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

almasi,cascaval,castanos,denneau,wdonath,mariae,giampapa,ctho,
lieber,jmoreira,dennisn,snir,hankw@us.ibm.com

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

ABSTRACT

The IBM Blue Gene project has endeavored to develop a cellular architecture computer with millions of concurrent threads of execution. One of the major challenges of this project is demonstrating that applications can successfully exploit this massive amount of parallelism. 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 the Blue Gene cellular architecture. We perform both analytical and simulation studies of the behavior of this application when executed on a very large number of threads. As a result, we demonstrate that this class of applications can execute efficiently on a large cellular machine.

Keywords

Massively parallel computing, molecular dynamics, performance evaluation, cellular architecture, Blue Gene

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 [10] propose the use of thousands of very high speed processors (hundreds of gigahertz). On the other hand, projects like IBM's Blue Gene [2] 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.

Massively parallel 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. Combined logic-memory microelectronics 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, and can be used as building blocks for cellular machines. Examples include the Illinois FlexRAM [5, 12] and Berkeley IRAM [7] projects. Nevertheless, because of sheer size, building a cellular machine to deliver a Petaflop/s – or 10^{15} floating-point operations per second – is quite a challenge. This enterprise 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 application. We analyze the behavior of this application on a cellular architecture with millions of concurrent threads of execution, which is a design for IBM's Blue Gene project. This is one example of a class of applications that can indeed exploit the multiple levels of massive parallelism offered by a Petaflop-scale cellular machine, as discussed in [11]. This parallel application was derived from the serial version of a molecular dynamics code developed at the University of Pennsylvania [6]. The application was rewritten with new partitioning techniques to take advantage of multiple levels of parallelism.

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, using the sequential version of the code, can be solved on a Power 3 workstation with peak performance of 800 Mflop/s in 140 seconds per time step. 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 considered, at the level necessary to understand how the 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 infrastructure in deriving experimental performance results which are presented, together with the results from the analytical model, in Section 5. Section 6 discusses some of the changes to the Blue Gene architecture that were motivated by this work. Finally, Section 7 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 the architecture of Blue Gene is that performance is obtained by exploiting massive amounts of parallelism, rather than the very fast execution of any particular thread of control. This premise has significant technological and architectural impacts. First, individual processors are kept simple, to facilitate design and large scale replication in a single silicon chip. Instead of spending transistors and watts to enhance single-thread performance, the real estate and power budgets are used to add more processors. The architecture can also be characterized as *memory centric*. Enough threads of execution are provided in order to fully consume all the memory bandwidth while tolerating the latency of individual load/store operations. Because a single silicon chip consists of many replicated units, it is naturally fault tolerant. The replication approach also improves the yield of fabrication, even when large chips are used, thus reducing cost. Moreover it allows the machine to gracefully degrade but continue to operate when individual units fail.

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. 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 strictly in order. Each group of 8 threads shares one data cache and one floating-point unit. The data cache configuration for Blue Gene is 16 KB, 8-way set associative. We call this group of threads, floating-point unit, and cache, a *processor*. Such simplifications are necessary to keep instruction units simple, resulting in large numbers of them on a die. The floating-point units (32 in a node) are pipelined and can complete 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 GB/s per channel on each direction is achieved. Nodes communicate by streaming data through these channels.

Larger systems are built by interconnecting multiple nodes in a regular pattern. 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. (See Figure 1)

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 messages. 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.

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

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

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. The notation $\{\vec{x}_j(t)\}$ represents the set of positions of all atoms at time t. 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. In the particular molecular dynamics approach we chose, the system to be simulated is shaped in the form of a box, which is replicated indefinitely in all three dimensions, giving rise to a periodic system. This is illustrated in two dimensions in Figure 2.

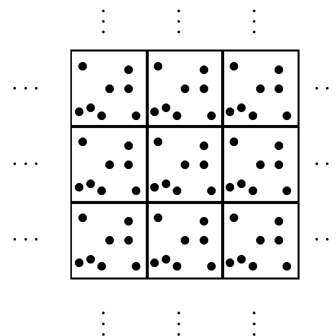

Figure 2: The molecular system we simulate has a periodic structure. In principle, we need to consider the interactions among an infinite number of atoms.

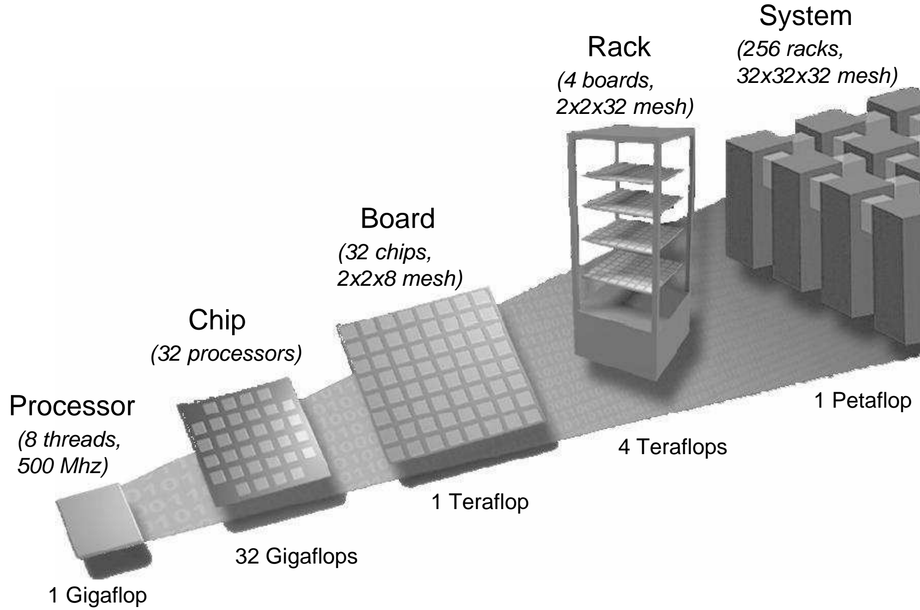

Figure 1: Blue Gene's cellular architecture hierarchy. The entire system consists of 32,768 nodes (1 million processors) connected in a three-dimensional mesh.

