Petaflop Computing for Protein Folding

Shannon K. Kuntz, Richard C. Murphy, Michael T. Niemier, Jesus Izaguirre, and Peter M. Kogge

1 Introduction

Protein Folding is considered one of today's most significant "grand challenge" problems and, together with other computational chemistry problems, has been one of the driving forces for the development of bigger, better, faster supercomputers. IBM is currently proposing to build the "Blue Gene" [5], a petaflop computer to tackle the protein folding problem, while Silicon Graphics has been continually working to produce more powerful, multiprocessor systems for researchers working in this area [8]. Protein folding has significance in numerous areas of health care including better understanding of diseases and the development of drugs to combat them. However, the size, complexity, and time scale of the three-dimensional structures in protein folding make simulation extremely memory and computationally intensive. Therefore, larger more powerful machines are necessary to enable the larger, more accurate simulations needed to further understand this complex process.

This paper addresses the mapping of protein folding to a million-processor array for petaflop performance. A new scalability model is defined which incorporates the characteristics of highly-distributed multithreaded architectures. The model is used to compare the scalability of various algorithms and decompositions and facilitates the prototyping of a million node Processing-In-Memory array to achieve petaflop performance during a molecular dynamics simulation typical of protein folding. In addition, simulations are used to validate the model and parallel implementations are demonstrated. Although parallel implementations of MD simulations exist, none have been attempted at the scale that we are proposing and more accurate algorithmic complexity models are needed to assist in such attempts.

2 The Core of Protein Folding

Protein folding is used to describe the interactions and motions of proteins due to the forces around them. Protein folding is estimated to take 10's of microseconds and, due to its complexity, cannot be fully simulated with the current computing technology. Molecular dynamics solves the classical equations of motion for a system of N atoms interacting according to a potential energy force field and, as such, can provide a good estimate of protein folding.

In a molecular dynamics simulation there are two main components of each time step, calculating forces and updating the positions and velocities of atoms. The force computations make up the majority of the simulation time and can be divided into two types, bonded and non-bonded with the non-bonded force computations making up 80 to 95 percent of the computation and involving many pairwise interactions often over long distances. However, because the effects of these forces over very long distances are minimal a cutoff is often used to compute only the force contributions from those atoms within the cutoff radius, reducing the number of force computations needed.

Also, for accurate solutions the time step must be very small, on the order of femtoseconds. This small time step poses a serious limitation to the total time of simulation because even a nanosecond simulation would involve a million time steps, therefore a million calculations for energy, position and velocity. This is a problem not only because of the time it would take to execute these computations but because of the degradation of accuracy during such long simulation runs. Finally, molecular dynamics simulations may contain a large number of atoms making it difficult to simulate on a traditional memory machine.

Our work focuses on mapping the core computations of molecular dynamics simulation to a PIM array utilizing the mobile thread model. The issues involved in decomposing the computations as well as the structure of the PIM array and execution model will be explained in more detail in the following sections.

3 PIM Array Configuration

The system to which we are mapping the MD simulation is an array of Processing-In-Memory (PIM) chips. In classic systems, processing logic and memory logic are placed on separate chips. However, new fabrication processes have allowed both processor and memory to be placed on the same chip, creating PIMs. On a given PIM chip you may have more than one processor/memory pair termed a node. Each of these nodes can communicate with other nodes on chip very quickly while they must communicate with nodes on other chips via the communication network. As such, PIMs provide the basic building blocks for many different types of systems.

A PIM system could come in several forms (see Figure 1): it could be an array consisting entirely of PIM nodes; relatively small array which constituted part of the memory hierarchy of a conventional processor; or one or more levels of a complex parallel machine's memory hierarchy. Given the diverging assumptions needed by any of the choices above, this work will concentrate on inter-PIM memory addressing and communication.

