RC 21965 (98713) 13FEB01 Computer Science/Mathematics

Research Report

Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflop Computer

George S. Almasi, Călin Caşcaval, José G. Castaños, Monty Denneau, Wilm Donath, Maria Eleftheriou, Mark Giampapa, Howard Ho, Derek Lieber, José E. Moreira, Dennis Newns, Marc Snir, Henry S. Warren, Jr.
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center [Publications 16-220 ykt] P. O. Box 218, Yorktown Heights, NY 10598. email: reports@us.ibm.com
Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

This page intentionally left blank.

Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflop Computer*

George S. Almasi Călin Caşcaval José G. Castaños Monty Denneau
Wilm Donath Maria Eleftheriou Mark Giampapa Howard Ho Derek Lieber
José E. Moreira Dennis Newns Marc Snir Henry S. Warren, Jr.

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218

Abstract

In this paper we demonstrate that a class of molecular dynamics applications can be effectively parallelized on the scale required for efficient execution on a massively parallel computer with millions of concurrent threads of execution. Such cellular architectures have been advocated as one approach to building Petaflop-scale computers, as proposed in the IBM Blue Gene project. Starting from the sequential version of a well known molecular dynamics code, we developed a new application that exploits the multiple levels of parallelism in a cellular architecture. We perform both an analytical and a simulation study of the behavior of this application when executed on a very large number of threads. We demonstrate that this class of applications can execute efficiently on a large cellular machine and, thus, provide support for this approach to achieving a Petaflop computer.

1 Introduction

Now that several Teraflop-scale machines have been deployed in various industrial, governmental, and academic sites, the high-performance computing community is starting to look for the next big step: Petaflop-scale machines. At least two very different approaches have been advocated for the development of the first generation of such machines. On one hand, projects like HTMT (`http://htmt.cacr.caltech.edu`) propose the use of thousands of very high speed processors (hundreds of GigaHertz). On the other hand, projects like IBM's Blue Gene (`http://www.research.ibm.com/bluegene`) advance the idea of using a very large number (millions) of modest speed processors (hundreds of MegaHertz). These two approaches can be seen as the extremes of a wide spectrum of choices. We are particularly interested in analyzing the feasibility of the latter, Blue Gene style, approach.

Million-processor machines can be built today, in a relatively straightforward way, by adopting a *cellular* architecture: a basic building-block containing processors, memory, and interconnect support (preferably implemented on a single silicon chip) is replicated many times following a regular pattern. Building a cellular machine to deliver a Petaflop/s – or 10^{15} floating-point operations per second – is quite a challenge, which can only be justified if it can be demonstrated that applications can execute efficiently on such a machine.

In this paper, we report on the results of analytical- and simulation-based studies on the behavior of a computational molecular dynamics algorithm. This is one example of a class of applications that can indeed exploit the massive levels of parallelism offered by a Petaflop-scale cellular machine, as discussed in [8]. This parallel application was derived from the serial version of a molecular dynamics code developed at the

*Corresponding author: José E. Moreira (`jmoreira@us.ibm.com`)

University of Pennsylvania (`http://www.cmm.upenn.edu/~moore/code/code.html`). The application was rewritten with new partitioning techniques to take advantage of multiple levels of parallelism. We analyzed the behavior of this application on a cellular architecture with millions of concurrent threads of execution. Our target cellular architecture is similar to some of the designs that have been proposed for IBM's Blue Gene project. In addition, we developed an analytical model for the performance of our application and compared it with direct simulation measurements.

The quantitative results for our molecular dynamics code demonstrate that this class of applications can successfully exploit a Petaflop-scale cellular machine. In terms of absolute performance, simulations indicate that we can achieve 0.14 Petaflop/s (0.14×10^{15} floating-point operations per second) and 0.87 Petaop/s (0.87×10^{15} operations per second) of sustained performance. In terms of speed of molecular dynamics computation, we integrate the equations of motion for a typical problem with 32,000 atoms at the rate of one time-step every $375\mu s$. The same problem can be solved on a Power 3 workstation with peak performance of 800 Mflop/s in 140 seconds per time step, using the sequential version of the code. Thus, on a machine with 1,250,000 times the total peak floating-point performance of a uniprocessor, we achieve a speedup of 368,000.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the cellular machine we consider, at the level necessary to understand how our molecular dynamics application can be parallelized for efficient execution. Section 3 introduces the parallel molecular dynamics algorithm we used. It also presents an analytical performance model for the execution of that algorithm on our cellular architecture. Section 4 describes the simulation and experimental infrastructure. We use this infrastructuring in deriving experimental performance results which are presented, together with the results from the analytical model, in Section 5. Finally, Section 6 presents our conclusions.

2 A Petaflop cellular machine

We are interested in investigating how a machine like IBM's Blue Gene would perform on a class of molecular dynamics applications. The fundamental premise in this design is that performance is obtained by exploiting massive amounts of parallelism, rather than the very fast execution of any thread of control. The various design and technological impacts of this premise are beyond the scope of this paper and we restrict ourselves to describing and analyzing a particular design.

The building block of our cellular architecture is a *node*. A node is implemented in a single silicon chip and contains memory, processing elements, and interconnection elements. A node can be viewed as a single-chip shared-memory multiprocessor. Multiple nodes are interconnected to form a much larger system. In our design, a node contains 16 MB of shared memory and 256 instruction units. Each instruction unit is associated with one thread of execution, giving 256 simultaneous threads of execution in one node, and executes instructions in a thread strictly in order. Each group of 8 threads shares one data cache and one floating-point unit. Such simplifications are necessary to keep instruction units simple, resulting in large numbers of them in a die. The floating-point units (32 in a node) are pipelined and can perform a multiply and an add on every cycle. With a 500 MHz clock cycle, this translates into 1 Gflop/s of peak performance per floating-point unit, or 32 Gflop/s of peak performance per node. Each node also has six channels of communication, for direct interconnection with up to six other nodes. With 16-bit wide channels operating at 500 MHz, a communication bandwidth of 1 Gbyte/s per channel, on each direction, is achieved. Nodes communicate by streaming data through these channels.

Our design for a node is ambitious, but within the realm of current or near-future silicon technology. Combined logic-memory microelectronic processes will soon deliver chips with hundreds of millions of transistors. Several research groups have advanced processor-in-memory designs that rely on that technology. Examples include the Illinois FlexRAM [4, 9] and Berkeley IRAM [5] projects.