3.1 A molecular dynamics algorithm

The force $\vec{F}_i(t) = \vec{F}_i(\{\vec{x}_j(t)\})$ 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 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. In the model we use, there are three different types of intra-molecular forces: *bonds*, *bends*, and *torsions*. 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. We note that the sphere for atom i may include atoms in a neighboring (replicated) box.

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 [4]. In this method, the computation of electrostatic forces is divided into two fast converging sums: a real space part and a reciprocal space, also called *k-space*, part:

$$\vec{F}_i^e(t) = \vec{F}_i^r(t) + \vec{F}_i^k(t). \quad (2)$$

For the real space part ($\vec{F}_i^r(t)$), interactions are computed between an atom i and all the atoms inside a sphere centered at atom i, as for Lennard-Jones forces (although the cutoff radius could be different). The pairs of atoms whose distance is within these cutoff radii are commonly stored in a linked list, also known as the *Verlet* list. The generation of this list is usually performed once every several time-steps because atoms move slowly. The frequency of generation is an optimization problem. Since its computational cost is amortized over several time-steps, we do not further discuss generation of Verlet lists in this paper.

For the computation in reciprocal space ($\vec{F}_i^k(t)$), 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

$$\eta_{\vec{k}} = \sum_{i=0}^{n-1} q_i e^{j\vec{k}\cdot\vec{x}_i} = \sum_{i=0}^{n-1} q_i(\cos(\vec{k}\cdot\vec{x}_i) + j\sin(\vec{k}\cdot\vec{x}_i)), \quad (3)$$

where n is the number of atoms, q_i and $\vec{x}_i$ are the charge and position of atom i, respectively. The $\eta_{\vec{k}}$ terms are also called *k-factors* in this paper. 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 $\vec{k}$:

$$\vec{F}_i^k = \sum_{\forall \vec{k} \in K} \vec{f}^k(q_i, \eta_{\vec{k}}). \quad (4)$$

The $\vec{f}^k$ forces are also called *k-forces* in this paper. The exact formula for function $\vec{f}^k(\cdot)$ is not important for structuring the parallelization. (The precise expression is specified in [1]). We note that for each atom we need to perform a summation over all $\vec{k} \in K$, and that the value of any $\eta_{\vec{k}}$ depends on the position of all atoms in the system.

3.1.1 Inter-node parallelization

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 [8], and a new parallel formulation for the k-space decomposition. 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, we 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. With this configuration, node $n_{I,J}$ (row I and column J of the logical mesh) is assigned source set S_J and target set T_I, as shown in Figure 3.

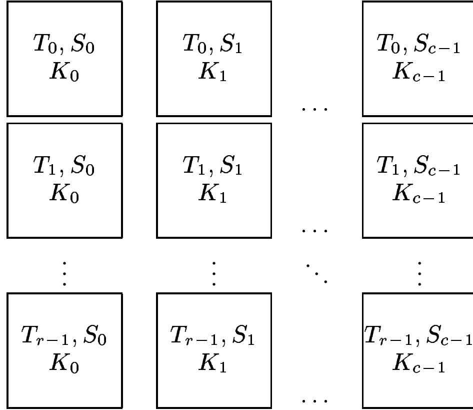

Figure 3: Force decomposition adopted for the molecular dynamics application.Each target set T_I is replicated along all nodes in row I. Each source set S_J and reciprocal space vector set K_J is replicated along all nodes in column J.

Using the decomposition described above, node $n_{I,J}$ can compute locally and with no communication the Lennard-Jones and real space electrostatic forces exerted by atoms in source set S_J over atoms in target set T_I. As previously mentioned, only the interactions between atoms that are closer than a certain cutoff radius are actually computed. In other words, let $\vec{F}^{LJ}_{l,m}$ be the Lennard-Jones force between atoms l and m, and let $\vec{F}^{r}_{l,m}$ be the real part of the electrostatic force between atoms l and m. Then, node $n_{I,J}$ computes $\vec{F}^{LJ}_{l,m}$ and $\vec{F}^{r}_{l,m}$, $\forall l \in T_I, \forall m \in S_J$, such that the distance between atoms l and m is less than the cutoff radius for Lennard-Jones and electrostatic forces, respectively. Each node $n_{I,J}$ also computes the partial vector sum of the Lennard-Jones and real part electrostatic forces that act upon the atoms in its target set:

$$\vec{F}^{LJ}_{l,J} = \sum_{\forall m \in S_J} \vec{F}^{LJ}_{l,m}, \ \forall l \in T_I, \quad (5)$$

$$\vec{F}^{r}_{l,J} = \sum_{\forall m \in S_J} \vec{F}^{r}_{l,m}, \ \forall l \in T_I. \quad (6)$$

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 $\approx \left\lceil \frac{|K|}{c} \right\rceil$. We replicate each set K_J along all nodes in column J of the logical mesh. This is also shown in Figure 3. The actual computation is performed in three phases. In the *k-factors* phase, every node $n_{I,J}$ computes the contributions of the atoms in T_I to all $\eta_{\vec{k}}$ such that $\vec{k} \in K_J$. That is, node $n_{I,J}$ computes

$$\eta_{\vec{k},I} = \sum_{\forall l \in T_I} q_l e^{j \vec{k} \cdot \vec{x}_l}, \ \forall \vec{k} \in K_J. \quad (7)$$