Since the purpose of PIM is to utilize the on-chip bandwidth of a local memory macro (hence eliminating the von Neumann bottleneck), emphasis will be placed on the ability to

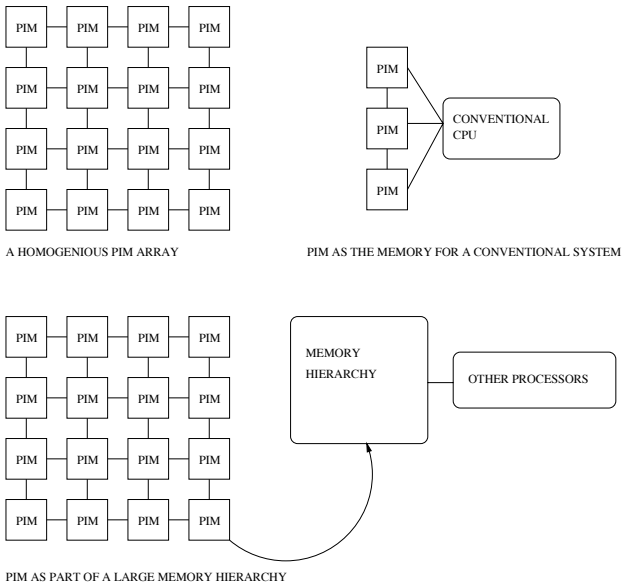

Figure 1. *Types of PIM Systems*

effectively use local memory. Since a full row of memory (approximately 2 K-bits) must be read during each memory access, and this full row can be reused at low cost, effective re-use of the data in these rows is also important. Thus, the question of data placement will consist of several parts: placement within the array; placement within a given node; and placement within an "open row" and potentially even a "wide word" (256 bits).

The PIM communication mechanism is assumed to be a *parcel*. *Parcels*, as defined in [3] have the capability both of simple message-based communication, and of thread initiation (similar to active messages [12]). Since the system is multithreaded, special attention will be made in regard to code access and loading, as well as the potential for covering memory access latency with threads. It should be noted that because parcels can contain the state of a running thread, they are **not** merely a light-weight remote procedure call.

The fabrication process being considered is IBM's Blue Logic SA-27E [4]. Assuming we can fabricate .13 μ PIM chips in this technology we can determine many of the configuration and performance characteristics of the PIM chips in our array. These basic configuration and performance numbers are: 250,000 chips, 4 nodes per chip, 8 MB of memory per node, 13 ns memory random access time, 6 ns memory access time in page mode, and a simple 32-bit RISC processor running at 1 GHz. Given these numbers we have $250,000 * 4 = 1$ million processors running at 1 GHz which gives us the potential for petaflop performance.

We must also consider the interconnection network. The configuration that provides the best performance is one of the parameters we need to determine. A hypercube with an assumed 4 Gb/s is one that we are considering, based on work by Dally and Horowitz at Stanford [9]. The Data Vortex [2], a high bandwidth optical network with a parallel

cylindrical topology, is another possibility as it boasts 640 Gb/s on each port. As we further develop our model and simulations we will be better able to analyze the interconnection configuration.

4 The Execution Model

Previous work [6] indicates that a given PIM node can sustain significant computation by accessing only the contents of its working set, which is defined as the quantum of data upon which a given thread operates over a finite period of time. This is primarily how all parallel machines function. Execution proceeds over the working set until the working set must be updated or altered. When that happens, execution stops (at least for the thread which is attempting to access data outside working set) until the set can be updated. The expense of updating the working set is defined by each architecture. On more traditional parallel machines, that cost is driven by the speed of the interconnection network and coherency hardware. Multithreading ameliorates this cost by allowing another thread to execute in the place of the blocked thread. However, on a very large PIM system the sheer scale may make it very difficult for the program to provide enough threads to keep the machine occupied.

The proposed method of updating the working set for the purposes of this paper is the *mobile thread*, or *rolling snowball*. Rather than bringing the needed data to a thread which requests it, the thread's execution is brought to the data. PIMs are designed to take advantage of local memory accesses. Therefore, the ideal situation is for a thread to execute for a very long period of time on one node then move to the next.The cost of *local* memory accesses is tolerated by having enough threads executing on a given node. The cost of a *remote* memory access is amortized by the presumption that a thread, once moved, will execute on the node to which it moves for a significant period of time. The data placement and structure of the program must match these assumptions so that high performance (measured in terms of the system's throughput) can be achieved. Data placement, as well as capturing the small amount of data which must be moved with a thread, is critical to avoid thrashing. Additionally, "hot-spots" must also be carefully avoided. How this execution model is utilized to map a molecular dynamics simulation to a PIM array will be discussed in more detail later in the simulation description.

