

# Performance Evaluation of an Irregular Application Parallelized in Java

Christopher D. Krieger and Michelle Mills Strout  
Computer Science Department  
Colorado State University  
Fort Collins, CO 80526  
krieger|mstrout@cs.colostate.edu

**Abstract**—Irregular scientific applications are difficult to parallelize in an efficient and scalable fashion due to indirect memory references (i.e. A[B[i]]), irregular communication patterns, and load balancing issues. In this paper, we present our experience parallelizing an irregular scientific application written in Java. The application is an N-Body molecular dynamics simulation that is the main component of a Java application called the Molecular Workbench (MW). We parallelized MW to run on multicore hardware using Java’s *java.util.concurrent* library. Speedup was found to vary greatly depending on what type of force computation dominated the simulation. In order to understand the cause of this appreciable difference in scalability, various performance analysis tools were deployed. These tools include Intel’s VTune, Apple’s Shark, the Java Application Monitor (JaMON), and Sun’s VisualVM. Virtual machine instrumentation as well as hardware performance monitors were used.

To our knowledge this is the first such performance analysis of an irregular scientific application parallelized using Java threads. In the course of this investigation, a number of challenges were encountered. These difficulties in general stemmed from a mismatch between the nature of our application and either Java itself or the performance tools we used. This paper aims to share our real world experience with Java threading and today’s parallel performance tools in an effort to influence future directions for the Java virtual machine, for the Java concurrency library, and for tools for multicore parallel software development.

**Keywords**-Java performance tuning, performance analysis tools for Java, multicore, molecular dynamics, irregular applications

## I. INTRODUCTION

A number of critical scientific computations are considered irregular: finite element analysis, N-body problems, sparse matrix computations. Molecular dynamics simulation is an example of an irregular application that is critical to many areas of science, such as materials science, biochemistry, and pharmaceutical development. For this reason, much research [1], [2], [3] continues to center around the parallelization and performance optimization of molecular dynamics simulations.

In this paper, we present our experience parallelizing and analyzing the performance of a 3D molecular dynamics simulation engine written in Java. This irregular

application is the main component of the Molecular Workbench (MW) [4]. Molecular Workbench is a pedagogical tool designed for use in teaching science principles to high school students. The MW simulation engine is coupled with an interactive graphical interface, enabling the user to watch simulated atoms act under the influence of fields and other atoms. We are using MW as part of a high-school outreach component of the Parallelization using Inspector/Executor Strategies project<sup>1</sup>.

The parallelization of MW is motivated by the need to execute simulations fast enough to maintain a non-jerky refresh rate for the display. At present, the serial version of MW can satisfy this requirement for simulations of at most a few hundred atoms on newer machines. Ideally, MW would be able to smoothly simulate one thousand atoms on a recent quad-core system. As a result of parallelization, this goal has largely been reached. On a quad-core system, MW can now sustain refresh rates as high as 32 updates per second on some 1000 atom benchmarks.

To improve the performance of the MW 3D molecular dynamics simulation engine, we did a relatively straightforward parallelization using the Java concurrency library *java.util.concurrent*. One or more fixed sized thread pools are created when the MW application starts. At each phase of each timestep in the simulation, the work at hand is sent to the worker threads assigned to these thread pools. Synchronization between threads is handled by simple barriers.

We tested three benchmarks that exercised different parts of the MW molecular dynamics engine. These benchmarks contained on the order of 1000 atoms, so they were larger than could be practically handled by serial MW code. We found that the scalability of the parallelized code was inconsistent across the benchmarks. Two of the benchmarks scaled adequately, while the third exhibited very little speedup going from one to four threads. Because this third benchmark is representative

<sup>1</sup><http://www.cs.colostate.edu/hpc/PIES/>

of the most commonly exercised code, its performance is critical.

We turned to instrumentation and performance monitoring tools to determine ways of improving the performance. A range of tools were used to analyze the behavior of MW, including Intel’s VTune, Apple’s Shark, the Java Application Monitor (JaMON), and Sun’s VisualVM. Virtual machine instrumentation as well as hardware performance monitoring data were used.

Based on work done to improve performance of molecular dynamics simulations in languages other than Java [1], we hypothesized that the poor scalability was due to either load imbalance between threads or limitations on memory subsystem bandwidth.

Initial measurements appeared to show that load balance was quite good. Each worker thread reported running for approximately the same amount of time. As we dug into these measurements, we found that these measurements alone provide an overly simplistic view of load balance. An equal total amount of time spent by a worker thread in its work routines may or may not indicate good load balance. Imbalance on any particular iteration can disappear when averaged over many iterations. Imbalance can also occur at a finer granularity than performance tools can measure. Finally, the impact of the so-called *observer effect* of instrumentation can be much larger than the load imbalance one is trying to measure. In Section IV, we discuss further the issues with measuring load balance.

Additional measurements indicated that load imbalance was not responsible for the majority of the scaling issues, so we turned our attention to memory bandwidth. In this area, we again encountered difficulties with both Java and performance measurement tools. In fact, simply determining the degree of data locality is challenging because obtaining the actual location in memory of Java objects is difficult. Tools that provide a wealth of other information, such as garbage collector performance, do not expose the virtual addresses of objects. We were forced to rely on indirect measures, such as cache miss rates, to determine the extent of locality issues.

