, 1994, Knoxville, Tennessee. IEEE Computer Society Press, Los Alamitos, California, 1994.

16. Plimpton, S., Hendrickson, B.: A New Parallel Method for Molecular Dynamics Simulation of Macromolecular Systems. 1994. Technical Report SAND94-1862. Sandia National Laboratories.
17. Hockney, R. W., Eastwood, J. W.: Computer Simulation Using Particles. McGraw-Hill, New York, 1981.
18. Kalé, L. V.: The Chare Kernel parallel programming language and system. *In* Proceedings of the international conference on parallel processing vol. II. CRC Press, Boca Raton, Florida, 1990.
19. Kalé, L., Krishnan, S.: Charm++: A Portable Concurrent Object Oriented System Based on C++. *In* Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications. A. Paepcke, editor. ACM Press, New York, N.Y., 1993.
20. Kalé, L. V., Bhandarkar, M., Jagathesan, N., Krishnan, S., Yelon, J.: Converse: An interoperable framework for parallel programming. *In* Proceedings of the 10th international parallel processing symposium. IEEE Computer Society Press, Los Alamitos, California, 1996.
21. Kalé, L. V., Bhandarkar, M., Brunner, R., Krawetz, N., Phillips, J., Shinozaki, A.: NAMD: A case study in multilingual parallel programming. *In* Proceedings of the 10th international workshop on languages and compilers for parallel computing. Springer-Verlag, Berlin, 1998.
22. Kalé, L. V., Bhandarkar, M.: Structured Dagger: A coordination language for message-driven programming. *In* Proceedings of the second international europar conference. Lecture Notes in Computer Science, vol. 1123–1124. Springer-Verlag, Berlin, 1996.
23. Ousterhout, J.: Tcl and the Tk Toolkit. Addison-Wesley, Reading, Massachusetts, 1994.
24. Watters, A., Rossum, G. V., Ahlstrom, J. C.: Internet Programming With Python. M & T Books, Sebastopol, California, 1996.