5 Computational Model

A computational model of algorithmic scalability is being developed to allow us to determine computationally the various scalability and system parameters needed to achieve a petaflop. This model takes into account the multithreaded execution, communication costs, and other characteristics of the molecular dynamics simulation and PIM arrays. Computational models and simulation are used frequently in the development and analysis of architectures both to determine design parameters and to analyze performance. These models frequently provide numbers which fairly accurately reflect actual execution data as illustrated by Agarwal in [1] where he proposed and validated an analytical performance model for multithreaded processors that included cache interference, network contention, and context-switching overhead effects.

In order to analyze the molecular dynamics simulation on a PIM array we must con-

sider the mapping of the simulation functions to individual threads and their resulting computation and communication behavior as well as the mapping of the data to PIMs. At the thread level this requires considering such issues as the average number of integer and floating point instructions between loads and the number of loads from local versus non-local memory, while at the system level we must consider the number of threads per processor and the number and size of threads that move between processors.

We have an initial model that addresses the issue of the number of threads needed per processing node to mask the memory access latencies. This model focuses on a number of different variables including:

- N_I = Number of instructions
- N_{Ops} = Number of operations between memory accesses
- N_C = Number of remote memory accesses (communications)
- N_{PIM} = Number of local memory accesses

The model can then use this information combined with latencies to predict the number of threads needed for optimum throughput.

In order to develop this computational model we focus on the instructions, most specifically the memory reference instructions, found in a given program and their breakdown. Every instruction can either be a memory reference or a non-memory reference (operation) and the associated probability of each type can be statistically determined for a specific piece of code. Memory reference instructions can again be divided into two categories depending on whether they are local or non-local references. Finally, local references can be differentiated depending on whether the item referenced is already held in the memory macro's row buffer or whether it has to be randomly accessed and pulled into the row buffer. Each of these different types of memory accesses has a different access latency as well as a different probability of for a specific code/data set.

We define three different types of memory access latency equations: a "normal access" (row hit), a local random memory access, and a remote memory access. In a "normal access" the data is found in the local memory macro's row buffer and the equation is

$$P_{MR} * P_{Local} * P_{Row} * L_{Row}$$

where P_{MR} represents the probability of a memory reference, P_{Local} represents the probability that the reference is local, P_{Row} represents the probability the data is in the row buffer, and L_{Row} represents the latency of an access to the row buffer which we assume to be approximately one clock cycle. In a local random memory access the data is not found in the row and the associated equation is

$$P_{MR} * P_{Local} * (1 - P_{Row}) * L_{PIM}$$

where L_{PIM} represents the latency for a random memory access. Finally, a remote memory access is represented by

$$P_{MR} * (1 - P_{Local}) * L_{Remote}$$

where L_{Remote} is the latency for access to data in a remote memory. Currently L_{Remote} is a single value but, in actuality, it depends on the location of the remote value and the characteristics of the communication system. As we revise the model to increase the accuracy

this will be taken into account and L_{Remote} will likely become a function reflecting these parameters.

Given these latency equations we can define our model for determining the number of threads needed to mask memory access latency. Ideally we want to mask the remote access latency as that would be the longest. In order to compute the number of threads needed to do this we compute the latency of a remote access divided by the number of operations per thread. Thus the number of threads is:

$$\frac{L_{Remote}}{N_{ops}}$$

However, in more practical terms we want to be able to mask the *average* memory access latency where L_λ represents the average time for an arithmetic operation. In this case the number of threads is represented by:

$$\frac{P_{MR}}{L_\lambda}(P_{Local}P_{Row}L_{Row} + P_{Local}(1 - P_{Row})L_{PIM} + (1 - P_{Local})L_{Remote})$$

This initial model allows us to determine the number of threads necessary to mask the average memory latency for a given piece of code. The use of simulations will both provide some of the parameters for the model as well as provide a means of comparison to determine the accuracy of model.

Given this simple model we have begun to extend it to allow comparison of different implementations. We assume the following characteristics:

- L_C = Latency for a parcel communication (remote access)
- L_{0ps} = Latency for an arithmetic operation
- L_{PIM} = Latency for a local memory reference

So, our equation for total execution time per thread is

$$N_C(L_{Ops}N_{Ops} + L_{PIM}N_{PIM} + L_C)$$

For each of these values, averages can be assumed such as the average remote access latency that we assumed for our initial model or statistics can be gathered to further break down the characterization such as was done for memory accesses.