For the system design, we use the six communication channels in each node to interconnect them in a three-dimensional mesh configuration. A node is connected to two neighbors along each axis, X, Y, and Z. (Nodes on the faces or along the edges of the mesh have fewer connections.) We use a mesh topology because of its regularity and because it can be built without any additional hardware. We directly connect the communication channels of a node to the communication channels of its neighbors. With a $32 \times 32 \times 32$ three-dimensional mesh of nodes, we build a system of 32,768 nodes. Since each node attains a peak computation rate of 32 Gflop/s, the entire system delivers a peak computation rate of approximately 1 Petaflop/s.

An application running on this Petaflop machine must exploit both inter- and intra-node parallelism. First, the application is decomposed into multiple tasks and each task assigned to a particular node. As discussed previously, the tasks can communicate only through data streams. Second, each task is decomposed into multiple threads, each thread operating on a subset of the problem assigned to the task. The threads in a task interact through shared-memory. The goal of the scalability analysis presented in Section 5 is to demonstrate that at least one class of molecular dynamics applications can be structured to effectively exploit the parallelism offered by cellular machines on the scale of IBM's Blue Gene.

3 The molecular dynamics algorithm

The goal of a molecular dynamics algorithm is to determine how the state of a molecular system evolves with time. Given a molecular system with a set A of n atoms, the state of each atom i at time t can be described by its mass m_i, its charge q_i, its position $\vec{x}_i(t)$ and its velocity $\vec{v}_i(t)$. The evolution of this system is governed by the equation of motion

$$m_i \frac{d^2\vec{x}_i(t)}{dt^2} = \vec{F}_i(\{x_j(t)\}) \tag{1}$$

subject to initial conditions $\vec{x}_i(0) = \vec{x}_i^0$ and $\vec{v}_i(0) = \vec{v}_i^0$ for each atom i, where $\vec{F}_i$ is the force acting on atom i and $\vec{x}_i^0$ and $\vec{v}_i^0$ are the initial position and velocity, respectively, of atom i. Equation 1 can be integrated numerically for a particular choice of time step Δt. The positions of the atoms $\{x_j(t)\}$ at time t are used to compute the forces $\vec{F}_i$ at time t. Those forces are then used to compute the accelerations at time t. Velocities and accelerations at time t are finally used to compute the new positions and velocities at time $t + \Delta t$, respectively. The simulation cell is typically in the form of a box, which is replicated indefinitely in all three dimensions, giving rise to a periodic system.

3.1 A molecular dynamics algorithm

The force $\vec{F}_i(\{x_j(t)\})$ applied on an atom i at time t is the vector sum of the pairwise interactions of that atom i with all the other atoms j in the system. Those pairwise interactions can take several forms, as described in [1]. We divide them into two main groups: *intra-molecular* forces and *inter-molecular* forces. Intra-molecular forces occur between adjacent or close-by atoms in the same molecule. There are four different types of intra-molecular forces: *bonds*, *bends*, *torsions*, and *one-four* forces. Each one of these forces has a different expression. Inter-molecular forces occur between any pair of atoms, and take two forms: *Lennard-Jones* (or *van der Walls*) forces, and *electrostatic* (or *Coulombic*) forces. The Lennard-Jones forces decay rapidly with distance. Therefore, for any given atom i, it is enough to consider the Lennard-Jones interactions with the atoms inside a sphere centered at atom i. The radius of this sphere is called the *cutoff* radius.

When computing electrostatic forces, one has to take into account the periodic nature of the molecular system. One of the most commonly used approaches is the *Ewald* summation [3]. In this method, the com-

putation of electrostatic forces is divided into two fast converging sums: a real space part and a reciprocal space, also called *k-space*, part. For the real space part, interactions are computed between an atom i and all the atoms inside a sphere centered at atom i, as for Lennard-Jones forces. For the computation in reciprocal space, first a set K of reciprocal space vectors is computed. Then, for each $\vec{k} \in K$, we compute $\eta_{\vec{k}}$, the Fourier transform of the point charge distribution (structure factor) for that particular value of $\vec{k}$, as

$$\eta_{\vec{k}} = \sum_{i=0}^{n-1} q_i e^{j\vec{k}\cdot\vec{x}_i}, \tag{2}$$

where n is the number of atoms, q_i and x_i are the charge and position of atom i, respectively. The $\eta_{\vec{k}}$ terms are also called *k-factors*. The contribution of the reciprocal space part to the electrostatic force acting on each atom can then be computed with a summation for all values of $\vec{k}$:

$$\vec{F}_i^k = \sum_{\forall \vec{k} \in K} \vec{f}^k(q_i, \eta_{\vec{k}}). \tag{3}$$

The $\vec{f}^k$ forces are also called *k-forces*. The exact formula for function $\vec{f}^k(\cdot)$ is not important for structuring the parallelization; the precise expression is specified in [1]. The point we want to make is that for each atom we need to perform a summation over all $\vec{k} \in K$.

An efficient parallel molecular dynamics algorithm requires the various forces to be distributed evenly among compute nodes. Our molecular dynamics application uses an extension of the decomposition of inter-molecular forces described in [6]. Although our cellular machine is a three-dimensional mesh of nodes, we view its p^3 nodes as a logical two-dimensional mesh of c columns and r rows. (In particular, we use $c \times r = p^3$.) The two- to three-dimensional mesh embedding we use is described later. For now, let us consider the two-dimensional logical mesh.

We partition the set of atoms A in two ways: a *target partition* of A into r sets $\{T_0, T_1, \ldots, T_{r-1}\}$, each of size $\leq \left\lceil \frac{|A|}{r} \right\rceil$, and a *source partition* of A into c sets $\{S_0, S_1, \ldots, S_{c-1}\}$ of size $\leq \left\lceil \frac{|A|}{c} \right\rceil$. We replicate the target sets across rows of the two-dimensional mesh. Every node in row i contains a copy of the atoms in target set T_i. Similarly, we replicate the source sets across columns. Every node in column j contains a copy of the atoms in source set S_j.

Using the decomposition described above, node $p_{i,j}$ (row i and column j of the logical mesh) can compute locally and with no communication the Lennard-Jones and real space electrostatic forces between atoms in source set S_j and target set T_i. As previously mentioned, only the interactions between atoms that are closer than a certain cutoff radius are actually computed.