Computing the sum of these contributions along the columns of the two-dimensional mesh constitutes the second phase. The sum is computed as an *all-reduce* operation: every node obtains the value of the sum of contributions. After this reduction, every node $n_{I,J}$ will have the value of $\eta_{\vec{k}}$ for all $\vec{k} \in K_J$:

$$\eta_{\vec{k}} = \sum_{I=0}^{r-1} \eta_{\vec{k},I}, \ \forall \vec{k} \in K_J. \quad (8)$$

The number of items (k-factors) for which a reduction is computed along column J is equal to the size of the reciprocal space vector set 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 $\vec{F}^k$ forces, described in Equation (4), for each of the atoms in set T_I. That is, node $n_{I,J}$ computes

$$\vec{F}^{k}_{l,J} = \sum_{\forall \vec{k} \in K_J} \vec{f}^{k}(q_l, \eta_{\vec{k}}), \ \forall l \in T_I. \quad (9)$$

Once all inter-molecular forces are computed, we perform an all-reduction of forces along the rows of the two-dimensional logical mesh. That is, all nodes $n_{I,J}$ in row I obtain the value

$$\vec{F}_l = \sum_{J=0}^{c-1} (\vec{F}^{r}_{l,J} + \vec{F}^{k}_{l,J} + \vec{F}^{LJ}_{l,J}), \ \forall l \in T_I. \quad (10)$$

As a result, every node in row I obtains the resulting forces over all its atoms in T_I. We note that the number of items (force vectors) for which a reduction is computed along row I is equal to the size (number of atoms) of target set T_I.

The computation of intra-molecular forces (bonds, bends, torsions) 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 $n_{I,J}$ 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 [13, 14] to reduce the frequency of inter-molecular forces computation. We illustrate this concept in Figure 4, where a computational time step is divided into a sequence of short, intermediate and long steps. More precisely, in our configuration,

a computation time step consists of ten components: four short steps, followed by an intermediate step, followed by four short steps, followed by a long step. In a short step, only intra-molecular bonds and bends forces are computed. In an intermediate step, all intra-molecular forces (bonds, bends, and torsions) are computed, together with Lennard-Jones and real space electrostatic forces within a short cutoff radius. The computation of inter-molecular forces requires a reduction along rows. Finally, in a long step, all intra- and inter-molecular forces, including k-space electrostatic, are computed. The long step requires a reduction along columns (for the k-factors) and along rows (for the inter-molecular forces). Within steps, computation threads in the same node wait in a barrier (indicated by "B" in Figure 4) 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 4 also presents summary performance numbers, which will be discussed in Section 5.

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 5. Note that 8 logical columns are embedded in one physical plane. The broadcast of positions and the reduction of k-factors 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 $32 \times 32 \times 32$ 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.

3.1.2 *Intra-node parallelization*

Intra-node parallelism for the molecular dynamics code is exploited through multithreading. For simplicity, each set of forces (bonds, bends, torsions, Lennard-Jones, and electrostatic) to be computed in a node is statically partitioned among the threads in that node. That is, if there are N_{force} forces of a certain type to be computed, and N_{thread} compute threads in a node, then each thread computes $\leq \left\lceil \frac{N_{\text{force}}}{N_{\text{thread}}} \right\rceil$ forces. Movement of atoms is partitioned in the same way. If there are N_{atom} atoms in the target set of a node, then each thread is responsible for moving $\leq \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil$ atoms.

As each force that acts on an atom in the target set of a node is computed, it is added to a force accumulator for that atom. Each Lennard-Jones or real electrostatic force computed acts on only one target atom. Each bond, bend, and torsion (intra-molecular forces) computed however, acts on 2, 3, or 4 atoms of the target set, respectively. Before an update on a force accumulator can be performed by a thread, that thread needs to acquire a lock for the accumulator. Therefore, for each Lennard-Jones or real electrostatic force computed, the thread has to acquire one lock. For each bond, bend, or torsion computed, the thread needs to acquire 2, 3, or 4 locks, respectively.

Parallelization of the k-space computations is different, and performed in four steps inside a node. We start by first computing the contribution of each atom in the target set to all k-factors assigned to the node. This computation is statically scheduled: each thread performs $\leq \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil$ computations. To evaluate the k-factors $\eta_{\vec{k}}$, we have to sum the contribution of each atom. This is done with a binary reduction tree. There is a third step which completes the computation of the k-factors within a node, in preparation for the reduction across columns. Finally (after the column reduction is performed), the k-forces on each atom are computed. These computations are statically scheduled, and each thread computes $\leq \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil$ forces of the form $\vec{F}^k_{l,J} = \sum_{\forall \vec{k} \in K_J} \vec{f}^k(q_l, \eta_{\vec{k}})$.

3.2 A performance model for the molecular dynamics computation

Based on the previous discussion of the molecular dynamics algorithm, in this section we derive an analytical performance model for the behavior of that algorithm on a cellular architecture like IBM's Blue Gene. In Section 5, we compare results from this performance model with direct measurements from simulations. We will also use the model to evaluate the impact of certain architectural features.

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 4) can be computed as

$$T = sT_s + T_i + T_l, \tag{11}$$

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 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 during a short step. 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 as shown in Equation 12 of Figure 6.