$$L_{PIM}N_{PIM} = N_I * P_{MR} * P_{Local} * (1 - P_{Row}) * L_{PIM}$$

A similar breakdown for the arithmetic operations would require that we consider differences in instruction latencies. For simplicity in our initial model we assume that one instruction is equivalent to one clock cycle while in actuality there are differences between various integer and floating point operations. Most significantly, divides take much longer than other operations. By finding the percentage of divides in a given code and the associated latencies and incorporating these factors into the model we can increase the accuracy of the execution time.

$$L_{Ops}N_{Ops} = N_I * (P_{DIV} * L_{DIV} + (1 - P_{DIV}) * L_{INST})$$

Finally, we can extend the communication to include parcel packing and unpacking overhead, L_{PO}, parcel size, S_P, and interconnection bandwidth, BW, yielding:

$$L_C = L_{PO} + S_P/BW$$

Together these give us a much more accurate representation for the execution time of a thread. However, the model still does not accurately represent the interaction of multiple threads both within a processor and within the interconnection network. Currently our simulations focus on the activities within a single thread but further simulations will address the issues of the movement and interaction of threads among nodes in the system and will help to provide this additional data. The final model, in conjunction with simulation, will be used not only to predict the number of threads, but also other system configuration parameters to allow us to reach petaflop execution. The details of the various simulations and their roles is discussed in the next section.

6 Code Decomposition

The simulation code used for our initial numbers is SAMD2, a serial object-oriented molecular dynamics simulation using Blitz++ [13]. Force computation is the most performance-intensive part of the simulation, so this part of the program has been optimized. To compute the non-cutoff non-bonded forces, blocks of atom pairs are evaluated, and each distinct pair is passed to the one-atom-pair evaluator. For cutoff non-bonded forces, cell list pairs are pre-tested, and if they pass, they are looped through and evaluated by the same do-one-atom-pair evaluator as before. On each pair, if the switching function is enabled, a rough test is performed to determine if this pair is completely out of range. If it is not, the switching function and non-bonded force function are evaluated. The product rule for derivatives is used to compute the forces, which are then multiplied by the difference between the atoms, and this is added into the atom forces. To compute bonded forces, a list of the atoms and type of each bonded force is kept. To evaluate the forces, this list is looped through one force at a time, and a bonded force class is called on each one. The bonded and non-bonded forces are from a CHARMM force field from the NAMD *Programming Guide*, Version 1.5.

Because of the size, complexity, and time scale of molecular dynamics simulations, parallel implementations are often necessary. There are a number of approaches to decomposing an MD simulation to enable it to be implemented in parallel across a number of processors. These approaches focus on both the distribution of the atoms and associated state as well as the distribution of the force computation. Force decomposition (FD) is an approach where the N x N force matrix is partitioned into P blocks and each block assigned to a different processor. Each block requires $2 * (N/\sqrt{P})$ atoms to compute the force and, as such, may require up to $O(N/\sqrt{P})$ non-bonded communications. Spatial decomposition (SD) is an approach where the system is partitioned into boxes (cells) of a size a little larger than the cutoff distance. These boxes are then distributed across the processors in the system and the computation of the forces for each atom in the box is computed. In this way all the communications for cutoff computations are with neighboring boxes, possibly on the same processor.

The type of decomposition that is most efficient depends largely on the configuration and execution model of the system on which the simulation is being run. For our