To compute the reciprocal space electrostatic forces, the set K of reciprocal space vectors is partitioned into c sets $\{K_0, K_1, \ldots, K_{c-1}\}$, each of size $\approx \left\lceil \frac{|A|}{c} \right\rceil$. We replicate each set K_j along all nodes in column j of the logical mesh. The actual computation is performed in three phases. In the *k-factors* phase, every node $p_{i,j}$ computes the contributions of the atoms in T_i to all $\eta_{\vec{k}}$ such that $\vec{k} \in K_j$. An all-reduction of these contributions along the columns of the two-dimensional mesh constitutes the second phase. After this reduction, every node $p_{i,j}$ will have the value of $\eta_{\vec{k}}$ for all $\vec{k} \in K_j$. In the third and final phase, also called the *k-forces* phase, the $\eta_{\vec{k}}$ values are used in each node to compute the contribution to the F^k forces, described in Equation (3), for each of the atoms in set T_i.

Once all inter-molecular forces are computed, we perform an all-reduction (a summation) of forces along the rows of the two-dimensional logical mesh. As a result, every node in row i obtains the resulting forces over all its atoms in T_i.

The computation of intra-molecular forces (bonds, bends, torsions, and one-four) is replicated on every column of the logical mesh. (Note that every column has an entire set of atoms, formed by the union of its T

sets.) We partition the atoms among the T_i sets so that adjacent atoms in a molecule are assigned to the same or adjacent target sets. The only communication required to compute the intra-molecular forces is to update the positions of the atoms in the same molecule that are split between adjacent rows of the two-dimensional logical mesh.

Once we have computed all the forces over the atoms in T_i we can compute their new positions and velocities. We finally update the positions of the atoms in S_j from the atoms in T_i by performing a broadcast operation along the columns of the two-dimensional mesh. In this broadcast phase each node $p_{i,j}$ sends the positions of the atoms in $T_i \cap S_j$ to all the nodes in column j. Correspondingly, it receives the positions of the atoms in $S_j - T_i \cap S_j$ from other nodes in that column.

Interactions between atoms that are close together need to be evaluated more often than interactions between atoms that are far apart. For this reason, we use a multi-timestep algorithm similar to RESPA [10, 11] to reduce the frequency of inter-molecular forces computation. We illustrate this concept in Figure 1, where a computational time step is divided into a sequence of short, intermediate and long steps. Between steps, computation threads wait in a barrier (indicated by "B" in Figure 1) until the positions of all atoms are received from other nodes and updated. The threads then compute the forces required in the step in parallel and synchronize the update of a shared vector of forces using critical regions. Threads wait in a second barrier until all the forces in the step have been computed before updating the positions and velocities of the atoms. Figure 1 also presents summary performance numbers, which will be discussed in Section 5.

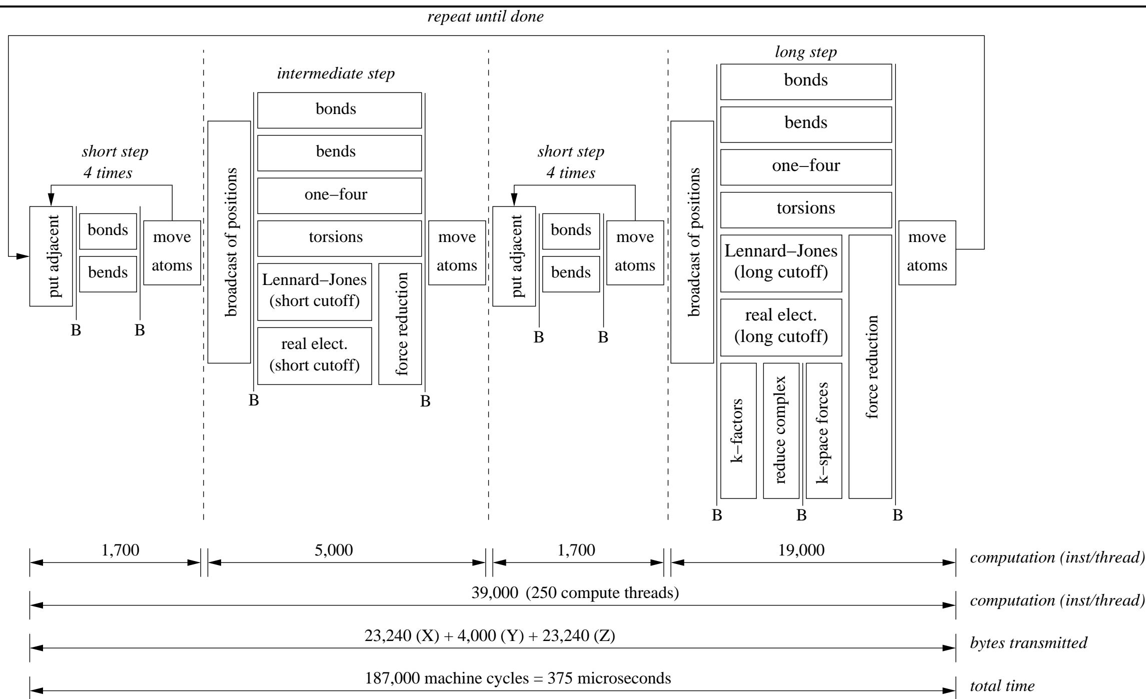

Figure 1: Flowchart for the molecular dynamics computation for a complete long time step and computational performance of the molecular dynamics application.

We now focus on the embedding of the two-dimensional logical mesh on the three-dimensional mesh. We consider a two-dimensional mesh of 128 rows and 256 columns. Each column of the logical mesh (128 nodes) is mapped onto one physical plane of the three-dimensional cellular machine, as shown in Figure 2. Note that 8 logical columns are embedded in one physical plane. The broadcast of positions and the reduction of complex numbers in reciprocal space in each column of the logical mesh do not interfere with

broadcasts and reductions in other logical columns and use only the X and Y dimensions of the three-dimensional mesh. The reduction of forces along each row of the two-dimensional mesh is performed between adjacent planes of the three-dimensional physical mesh and uses disjoint wires in the Z dimension. The nodes of each logical row assigned to the same plane also need to perform a reduction but only one wire is needed for each row along the X dimension.