Let N_{torsion} be the number of torsions that need to be computed by a node during an intermediate step. 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 during an intermediate step. 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 T^p_i as shown in Equation 13 of Figure 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 during a long step. Let $N_{k\text{-factor}}$ be the number of $\eta_{\vec{k}}$ terms in the node, that is, 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

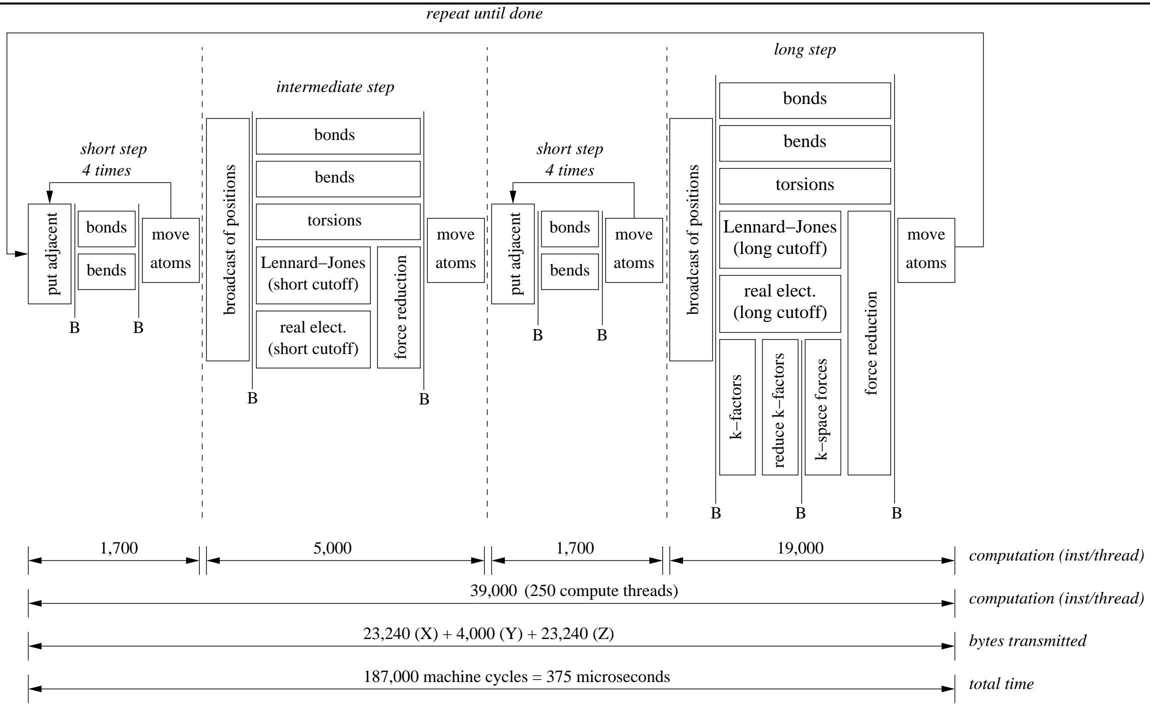

Figure 4: Flowchart for the molecular dynamics computation for a complete long time step and computational performance of the molecular dynamics application.

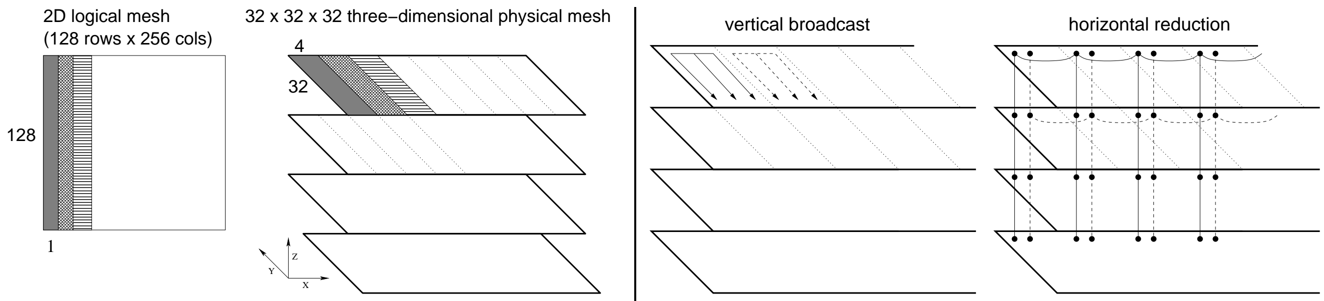

Figure 5: Mapping columns of the logical mesh to the physical machine (*left*). Communication patterns for vertical broadcast of positions and horizontal reduction of forces (*right*).

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 T_l^p as shown in Equation 14 of Figure 6.

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 binary tree, as shown in Figure 7. In each of those figures, the root of the tree is marked by a black circle. Referring to Figures 5 and 7 (*a*, *b*) , we note that each 128-node column of the logical mesh consists of four adjacent strands of 32-nodes in the physical mesh. Referring to Figures 5 and 7 (*c*, *d*), we note that each 256-node row of the logical mesh consists of eight strands of 32-nodes in the physical mesh, and those strands are 4 nodes apart. The communications of atom positions to evaluate intra-molecular forces make only a small contribution to the total communication and are always performed between neighboring 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

$$T_s^p = \left\lceil \frac{N_{\text{bond}}}{N_{\text{thread}}} \right\rceil T_{\text{bond}} + \left\lceil \frac{N_{\text{bend}}}{N_{\text{thread}}} \right\rceil T_{\text{bend}} + 2T_{\text{barrier}}(N_{\text{thread}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{move}}. \quad (12)$$