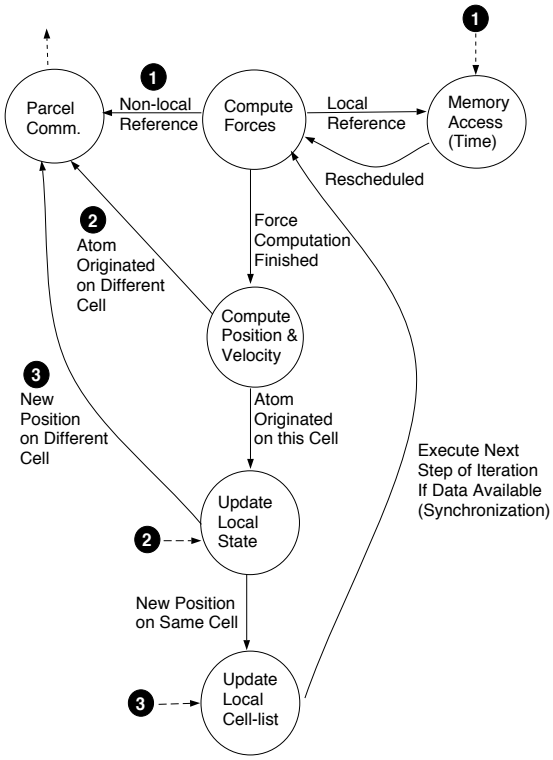

Figure 2. *Execution Flow of a Thread*

simulations, the serial code for the non-bonded force computation was broken down into individual threads to be mapped onto the multithreaded SHADE simulator. A spatial decomposition was used whereby each cell-list was placed on a different processor. Each atom was then instantiated as a separate thread and that thread computed the force computations for the atom associated with it. In this way, each thread iterated through the atoms in the cell-list computing the accumulated force on its atom and then moving to another processor's cell list. The instructions in each individual thread were tracked to determine data such as the number of integer and floating point instructions between memory accesses and the number of local and remote memory accesses. The details of the simulation will be discussed in the next section.

In addition to the SHADE simulation, a hybrid simulation was developed that considers a system of chips and the communication and movement of threads among chips. In this way we will be able to gather data on communication patterns and latencies to facilitate the evaluation of various configuration parameters. The sequential code and SHADE results were used to develop the execution flow shown in Figure 2 for mapping the molecular dynamics simulation to the hybrid PIM system based on the spatial decomposition described. The flow represents the execution for one thread during one iteration of a molecular dynamics simulation. The bulk of the execution takes place during the computation of forces.

This thread represents one atom computing the sum of its forces with every other atom within its cutoff range. For each instruction issued it is either computing the forces or it is accessing memory with the memory access time determined by the type of access as discussed previously. If it does a non-local memory access this implies that the reference has moved outside of its current cell and off-chip. This produces a parcel communication (labeled 1) which carries the thread to a new chip (and new cell) within which it will continue with its memory access and force computations. (Note that the dotted lines in the figure represent parcel movement to a new chip.) Ultimately, when the thread has completed its force computations it must update its local position and velocity which may or may not be on its current chip. Then, based on that new position, it may or may not reside in the same cell as previously and must update the cell-list of the cell within which it now resides before repeating the process for the next iteration. It is also important to note that all atoms in a cell must have updated their local state and the associated cell-lists before the next iteration can begin.

This decomposition allows many threads to be running simultaneously, taking advantage of the multithreading available in the PIM system, and allows the thread to move its computation to take advantage of the locality provided. Additional discussion of the simulations and associated results follows.

7 Simulation

Two phases of simulation are used in our work. First, a SHADE based simulation was used to analyze the functions that implement a single thread and to trace memory access patterns over several iterations of the force calculations (currently only the non-bonded calculation since it is the most computationally intensive). Second, using the information collected from SHADE, a hybrid PIM system simulator creates theoretical threads executing on a machine of virtual nodes. The virtual nodes do not perform the actual execution, rather they account for thread execution and remote and local memory access latencies using a dynamically reconfigurable machine model and provide additional information on system overhead and other aspects of the execution. Ultimately the simulator can be extended to simulate up to millions of threads executing on millions of processors.

7.1 SHADE Simulation

One principle mechanism for benchmarking throughout this research was the use of the SHADE suite [10] developed by Sun Microsystems. The tool allows any SPARC binary to be analyzed architecturally in detail by providing a simple mechanism for analysts to write their own code to track the execution of an application. SHADE provides information about every instruction which is executed, as well as the effects which that instruction had on the SPARC machine simulated.

As Figure 3 shows, programs are viewed simply as streams of instructions. The simulator written for the purposes of this work uses those streams of instructions, combined with information SHADE provides about the state of the machine, to perform accounting for whatever is being tracked. There are some key things which SHADE **does not** do. It is incapable of tracing calls to the kernel, and therefore does not include accounting for system