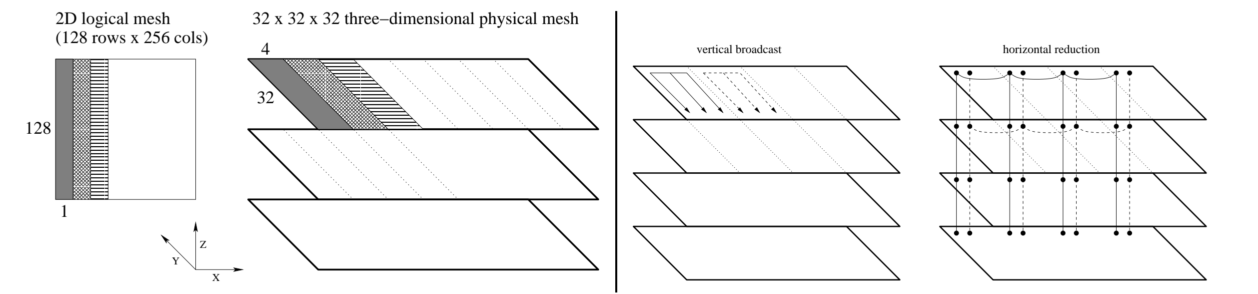

Figure 2: Mapping columns of the logical mesh to the physical machine *(left)*. Communication patterns for vertical broadcast of positions and horizontal reduction of forces *(right)*.

3.2 A performance model for the molecular dynamics computation

Let T_s be the execution time of a short step, let T_i be the execution time of an intermediate step, and let T_l be the execution time of a long step. Then, the total execution time T of one computational time step (as shown in Figure 1) can be computed as

$$T = sT_s + T_i + T_l, \tag{4}$$

where s is the number of short steps in one computational time step. We can decompose the time for each step into two parts: a *computation* time $T^p_{[s,i,l]}$ and an *exposed communication* time $T^m_{[s,i,l]}$. The exposed communication time is the part of the communication time that is not overlapped with computation.

3.2.1 Computation time modeling

Let N_{bond} and N_{bend} be the total number of bonds and bends, respectively, that need to be computed by a node. Let N_{thread} be the number of computation threads available in the node. Let T_{bond} and T_{bend} be the time it takes to compute one bond and one bend, respectively, and let T_{move} be the time needed to update an atom's position and velocity. Let N_{atom} be the number of target atoms assigned to the node. Finally, let $T_{\text{barrier}}(n)$ be the time it takes to perform a barrier operation on n threads. The time for a short step can be expressed as

$$T^p_s = \left\lceil \frac{N_{\text{bond}}}{N_{\text{thread}}} \right\rceil T_{\text{bond}} + \left\lceil \frac{N_{\text{bend}}}{N_{\text{thread}}} \right\rceil T_{\text{bend}} + 2T_{\text{barrier}}(N_{\text{thread}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{move}}. \tag{5}$$

Let N_{torsion} be the number of torsions that need to be computed by a node. Let N_{LJshort} and N_{ESshort} be the number of Lennard-Jones and (real part) electrostatic forces within a short cutoff, respectively, that need to be computed in a node. Let T_{LJ} and T_{ES} be the time it takes to compute one Lennard-Jones and one (real part) electrostatic force, respectively. The time for an intermediate step can be expressed as

$$\begin{aligned} T^p_i = {} & \left\lceil \frac{N_{\text{bond}}}{N_{\text{thread}}} \right\rceil T_{\text{bond}} + \left\lceil \frac{N_{\text{bend}}}{N_{\text{thread}}} \right\rceil T_{\text{bend}} + 2T_{\text{barrier}}(N_{\text{thread}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{move}} + \\ & \left\lceil \frac{N_{\text{torsion}}}{N_{\text{thread}}} \right\rceil T_{\text{torsion}} + \left\lceil \frac{N_{\text{LJshort}}}{N_{\text{thread}}} \right\rceil T_{\text{LJ}} + \left\lceil \frac{N_{\text{ESshort}}}{N_{\text{thread}}} \right\rceil T_{\text{ES}}. \end{aligned} \tag{6}$$

Let N_{LJlong} and N_{ESlong} be the number of Lennard-Jones and (real part) electrostatic forces within a long cutoff, respectively, that need to be computed in a node. Let $N_{k\text{-factor}}$ be the number of $\eta_{\vec{k}}$ terms in the node, i.e., the size of the K set in the node. Let $T_{\text{eta}}(N_{k\text{-factor}})$ be the time it takes to compute one atom's contribution to the $\eta_{\vec{k}}$ terms in the node. Let $T_{\text{redux}}(N_{k\text{-factor}})$ be the time it takes to add two sets of contributions to the $\eta_{\vec{k}}$ terms. Let $T_{k\text{-factor}}(N_{k\text{-factor}})$ be the time it takes to finalize computing the $\eta_{\vec{k}}$ terms in a node. Finally, let $T_{k\text{-force}}$ be the time it takes to compute the k-force on one atom. The time for a long step can be expressed as

$$\begin{aligned} T_l^p = \; & \left\lceil \frac{N_{\text{bond}}}{N_{\text{thread}}} \right\rceil T_{\text{bond}} + \left\lceil \frac{N_{\text{bend}}}{N_{\text{thread}}} \right\rceil T_{\text{bend}} + 3T_{\text{barrier}}(N_{\text{thread}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{move}} + \\ & \left\lceil \frac{N_{\text{torsion}}}{N_{\text{thread}}} \right\rceil T_{\text{torsion}} + \left\lceil \frac{N_{\text{LJlong}}}{N_{\text{thread}}} \right\rceil T_{\text{LJ}} + \left\lceil \frac{N_{\text{ESlong}}}{N_{\text{thread}}} \right\rceil T_{\text{ES}} + \\ & \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{eta}}(N_{k\text{-factor}}) + \lceil \log_2(N_{\text{thread}}) \rceil T_{\text{redux}}(N_{k\text{-factor}}) + \\ & \left\lceil \frac{N_{k\text{-factor}}}{N_{\text{thread}}} \right\rceil T_{k\text{-factor}}(N_{k\text{-factor}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{k\text{-force}}(N_{k\text{-factor}}). \end{aligned} \tag{7}$$

3.2.2 Communication time modeling

The dominant communication operations in our molecular dynamics code are the force reductions along the rows and the k-factor ($\eta_{\vec{k}}$) reductions and position broadcasts along the columns of the two-dimensional logical mesh. Reductions and broadcasts are implemented in our cellular machine by organizing the nodes within a logical row or a logical column according to a spanning tree. The communication of atom positions to atoms adjacent in the molecule make a small contribution to the total communication and are always performed between adjacent nodes along columns of the logical mesh.