$$\begin{aligned} T_i^p = & \left\lceil \frac{N_{\text{bond}}}{N_{\text{thread}}} \right\rceil T_{\text{bond}} + \left\lceil \frac{N_{\text{bend}}}{N_{\text{thread}}} \right\rceil T_{\text{bend}} + 2T_{\text{barrier}}(N_{\text{thread}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{move}} + \\ & \left\lceil \frac{N_{\text{torsion}}}{N_{\text{thread}}} \right\rceil T_{\text{torsion}} + \left\lceil \frac{N_{\text{LJshort}}}{N_{\text{thread}}} \right\rceil T_{\text{LJ}} + \left\lceil \frac{N_{\text{ESshort}}}{N_{\text{thread}}} \right\rceil T_{\text{ES}}. \end{aligned} \quad (13)$$

$$\begin{aligned} T_l^p = & \left\lceil \frac{N_{\text{bond}}}{N_{\text{thread}}} \right\rceil T_{\text{bond}} + \left\lceil \frac{N_{\text{bend}}}{N_{\text{thread}}} \right\rceil T_{\text{bend}} + 3T_{\text{barrier}}(N_{\text{thread}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{move}} + \\ & \left\lceil \frac{N_{\text{torsion}}}{N_{\text{thread}}} \right\rceil T_{\text{torsion}} + \left\lceil \frac{N_{\text{LJlong}}}{N_{\text{thread}}} \right\rceil T_{\text{LJ}} + \left\lceil \frac{N_{\text{ESlong}}}{N_{\text{thread}}} \right\rceil T_{\text{ES}} + \\ & \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{\text{eta}}(N_{k\text{-factor}}) + \lceil \log_2(N_{\text{thread}}) \rceil T_{\text{redux}}(N_{k\text{-factor}}) + \\ & \left\lceil \frac{N_{k\text{-factor}}}{N_{\text{thread}}} \right\rceil T_{k\text{-factor}}(N_{k\text{-factor}}) + \left\lceil \frac{N_{\text{atom}}}{N_{\text{thread}}} \right\rceil T_{k\text{-force}}(N_{k\text{-factor}}). \end{aligned} \quad (14)$$

Figure 6: Analytical expressions for the computation time of short, intermediate, and long steps.

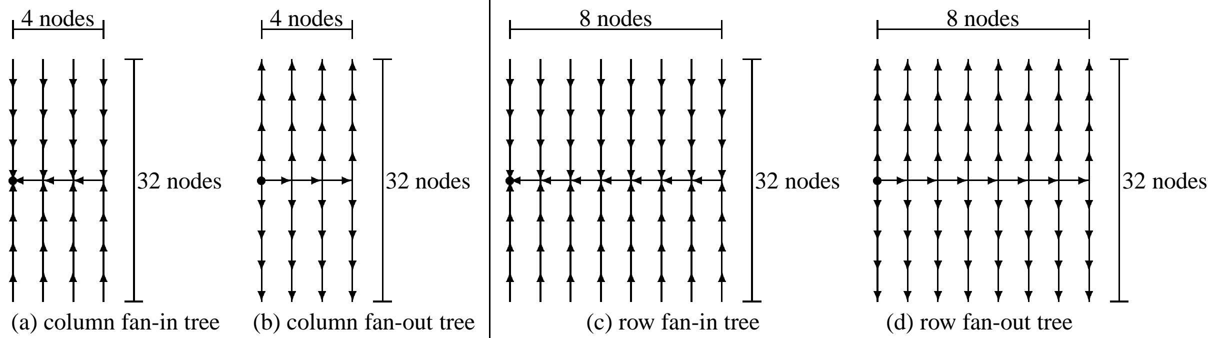

Figure 7: Fan-in and fan-out trees for reduction and broadcast operations along columns (*a* and *b*) and for reduction operations along rows (*c* and *d*).

number of elements in the vector. In equation form:

$$T_{\text{positions}} = T_{\text{positions}}^{\text{latency}} + T_{\text{triplet}} N_{\text{source}}, \quad (15)$$

$$T_{k\text{-factors redux}} = T_{k\text{-factors redux}}^{\text{latency}} + T_{\text{complex}} N_{k\text{-factor}}, \quad (16)$$

$$T_{\text{forces}} = T_{\text{forces}}^{\text{latency}} + T_{\text{triplet}} N_{\text{target}}, \quad (17)$$

$$T_{\text{put}} = T_{\text{put}}^{\text{latency}} + T_{\text{triplet}} N_{\text{put}}, \quad (18)$$

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

$$T_{\text{positions}}^{\text{latency}} = 2 \max_{\forall i}(N_{\text{hop_c}}^i) T_{\text{hop}}, \quad (19)$$

where the factor of two accounts for the round trip. Let $N_{\text{add_c}}^i$ 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

$$T_{k\text{-factors redux}}^{\text{latency}} = 2 \max_{\forall i}(N_{\text{hop_c}}^i) T_{\text{hop}} + \max_{\forall i}(N_{\text{add_c}}^i) T_{\text{add}}. \quad (20)$$

The analysis for the force reduction along the rows is similar. Let $N_{\text{hop_r}}^i$ 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 $N_{\text{add_r}}^i$ 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

$$T_{\text{forces}}^{\text{latency}} = 2 \max_{\forall i}(N_{\text{hop_r}}^i) T_{\text{hop}} + \max_{\forall i}(N_{\text{add_r}}^i) T_{\text{add}}. \quad (21)$$

We can derive worst case estimates for the exposed communications times $T_{[s,i,;]}^m$. In case there is no overlap between computation

and communication,

$$T_i^m = T_{\text{positions}} + T_{\text{forces}}, \quad (22)$$
$$T_l^m = T_{\text{positions}} + T_{k\text{-factors redux}} + T_{\text{forces}}, \quad (23)$$
$$T_s^m = T_{\text{put}}. \quad (24)$$

The upper bound on the exposed communication time is

$$T^m = 8 \times T_s^m + T_i^m + T_l^m, \quad (25)$$

where the factor of 8 comes from 8 short steps within a computational time step.