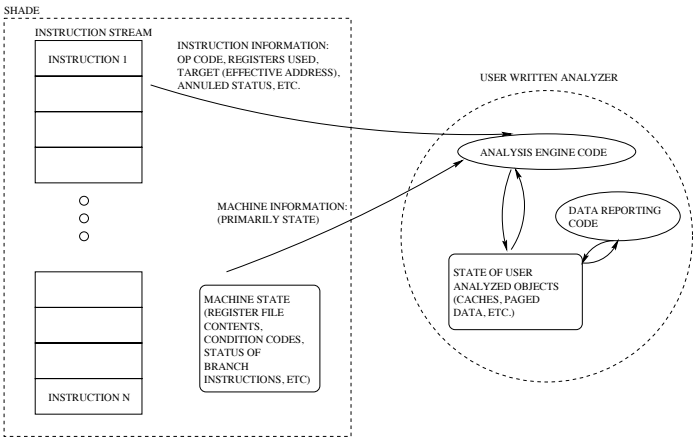

Figure 3. *SHADE Simulations*

overhead. Generally, for the type of benchmarking we are performing that is advantageous since only user code is of interest. Furthermore, SHADE does not trace multithreaded applications. However, a package to allow SHADE to perform accounting on simple run to completion threads was developed and used to simulate simple multithreading. This required extensive kernel modifications and accounting.

7.2 SHADE Results and Analysis

The data from the initial SHADE simulations have allowed us to compute the averages across the threads for a number of different variables. Further analysis can yield additional information such as probability distributions and access patterns, which sometimes yield additional insight. The important initial simulation results are as follows:

- Percent of remote memory accesses per thread – 23%
- Average number of integer operations between memory accesses – 5.44
- Average number of floating point operations between memory accesses – 2.56 FP
- Number of instructions per thread per iteration – 3,653

Additional SHADE data includes the number of each type of memory access and the total number of instructions. Given this information, the percentages that are needed by our computational model can be computed.

- $P_{MR} = 12.5\%$
- $P_{Row} = 35\%$
- $P_{Local} = 77.2463\%$
- $P_{Remote} = 1 - P_{Local} = 22.7537\%$

In addition to the memory access percentages, the memory access latencies are needed for the model. Assumptions for these latencies were made based on our work with various systems. These are as follows:

- $L_\lambda = 1$ ns (i.e, 1 GHz clock)
- $L_{Row} = L_\lambda = 1$ ns
- $L_{PIM} = 10$ ns
- $L_{Remote} = 500$ ns

Finally we have the information necessary to utilize our model and compute the total number of threads needed to mask the average memory latency in this application.
Number of threads (T) =

$$\lceil \frac{.125}{1ns}[(.772463)(.35)(1ns) + (.77)(.65)(10ns) + (.227537)500ns]] \rceil = 16$$

However, if we use the numbers specific to our IBM process based PIM we have $L_\lambda = 1$ ns (i.e, 1 GHz clock), $L_{Row} = 6$ ns, $L_{PIM} = 13$ ns, and $L_{Remote} = 500$ ns yielding:

$$\lceil \frac{.125}{1ns}[(.772463)(.35)(6ns) + (.77)(.65)(13ns) + (.227537)500ns]] \rceil = 18$$

These numbers are very close to what our initial estimates indicated. They imply that to achieve petaflop performance we would need 16 to 18 million threads running on a system of $250,000$ chips with 1 GHz processors. For some very large simulations this may be reasonable as we would have N_{atoms} threads computing forces and may have additional threads for other computations such as cutoffs, non-bonded force calculations, and other aspects of protein folding not included here. These results will be used as a part of the hybrid simulator as parameters to further examine the simulations on a system of PIMs. This hybrid simulation is discussed in more detail in the next section.

7.3 Hybrid PIM System Simulation

The Hybrid PIM System Simulator is a an object-oriented simulator developed in C++ with an integrated event-based simulation package. It models a system of multithreaded PIM chips with a user-defined interconnection scheme and allows a number of system configuration and timing parameters to be varied such as the number of processors, interconnection scheme, parcel transmission times, and memory access times. This allows the system to efficiently determine design parameters, analyze performance, and simulate applications. The timing parameters for the simulation are determined in a number of ways including modeling and other lower level simulations. For this work, the results from the SHADE simulation were used to specify the distribution of memory references and associated times as well as the average execution times between memory references.

The simulation was set up as illustrated in Figure 2. In this simulation the actual molecular dynamics computations are not executed but the associated timings and movements are accounted for. It is assumed that the data is distributed spatially with a cell on each processor and we focussed on the force computations. One thread is created for each atom in the space to compute the forces with every other atom. The data from the SHADE simulation is used to probabilistically create the average number of each type of memory reference and the average number of instructions between memory references for a thread. Then, based on the type of memory reference or instruction, the associated simulation time

is applied. In addition the overhead of thread and parcel management are included. The interconnection scheme used is a crossbar as it logically represents the Data Vortex which can send a message from any chip to any other simultaneously.