We want to compute the time for four communication operations, which operate on vectors of elements. Let $T_{\text{positions}}$ be the time to broadcast positions along columns, $T_{k\text{-factors redux}}$ the time to reduce k-factors along columns, T_{forces} the time to reduce forces along rows and T_{put} the time to put a new position in a neighboring row. The time for each of these operations can be decomposed into a latency time and a transfer time. The latency time is the time to complete the operation for the first element of a vector. The transfer time is the rate at which each element is processed times the number of elements in the vector. In equation form:

$$T_{\text{positions}} = T_{\text{positions}}^{\text{latency}} + T_{\text{triplet}} N_{\text{source}}, \tag{8}$$

$$T_{k\text{-factors redux}} = T_{k\text{-factors redux}}^{\text{latency}} + T_{\text{complex}} N_{k\text{-factor}}, \tag{9}$$

$$T_{\text{forces}} = T_{\text{forces}}^{\text{latency}} + T_{\text{triplet}} N_{\text{target}}, \tag{10}$$

$$T_{\text{put}} = T_{\text{put}}^{\text{latency}} + T_{\text{triplet}} N_{\text{put}}, \tag{11}$$

where N_{source} and $N_{k\text{-factor}}$ are the number of source atoms and k-factors assigned to the column, respectively, and N_{target} is the number of target atoms assigned to the row. N_{put} is the number of puts to neighbors that a node has to perform. T_{triplet} and T_{complex} are the time to transfer one triplet (force or position) or one complex number (k-factor), respectively. Each position and force is 24 bytes long (three double-precision floating-point numbers) and each k-factor is 16 bytes long (two double-precision floating-point numbers).

Let $N_{\text{hop_c}}^i$ be the number of hops (nodes) between a node i in a column and the root of the fan-in and fan-out trees for intra-column operations. Let T_{hop} be the time to go through a hop (node) in the cellular machine interconnect. Then

$$T_{\text{positions}}^{\text{latency}} = 2 \max_{\forall i} (N_{\text{hop_c}}^i) T_{\text{hop}}, \tag{12}$$

where the factor of two accounts for the round trip. Let $N^i_{\text{add_c}}$ be the number of additions that need to be performed to reduce an item originating at a node i in the column until it gets to the root of the fan-in tree. Let T_{add} be the time to add one item. Then

$$T^{\text{latency}}_{k\text{-factors redux}} = 2 \max_{\forall i}(N^i_{\text{hop_c}})T_{\text{hop}} + \max_{\forall i}(N^i_{\text{add_c}})T_{\text{add}}. \tag{13}$$

The analysis for the force reduction along the rows is similar. Let $N^i_{\text{hop_r}}$ be the number of hops between a node i in a row and the root of the fan-in and fan-out trees for intra-row operation. Let $N^i_{\text{add_r}}$ be the the number of additions that need to be performed to reduce an item originating at a node i in the row until it gets to the root for the fan-in tree. Then

$$T^{\text{latency}}_{\text{forces}} = 2 \max_{\forall i}(N^i_{\text{hop_r}})T_{\text{hop}} + \max_{\forall i}(N^i_{\text{add_r}})T_{\text{add}}. \tag{14}$$

From that table, we can derive worst case estimates for the exposed communications times $T^m_{[s,i,;]}$. In case there is no overlap between computation and communication,

$$T^m_i = T_{\text{positions}} + T_{\text{forces}}, \tag{15}$$
$$T^m_l = T_{\text{positions}} + T_{k\text{-factors redux}} + T_{\text{forces}}, \tag{16}$$
$$T^m_s = T_{\text{put}}. \tag{17}$$

The upper bound on the exposed communication time is

$$T^m = 8 \times T^m_s + T^m_i + T^m_l \tag{18}$$

4 Simulation environment

To complement the analytical modeling described above, we also executed the molecular dynamics application on an instruction-level simulator of our cellular machine. Our machine has a proprietary instruction set, which is typically RISC (load/store architecture, three-register instructions) in nature. The application was coded in C and compiled with the gcc (version 2.95.2) compiler, properly modified to generate code for our instruction set. Thread creation and management inside a node is performed at the application level by calls to a pthread-compatible library. Inter-node communication is accomplished through a proprietary communication library that implements put (one-sided communication), broadcast, and reduce operations.

Each node of the machine runs a resident system kernel, which executes with supervisor privileges. The purpose of the kernel is twofold: first, to protect machine resources (threads, memory, channels) from accidental corruption by a misbehaving application program, so that resources can be used reliably for error detection, debugging, and performance monitoring; and second, to isolate application libraries from details of the underlying hardware and communications protocols, so as to minimize the impact of evolutionary changes in those areas. The actual application runs with user privileges and invokes the kernel for input/output and inter-node communication.

The instruction-level simulator is architecturally accurate, executing both kernel and application code. It models all the features of a node and the communication between nodes. Each instance (process) of the simulator models one node, and multiple instances can be used to simulate a system with multiple nodes. Internally, the multiple simulator instances communicate through MPI. As a result of executing an application, the instruction-level simulator produces detailed traces of all instructions executed. It also produces histograms of the instruction mix. One trace and one histogram is produced for every node simulated.

The instruction-level simulator does not have timing information and, therefore, it does not directly produce performance estimates. Instead, we use the traces produced by the simulator to feed two other

performance tools. One of these tools is a cache simulator and event visualizer that provides measurements of cache behavior. The other tool is a trace analyzer that detects dependences and conflicts in the instruction trace, producing an estimate of actual machine cycles necessary to execute the code. The trace analyzer does not execute the instructions, but it models the resource usage of the instructions. In the trace analyzer, each instruction has a pre-defined latency to execute, and instructions compete for shared resources. The threads in a node execute instructions in program order. Thus, if resources for an instruction are not available when the thread tries to issue that instruction, the issue is delayed, and the thread stalls until all resources become available. For memory operations, the latency of the operations depend on how deep in the memory hierarchy we have to go to fetch the result. Our architecture provides one level of data cache plus memory. The trace analyzer uses a probabilistic cache model, with a 90% hit rate. This value was validated by running the trace through the cache simulator.