4. SIMULATION ENVIRONMENT

To complement the analytical modeling described above, we executed the molecular dynamics application on an instruction-level simulator for Blue Gene. Our machine has a proprietary instruction set, which is a typical RISC (load/store architecture with three-register instructions) model. 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 using 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, communication 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 kernel services 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 are 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 are 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 predefined execution latency, 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 trace analyzer uses a probabilistic cache model, with a 90% hit rate. This value was validated by running the trace through the cache simulator.

The performance parameters for the simulated architecture are shown in Table 1. These parameters are early estimates and may change when the low-level logic design is completed. The table is interpreted as follows: the delay for an instruction is decomposed in two parts, *execution cycles* and *latency cycles*. The execution unit is kept busy for the number of execution cycles and can not issue another instruction. The result is available after the number of execution+latency cycles, but the resources can be utilized by other instructions during the latency period. For example, a floating point add has execution of 1 cycle and latency 5 cycles. That means that the FPU is busy for 1 cycle executing this instruction (other instructions may be already in the pipeline), and the next cycle can execute another instruction. However, the result of the addition is available only after 6 (1+5) cycles. As another example, an integer division takes 33 execution cycles and 0 latency cycles. That means that it occupies exclusively the integer unit for 33 cycles and the result is available immediately after the execution period completes. Threads can issue an instruction every cycle, unless the stall on a dependence from the result of a previous instruction.

For memory operations, the latency of the operations depend how deep in the memory hierarchy we have to go to fetch the result. The memory is distributed within the chip, such that accessing the local memory is faster than accessing the memory across the chip. The memory is shared: threads within a chip can address each other's local memories. The latency of the operation depends on the physical location of the memory. Memory accesses are determined to be local or global based on the effective address. However, because the Blue Gene processors use scrambling, a global access may be in the local memory. Again, we model the scrambling probabilistically, with a $1/P$ probability that the global access is in local memory, where P is the number of processors in a chip ($P = 32$ in our configuration).

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, minus one instance of node (I, J)). This is illustrated in Figure 8. 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 set of operations similar to that performed by all other nodes in the system. Therefore, the performance of node (I, J) can be extrapolated to

Table 1: Estimates of instruction performance parameters for Blue Gene ISA. Each type of instruction is characterized by the number of cycles it keeps the execution unit busy (column "execution") and the latency for it to complete (column "latency").

instruction type	execution	latency
Branches	2	0
Integer multiply and divide	33	0
Floating point add, multiply and conversions	1	5
Floating point divide and square root	54	0
Floating point multiply-and-add	1	10
Prefetching	0	7
Memory operation (cache hit)	1	2
Memory operation (shared local)	1	20
Memory operation (shared remote)	1	27
All other operations	1	0

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).

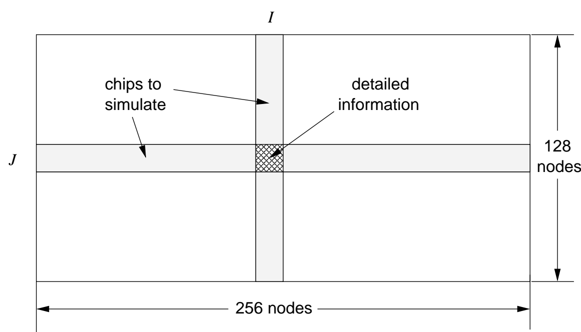

Figure 8: Strategy for performance estimation by simulating only one row and column of the logical mesh.

5. EXPERIMENTAL RESULTS

As a test case for our code, we assembled a molecular system with the human carbonic anhydrase (HCA) enzyme [9], which plays an important role in the disease glaucoma. The HCA enzyme was solvated in 9,000 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 [9]. 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 $\sim$ 70Å. The CHARMM22 [3] 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 [15].

Table 2 lists the values for the various parameters used in the analytical performance model in Section 3. The table shows the number of forces computed, per computational time step, by a serial version of the code. It also shows the number of forces computed by a single node participating in a parallel execution with a 256×128 mesh. The "scaling factor" column shows the ratio of those two values. Table 3 lists the number of instructions required for each force computation. Some forces require the use of locks, as discussed in Section 3.1.2. The number of instructions to acquire and release the locks is shown in Table 3. The number of instructions for a tree-based multi-threaded barrier, as a function of the number of threads participating in the barrier is shown in Figure 9. The data from Tables 2 and 3, and Figure 9 provide the information needed for estimating performance from the analytical performance model in Section 3. Table 4(a) summarizes the values of the various primitive communications parameters. They are obtained from intrinsic characteristics of the architecture and application. Table 4(b) contains the resulting values for the derived parameters, obtained from the equations above.

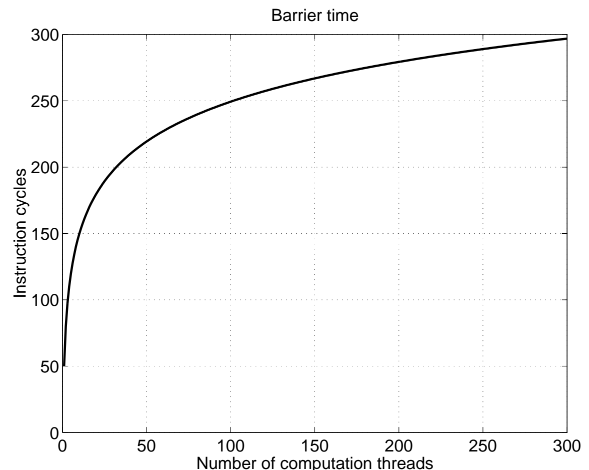

Figure 9: Cost of a barrier, as a function of the number of threads participating.