7.4 Hybrid Simulation Results and Analysis

Simulation was initially focused on one thread executing on a system of 1000 PIM processors. We assume that one time step in the simulation is 1 ns. The percentage of each type of instruction matched what was set in the parameters from the SHADE simulator, illustrating the accuracy of the program with respect to the SHADE parameters. However, it also provided information regarding additional overhead not taken into account by the model. The important initial thread results include:

- Total thread lifetime = 48780 ns
- Execution = 3600 ns = 7.3%
- Row Access = 119 ns = 0.24%
- Random Access = 1980 ns = 4.1%
- Remote Access = 41583 ns = 85%
- Overhead = 1498 ns = 3.1%

These numbers reflect the importance of multithreading in masking memory access time, especially the remote access time. As we incorporate more accurate parameters for the parcel communication scheme, such as the bandwidth, parcel handling overhead, and parcel sizes we can more accurately gauge this aspect of the application, which is extremely important for performance. The results also allowed us to examine the effect additional overhead of thread and parcel management have in comparison to memory accesses and computation. In this case the interconnection was a crossbar, so congestion on the network was not a factor. However, this could be a factor in cases with a different interconnection or simulations with thousands of threads running simultaneously on thousands of processors. The simulator allows access to various types of metadata including queue depths for interconnection networks, and time spent in thread scheduling and parcel handling. Now that the basic simulation structure is in place we plan to extend the simulation to use such metadata for an increasingly detailed model of parcel communication, instruction mix and associated execution times.

8 Future Work

In addition to the incorporation of a more detailed model parameters, we will work to develop and simulate different decomposition methods. For example, a force decomposition could be implemented with one thread per atom, or even one thread per block, illustrating the tradeoff between number of threads for amount of computation. Also, load balancing in the spatial decomposition would be an issue to consider for simulations where large changes in position occur. The hybrid simulation will also be extended to examine thread movement and communications issues in more detail and incorporate the results into the computational model. Finally, a larger and more detailed simulation of the PIM system configuration with up to millions of processors and threads will be developed to examine the configurations and parameters necessary for petaflop performance.

Bibliography

[1] Agarwal A., "Performance Tradeoffs in Multithreaded Processors," Private Communication, 1989.

[2] Bergman K., "Ultra-High Speed Optical LANs," *Conference on Optical Fiber Communications*, Workshop on LANs and WANs, San Jose, CA, February, 1998.

[3] Kogge, Peter M. and et al., "Final Report: PIM Architecture Design and Supporting Trade Studies for the HTMT Project", September, 1999.

[4] IBM: ASIC SA-27E Standard Cell/Gate Array. http://www.chips.ibm.com:80/products/asics/products/sa-27e.html (7 Apr. 2000)

[5] IBM: IBM Unveils $100 Million Research Initiative to Build World's Fastest Supercomputer. http://www.ibm.com/news/1999/12/06.phhtml (10 Apr. 2000)

[6] Murphy, R., *Design Parameters for Distributed PIM Memory Systems*, MS Thesis, CSE Department, University of Notre Dame, May 2000

[7] Narumi, T., R. Susukita, T. Ebisuzaki, G. McNivern, and B. Elmegreen, "Molecular Dynamics Machine: Special-Purpose Computer for Molecular Dynamics Simulations" in Molecular Simulation. Vol. 21, 1999, pp. 401-415.

[8] Silicon Graphics, Inc.: The Great Leap Forward in Molecular Structure Simulation and Modeling. http://www.sgi.com/features/1999/mar/chemistry/index.html (10 Apr. 2000)

[9] Stanford, "High Speed Signaling",http://pasta.stanford.edu:80/hssp/, (April 2000).

[10] Sun Microsystems, *Introduction to Shade*, June, 1997.

[11] Thornley, J., M. Hui, H. Li, T. Cagin, and W. Goddard, "Molecular Dynamics Simulation on Commodity Shared-memory Multiprocessor Systems with Lightweight Multithreading" in *Proceedings of High Performance Computing*, 1999.

[12] von Eicken, T., D. Culler, S. Goldstein, and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", *Proceedings of the 19th International Symposium on Computer Architecture*, ACM Press, 1992.

[13] Willcock, J., "SAMD2: A Molecular Dynamics Simulator Using Blitz++" *Proceedings of CSE598E, University of Notre Dame*, May 2000