Simulating a Petaflop machine is no easy task. Straightforward simulation of the entire machine would require 32,768 simulation processes, one for each simulated node. However, the molecular dynamics application has a structure that allows us to simulate completely only one node in the entire machine and extrapolate the performance results for the machine. As presented previously, the molecular dynamics code runs on a two-dimensional logical mesh of nodes. Each node simulates atomic interactions between two sets of atoms: a target set and a source set. Because of the particular decomposition method used, communication between nodes occurs intra-row and intra-column only. Thus, the simulation of one row provides information on the behavior of all other rows in the logical two-dimensional mesh. Similarly, the simulation of one column provides information on the behavior of all other columns in the logical two-dimensional mesh. This reduces the number of nodes that we need to simulate to 383 (256 in one row J plus 128 in one column I). This is illustrated in Figure 3. We still need to provide correct values for incoming messages at the boundary of the subsystem we simulate. To do so, we modified our communication layer to "fake" all communication except that between nodes in row J or column I. With a balanced work distribution, node (I, J) performs a similar set of operations to that performed by all other nodes in the system. Therefore, the performance of node (I, J) can be extrapolated to obtain the performance of the entire system. We use the instruction level simulator to simulate all nodes in column I and row J, but we collect and analyze trace information only for node (I, J).

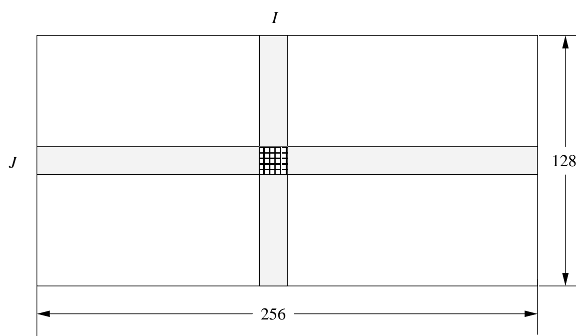

Figure 3: Strategy for performance estimation by simulating only one row and column of the logical mesh.

5 Experimental Results

As a test case for our code, we assembled a molecular system with the human carbonic anhydrase (HCA) enzyme [7], which plays an important role in the disease glaucoma. The HCA enzyme was solvated in 9000

water molecules, for a total of 32,000 atoms in the system. We used our molecular dynamics code, running on the simulator described above, to compute the evolution of this system. The starting experimental coordinates are taken from the NMR structure of carbonic anhydrase, as described in [7]. The molecular system was prepared by taking the crystallographic configuration of the HCA enzyme (Protein Data Bank identification label 1AM6 – `http://www.rcsb.org/pdb/`) and solvating it in a box of water of size ~ 70Å. The CHARMM22 [2] force field was used to treat the interactions between the atoms in this protein/water system. Newtonian dynamics was generated under standard conditions, room temperature and 1 atm pressure, through the use of the standard Verlet algorithm solver [12].

Table 1 lists the values for the various parameters used in the analytical performance model in Section 3. Table 2 lists the number of instructions required for each force computation. Some forces require the use of critical sections. The time to acquire and release the critical sections is shown in Table 2. This, together with the counts in Table 1, provide the information needed for estimating performance from the analytical performance model in Section 3. Table 3(a) summarizes the values of the various primitive communications parameters. They are obtained from intrinsic characteristics of the architecture and application. Table 3(b) contains the resulting values for the derived parameters, obtained from the equations above.

Table 1: Number of forces computed by each node in one time step of the simulation of the HCA protein, with a 14 Å long real cutoff and a 7 Å short cutoff, using a force decomposition method with a 1 by 1 (serial problem) and a 256 (columns) by 128 (rows) logical meshes.

	1×1	current allocation	
		256×128	scaling factor
target atoms	31,747	249	128
electrostatic forces (real part, long cutoff)	36,157,720	1,297	27,878
Lennard-Jones forces (long cutoff)	36,241,834	1,313	27,602
electrostatic forces (real part, short cutoff)	4,430,701	127	34,887
Lennard-Jones forces (short cutoff)	4,440,206	127	34,962
k-factors	8,139	23	354
sines and cosines in k-space	12,652,290	494	25,612
bonds	22,592	259	87
bends	16,753	456	37
torsions	11,835	748	16

Table 4 summarizes total instruction counts per node for various thread configurations, as obtained from the architectural simulator. These configurations differ in the number of threads that are allocated to perform computations. The number of floating-point units (FPUs) that are allocated to the computation is $\left\lceil \frac{N_{\text{thread}}}{8} \right\rceil$. In addition, a fraction of the threads is allocated to perform communication and other system services. The table lists the total number of loads, stores, branches, integer instructions, logical instructions, system calls, and floating-point instructions executed by all threads on a node. In the case of floating-point instructions we detail the number of multiply-add (FMAD) and multiply-subtract (FMSD) instructions. Each of those instructions performs two floating-point operations. The total number of floating-point operations is shown in row "Flops" and the total number of instructions in row "Total". We note that, as expected, the number of floating-point instructions does not change significantly with the number of threads. On the other hand, the number of loads and branches does increase significantly with the number of threads, as the threads spend more time waiting for barriers and locks.

Table 5 summarizes our results for all the configurations (different numbers of computational threads)

Table 2: Measured parameters for the computation of various forces.

parameter	instructions			description
	force	locks	total	
T_{move}	50		50	compute new position and velocity of an atom
T_{bond}	50	60	110	computation of a bond force (2 atoms)
T_{bend}	250	90	340	computation of a bend force (3 atoms)
T_{torsion}	600	120	720	computation of a torsion force (4 atoms)
T_{LJ}	200	30	230	computation of a Lennard-Jones force
T_{ES}	600	30	630	computation of real-part electrostatic force
T_{eta}	2,000		2,000	computation of one atom contribution to k-factor
T_{redux}	500		500	computation of one reduction step for k-factor
$T_{k\text{-factor}}$	1,000		1,000	final stage of computing k-factors
$T_{k\text{-force}}$	3,000		3,000	computation of Fourier-space electrostatic force

we tested. For each configuration we list the number of instructions/thread for each short step, intermediate step, and long step (See Figure 1). We show the number of instructions/thread for the k-factor and k-force components of a long step. We also show the total number of instructions/thread and computation cycles per time step, as determined by the architectural simulator and trace analyzer. The CPI (clocks per instruction) is computed as the ratio of those two last numbers. The CPF (clocks per floating-point instruction) is a measure of the average number of clocks per floating-point instruction, from the perspective of the floating-point units. We compute it as

$$\text{CPF} = \frac{N_{\text{cycles}} \times \lceil N_{\text{thread}}/8 \rceil}{N_{\text{float}}}, \tag{19}$$

where N_{float} is the total number of floating-point instructions, N_{cycles} is the number of computation cycles, and $\lceil N_{\text{thread}}/8 \rceil$ is the number of floating-point units utilized. (8 threads share one floating-point unit.)