Table 5 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 $\left\lceil \frac{N_{\text{thread}}}{8} \right\rceil$. Additional threads, not included in those numbers, are 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-

Table 2: 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. We show results for a single node execution and for a 256 (columns) × 128 (rows) logical mesh.

	single node	one node in 256 × 128 mesh	
	# of items	# of items	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 3: Measured parameters for the computation of various forces.

parameter	# of 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

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 6 identifies the major sources of overhead. We note that most of the additional instructions executed in multithreaded belong to two operations, barrier and lock. The additional lock instructions represent contention for locks, that increase with a larger number of threads. The increase in number of instructions spent in barriers has two sources. First, as the number of threads increases, the total time through a barrier increases. This is illustrated in Figure 9. In addition, a larger number of threads typically makes load balancing more difficult, and the faster threads have to wait more time in the barrier for the slower threads. The "total overhead" row in Table 6 is equal to the difference between the number of instructions for multithreaded and single-threaded execution. The effect on overall performance achieved if the barrier overhead and/or the lock overhead is avoided is discussed in Section 6.

Table 7 summarizes additional performance results for all the configurations (different numbers of computational threads) we tested. For each configuration we list the number of instructions/thread for each short step, intermediate step, and long step (See Figure 4). 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

$$\text{CPF} = \frac{N_{\text{cycles}} \times \lceil N_{\text{thread}}/8 \rceil}{N_{\text{fbat}}}, \tag{26}$$

where N_{fbat} 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 7 is obtained from Equation (25), 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.

Table 4: 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

The curves in Figures 10 and 11 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 figures indicate measurements made on the simulator. The values plotted are *accumulated*, that is, the value of each component is added to the previous components. Therefore, the top curve of Figure 10 also represents the total number of instruction cycles for a computational time step. The top curve of Figure 11 represents the total number of instruction cycle for k-space computation. Figure 10 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. Figure 11 shows the contribution of k-factor and k-force components to the total k-space computation. 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 4 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 instructions. 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 12. We plot the average cache miss rate for different cache sizes and set associativity values, for a configuration with 250 compute threads. The results show that our probabilistic model (90% data cache hit rate) is valid for 4- and 8-way set-associative caches of size 8 KB, and for any associativity with a 16 KB 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 4) in 375 μs (187,000 cycles at 2 ns/cycle). This code runs at 0.87 Petaop/s, and 0.14 Petaflop/s. However, we should state that the results presented here are only 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

Table 5: 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 6: Overhead instructions for sample node (I, J).

function	1 thread	50 threads	100 threads	150 threads	180 threads	200 threads	250 threads
barrier		525,330	1,480,916	1,903,394	3,028,426	3,408,496	3,211,434
lock	52,874	190,280	274,926	347,642	363,564	391,848	446,194
barrier+lock	52,874	715,610	1,755,842	2,251,036	3,391,990	3,800,344	3,657,628
total overhead	0	1,043,601	2,436,607	3,281,324	4,622,689	5,188,715	5,375,273

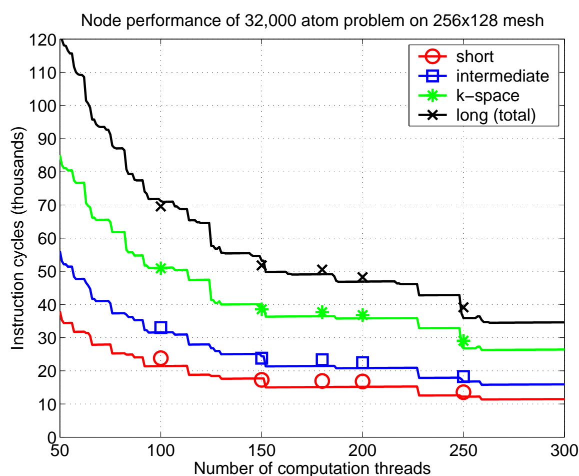

Figure 10: Performance of components of molecular dynamics code. The solid lines represent values from the analytical model. The markers are measurements from simulation.

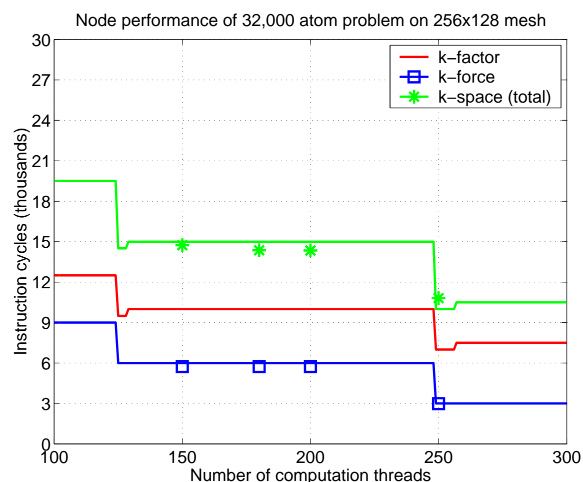

Figure 11: Performance of components of k-space computation. The solid lines represent values from the analytical model. The markers are measurements from simulation.

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. ARCHITECTURAL IMPACT

Since we completed the work described in this paper, the Blue Gene architecture has evolved. Some changes were motivated by silicon real estate constraints. For example, the number of threads per floating-point unit was reduced from 8 to 4, for a total of 128 threads/node. The memory was also reduced from 16 MB to 8 MB/node. Other important changes, however, were motivated by the results from our performance evaluation work. We discuss some of those changes.