The number of machine cycles for inter-node communication in Table 5 is obtained from Equation (18), and it is independent of the number of threads. The total number of cycles per time step is obtained by adding computation and communication cycles. We compute the multithreaded speedup as the ratio of total cycles for single- and multi-threaded execution. Finally, the efficiency is computed as the ratio between speedup (relative to single-threaded execution) and number of threads. The low CPI/high CPF numbers for one thread indicates good thread unit utilization, but low floating-point unit utilization. The other configurations have eight active threads per floating-point unit and perform roughly two times more work, per floating-point unit, than a single thread does.

The curves in Figure 4 indicate the number of instructions/thread derived from the analytical performance model for the execution of different components of the molecular dynamics application. The markers in the figure indicate measurements made on the simulator. The values plotted are *accumulated*, that is, the value of each component is added to the previous components. This figure shows the contribution of the short, intermediate, reciprocal space (k-space) and finally the long range (real part) electrostatic and Lennard-Jones forces to the total computational time step. There is a very good fit between analysis and simulation, indicating that the analytical models do indeed capture the behavior of the application.

The time charts at the bottom of Figure 1 summarize the computational performance of the molecular dynamics application. The first two time lines show the number of instructions/thread for a 250-thread execution. The entire computational time step takes approximately 39,000 instructions/thread. Each short step takes 1,700 instructions. The intermediate step takes 5,000 instructions. The long step takes 19,000 instruc-

Table 3: Summary of communication cost parameters.

(a) primitive parameters

parameter	value	description
N_{source}	125	number of source atoms assigned to a column
N_{target}	249	number of target atoms assigned to a row
$N_{k\text{-factor}}$	23	number of k-factors assigned to a column
N_{put}	13	number of puts to nearest neighbor
T_{triplet}	12	machine cycles to transfer 24 bytes through the interconnect
T_{complex}	8	machine cycles to transfer 16 bytes through the interconnect
T_{hop}	6	machine cycles to cross one node in the cellular interconnect
T_{add}	8	machine cycles to complete one floating-point addition
$\max_{\forall i}(N^i_{\text{hop_c}})$	20	maximum number of hops inside a column
$\max_{\forall i}(N^i_{\text{add_c}})$	28	maximum number of adds inside a column
$\max_{\forall i}(N^i_{\text{hop_r}})$	48	maximum number of hops inside a row
$\max_{\forall i}(N^i_{\text{add_r}})$	38	maximum number of adds inside a row

(b) derived parameters

parameter	machine cycles	description
$T^{\text{latency}}_{\text{positions}}$	240	latency to broadcast first position
$T^{\text{latency}}_{k\text{-factors redux}}$	464	latency to reduce first k-factor
$T^{\text{latency}}_{\text{forces}}$	880	latency to reduce first force
$T^{\text{latency}}_{\text{put}}$	30	latency to memory of nearest neighbor
$T_{\text{positions}}$	1,740	total time to broadcast positions
$T_{k\text{-factors redux}}$	648	total time to reduce k-factors
T_{forces}	3,838	total time to reduce forces
T_{put}	186	total time to put positions in nearest neighbor
T^m_i	5,608	communication time for intermediate step
T^m_l	6,256	communication time for long step
T^m_s	186	communication time for short step
T^m	13,352	upper bound on exposed communication time per time step

tions. The next time line summarizes communication behavior, as obtained from the simulator, showing the total number of bytes transmitted by the simulated node along each of the axes of the physical three-dimensional mesh. We note that much more data is transmitted along the X and Z axes than along the Y axis. This is a result of our particular embedding of the two-dimensional logical mesh into the three-dimensional physical mesh. The bottom line shows the total number of cycles and the estimated time for one iteration step.

Results from the cache simulator are shown in Figure 5. We plot the average cache miss rate for different cache sizes and set associativity values. The results show that our probabilistic model is valid for 4- and 8-way set-associative caches of size 8 kbytes, and for any associativity with a 16-kbyte or larger cache.

The results of this section indicate that a molecular dynamic simulation for 32,000 atoms can be run on a full Petaflop cellular machine with good performance. It is possible to exploit 32,768 nodes, each with 250 threads or more, and run a full computational time step (Figure 1) in 375 μs. This code runs at 0.87 Petaop/s, and 0.14 Petaflop/s. However, we should state that the results presented here are only

Table 4: Instruction counts for the sample node (I, J).

instruction class	1 thread	50 threads	100 threads	150 threads	180 threads	200 threads	250 threads
Loads	932,617	1,380,652	2,013,751	2,367,753	3,000,583	3,253,284	3,277,406
Stores	314,545	350,290	386,740	423,190	445,060	459,640	495,778
Branches	392,998	810,678	1,412,827	1,735,879	2,350,139	2,590,460	2,584,124
Integer ops	779,227	815,525	852,525	889,452	911,562	926,302	962,535
Logical ops	910,489	1,006,189	1,080,147	1,164,083	1,208,168	1,247,712	1,334,017
System	46,835	51,686	56,636	61,586	64,556	66,536	71,486
Floating-point ops	1,206,253	1,211,545	1,216,945	1,222,345	1,225,585	1,227,745	1,232,891
FMAD	288,217	288,805	289,405	290,005	290,365	290,605	291,169
FMSD	42,821	42,821	42,821	42,821	42,821	42,821	42,821
FLOPS	1,537,291	1,543,171	1,549,171	1,555,171	1,558,771	1,561,171	1,566,881
Total	4,582,964	5,626,565	7,019,571	7,864,288	9,205,653	9,771,679	9,958,237
% float instructions	26%	22%	17%	15%	13%	13%	12%

Table 5: Instruction and cycle counts for the sample node (I, J).

	1 thread	50 threads	100 threads	150 threads	180 threads	200 threads	250 threads
short	182,947	4,712	2,978	2,161	2,118	2,099	1,696
intermediate	656,383	15,191	9,226	6,533	6,409	5,683	4,703
long	2,460,789	56,954	34,867	26,061	25,221	23,713	18,791
k-factor	332,654	11,704	9,368	9,002	8,625	8,608	7,826
k-force	661,631	14,024	8,491	5,738	5,738	5,738	2,987
instructions/thread	4,582,740	111,975	69,585	51,790	50,502	48,228	39,162
computation cycles	10,847,196	607,530	346,797	249,002	228,668	214,153	173,896
CPI	2.37	5.41	4.99	4.80	4.53	4.44	4.44
CPF	8.99	3.51	3.70	3.87	4.29	4.36	4.51
inter-node communication cycles	13,352	13,352	13,352	13,352	13,352	13,352	13,352
total cycles	10,860,548	620,882	360,149	262,354	242,020	227,505	187,248
speedup	1	17.5	30.2	41.4	44.9	47.7	58.0
efficiency	1.00	0.35	0.30	0.28	0.25	0.24	0.23

approximate, as the simulator is not a detailed cycle level simulator. This was not feasible, both because of the slow performance of such a simulator and because of the lack of detailed logic design for our machine. Rather, the trace analyzer uses estimated information on the depth of various pipelines and uses a queuing model for congestion at key shared resources.

We expect that changes in software, algorithms, and mathematical methods will significantly further improve the performance of the code we simulated. On the other hand, we can also expect many surprises and challenges as we proceed from simulations to actual system.

6 Conclusions

We have estimated the execution of a molecular dynamic code for a system of 32,000 atoms on a full Petaflop cellular system, on the scale envisioned by IBM's Blue Gene project. A sequential version of the application executed at 140 s/time step in an 800 Mflop/s workstation. The parallel version executed at 375μs/time step in our Petaflop machine. This corresponds to a parallel speedup of 368,000 in a machine 1,250,000 times

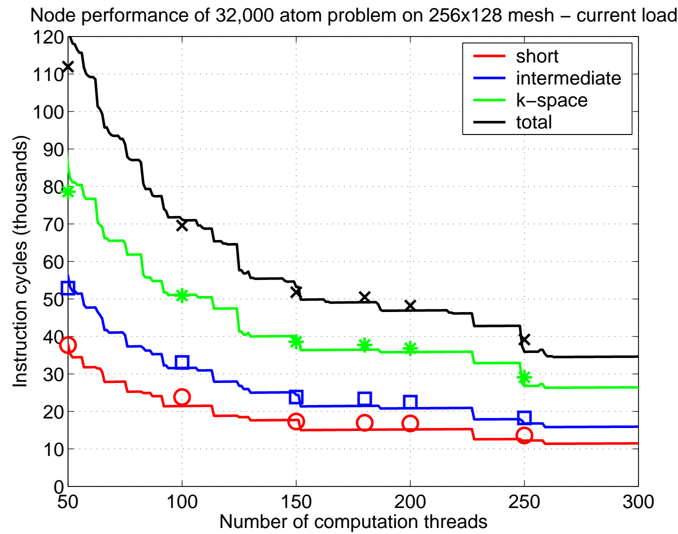

Figure 4: Performance of components of molecular dynamics code. The solid lines represent values from the analytical model. The markers are measurements from simulation.

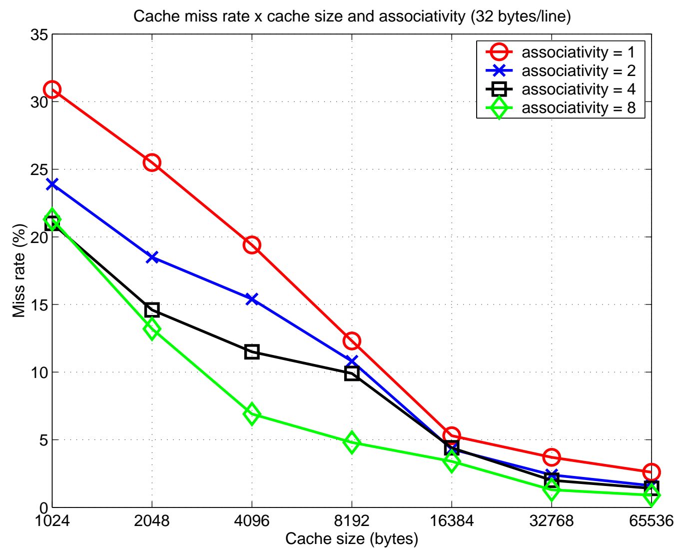

Figure 5: Cache simulation results for the traces collected from executing our molecular dynamics code.

faster, for an efficiency of 30%. This result is in agreement with the estimates derived in [8].

This exercise demonstrates that this class of molecular dynamics applications has enough parallelism to exploit millions of concurrent threads of execution with reasonable efficiency. It demonstrates, in broad lines, the validity of a massively parallel cellular system design as one approach to achieving 1 Petaflop of computing power. It also provides us with a clear understanding of a representative molecular dynamic application code, for which we now have an accurate performance model.

We still have much work to do to refine and improve the results presented here: the simulators have to be upgraded to represent a detailed hardware design; the performance models need to be upgraded and validated against cycle faithful simulations; the communication analysis needs to reflect overlap between computation

and communication; node failures in a machine of this scale have to be addressed; and algorithms and methods will continue to be improved.

References

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* Oxford Science Publications, Oxford, UK, 1987.

[2] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy minimization, and dynamics calculations. *J. Comput. Chem.*, 4:187–217, 1983.

[3] P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Ann. Phys.*, 64:253–287, 1921.

[4] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, October 1999.

[5] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. In *Proceedings of IEEE Micro*, April 1997.

[6] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comp. Phys.*, 117:1–19, 1995.

[7] L. R. Scolnick, A. M. Clements, J. Liao, L. Crenshaw, M. Hellberg, J. May, T. R. Dean, and D. W. Christianson. Novel binding mode of hydroxamate inhibitors to human carbonic anhydrase II. *J. Am. Chem. Soc.*, 119:850–851, 1997.

[8] V. E. Taylor, R. Stevens, and K. Arnold. Parallel molecular dynamics: Implications for massively parallel machines. *Journal on Parallel and Distributed Computing*, 45(2):166–175, September 1997.

[9] J. Torrellas, L. Yang, and A.-T. Nguyen. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[10] M. E. Tuckerman and B. J. Berne. Molecular dynamics in systems with multiple time scales. *J. Comp. Chem.*, 95:8362–8364, May 1992.

[11] M. E. Tuckerman, B. J. Berne, and G. J. Martyna. Reversible multiple time scale molecular dynamics. *J. Chem. Phys.,*, 97:1990–2001, 1992.

[12] L. Verlet. Computer experiments on classical fluids. I. thermodynamical properties of Lennard-Jones molecules. *Phys. Rev.*, 159:98–103, 1967.