Table 3 and Figure 9 show the direct cost of locks and barriers, respectively. To evaluate the impact of those in the final performance of the molecular dynamics code, we measured what would be the performance if the code were free of locks and barriers. Obviously, such code would not execute correctly. The results are shown in Figure 13. It shows that for a large number of threads (250), the instruction count could be reduced by up to 10,000, or 25%, if there were no locks or barriers. As a result, the Blue Gene architecture now includes fast hardware barriers and atomic mem-

Table 7: **Instruction and cycle counts for one compute thread in sample node (I, J). CPI is the average number of machine cycles per instruction. CPF is the number of machine cycles per floating-point instruction.**

	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

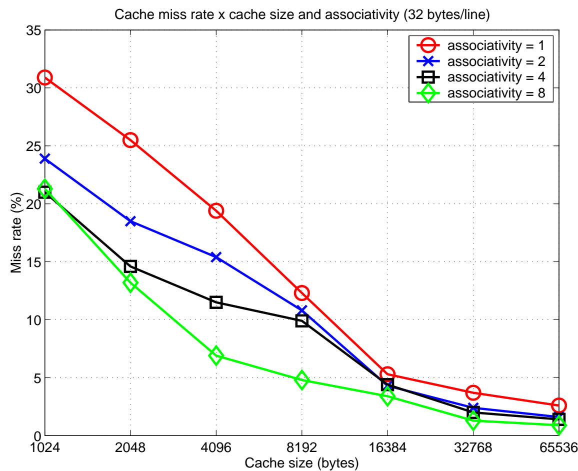

Figure 12: Data cache simulation results for the traces collected from executing our molecular dynamics code.

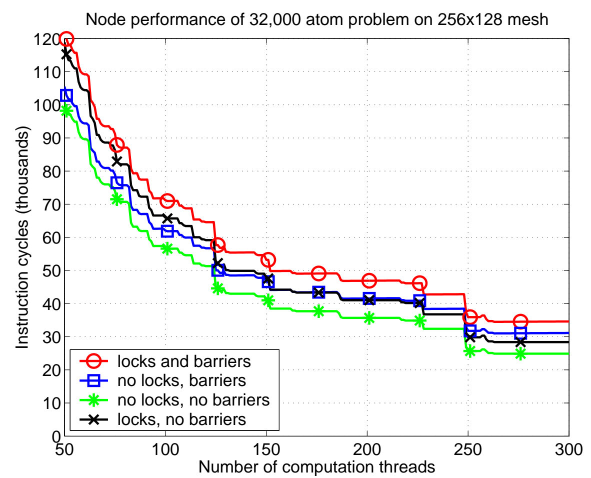

Figure 13: Impact of barriers and locks in the performance of the molecular dynamics code. Different lines represent different configurations of locks and barriers.

ory updates. The hardware barriers allow synchronization of all threads in a node in less than 10 machine cycles.

Another architectural enhancement motivated by our work was the *B-switch*. As shown in Table 7, the overall impact of communication cycles is small, less than 10% of the total. However, to achieve that goal, it is necessary to perform the reductions and broadcasts with a minimal software overhead. That motivated the development of the B-switch: a microprogrammed data streaming switch that operates at double word (64-bit) level. At each cycle, the B-switch can route data from any input to any combination of outputs. It can also perform floating-point operations on the streams and direct streams of data to/from the memory in the node. A diagram of the B-switch is shown in Figure 14.

7. 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 $375\mu s$/time step in our Petaflop machine. This corresponds to a parallel speedup of 368,000 in a machine 1,250,000 times faster, for an efficiency of 30%. This result is in agreement with the estimates derived in [11].

Our exercise demonstrates that a class of molecular dynamics applications exhibits 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 one 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

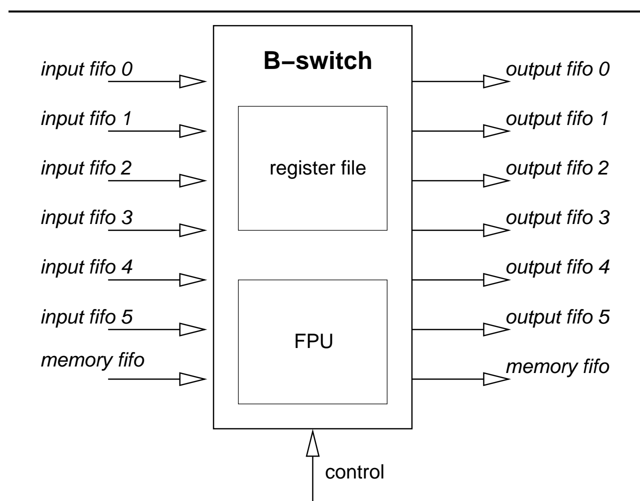

Figure 14: The B-switch is a device for streaming data from the input ports (and memory) to the output ports (and memory). The B-switch incorporates an FPU to perform arithmetic and logic operations directly on the data stream.

presented here: the simulators have to be upgraded to represent a more 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.

8. REFERENCES

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

[2] The Blue Gene project. `http://www.research.ibm.com/bluegene`.

[3] 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.

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

[5] 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.

[6] P. B. Moore. CM3D CMM MD Code. `http://www.cmm.upenn.edu/~moore/code/code.html`.

[7] 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.

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

[9] 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.

[10] T. Sterling et al. Hybrid technology multithreaded. `http://htmt.jpl.nasa.gov`.

[11] 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.

[12] 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.

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

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

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