The “hands-off” policies of the Java virtual machine prevented the application of techniques commonly used to improve memory performance of irregular applications written in other languages. For example, data packing to improve spatial locality is not practical in Java. The Java memory manager prevents direct user control over locating objects in adjacent locations in memory. Undeterred, we attempted to reduce memory traffic through management of caches instead. Again, our efforts were thwarted by the Java virtual machine. As Java lacks any mechanism for binding threads to physical cores, it is impossible to place work on threads such that the executing cores can share data or capitalize

on already warm caches. Performance tools revealed an additional issue with cache pollution as well.

Throughout our experience parallelizing MW, the mismatch between Java and high performance irregular scientific applications became increasingly apparent. Attempting to understand, quantify, and, where possible, resolve the issues by working with existing tools infrastructures exposed incompatibilities between those tools and what performance tuning of scientific applications required.

In Section II, we provide more details about the performance behavior of MW and the strategy used to parallelize it. We then give a brief description of our evaluation methodology, including benchmarks and reference hardware, in Section III. With that background in place, Sections IV and V chronicle our investigation into load balancing and memory performance, respectively. Sections VI and VII close the paper with related work and some comments regarding future directions for Java parallelization and parallel performance tools.

## II. STRUCTURE OF PARALLEL MOLECULAR WORKBENCH

In this section, we discuss the design of the parallel MW 3D molecular dynamics engine. In particular, we describe the overall layout of the application, what portions are independent and where barriers are needed to provide synchronization essential to correctness. Understanding this structure is a prerequisite to understanding work distribution and load balancing issues. Likewise, we discuss the different force computations found within MW. This information provides the necessary context for later discussions of memory bandwidth.

### A. Program Structure

Molecular Workbench uses a full classic molecular dynamics algorithm for simulating atom interactions [5]. At each timestep, the position and velocity of each atom is predicted by applying a second order Taylor expansion of the basic equations of motion to the current position, velocity, and acceleration. Next, the new forces acting on the atom are computed using these predicted values and other data from the previous timestep. Finally, a corrector step is performed that updates the atom velocities based on the newly computed forces. One pass through these stages constitutes a single timestep of the simulation, typically representing on the order of 1-2 femtoseconds of simulated time.

The work to be done during each timestep of the simulation breaks neatly into the following phases:

- 1) run the predictor for each atom
- 2) check whether the neighbor list is still valid,
- 3) if neighbor list is invalid, repopulate the linked cells and build the neighbor lists

- 4) calculate the forces on each atom from each relevant type of interaction
- 5) perform a reduction across all copies of the privatized force array, and
- 6) run the corrector for each atom.

Within each phase, the processing of each atom is independent of the others. However, each phase must complete its processing on all atoms before the next phase begins except for phases (3) and (4), which we fused into a single loop to improve data locality and reduce loop overhead.

### B. Force Computations

Three different interatomic forces are computed during phase 4, as necessary. The first is the force between non-bonded atoms, found using the Lennard-Jones (LJ) approximation. To improve performance, these forces are only computed between atoms that are within a cutoff distance, or neighborhood, of each other. When two atoms are farther apart than the neighbor limit, the Lennard-Jones force is considered to be zero. The generation of neighbor lists is done at the start of the simulation and when any atom moves in any dimension by more than a threshold value. We use a *linked-cell algorithm* [6] that keeps the complexity of the neighbor-finding algorithm to  $O(N)$ . Conceptually, the linked-cell approach superimposes a three-dimensional grid over the simulation space. The grid is sized such that the neighbors of any given atom must fall within the grid box containing the atom or in one of the grid boxes adjacent to that box.

The second type of force that MW calculates is the Coulombic force between charged particles. Unlike LJ forces, Coulombic forces are calculated between every pair of charged particles, regardless of distance. A particle-mesh-Ewald method [7] would have lower algorithmic complexity at  $O(N \log N)$ , but its use is a future work direction due to its implementation complexity.

The last category of forces calculated during the simulation are those caused by bonds. The forces between the bonded atoms are computed in the order the bonds appear in the bond list. Bond force equations are more complex than the other types, require more floating point operations, can involve up to four atoms, and exhibit indirect and therefore irregular indexing into the atom array.

The approach used to parallelize the MW application is a fairly straightforward work queue pattern. A number of fixed-sized thread pools, managed by Java ExecutorServices, is created at simulation start time. Typically, one thread is created per core in the system, but this parameter is adjustable. Each thread pool has a work queue associated with it. When work is present in the queue, the ExecutorService removes the work and assigns it to

a thread. If all threads are in a single thread pool, they share a single work queue. This has the advantage that if any work is waiting to be assigned, it will be picked up by the next available thread. On the other hand, having a single queue means that all threads are contending for access to that single resource. Conversely, having one queue per thread eliminates contention, but can result in the situation where one queue has considerable work while other threads, with empty work queues, sit idle. In the course of developing parallel MW, we used both single and multiple queue configurations.

During each phase as previously discussed, each thread is assigned a fraction  $1/N$  of the total atoms to process, where  $N$  is the number of threads. Because the processing of each atom in a given phase of the simulation is relatively uniform in terms of execution time, most phases do not require a sophisticated work balancing strategy. When the thread finishes its work, it decrements a countdown latch so the program knows when all work in the phase is complete.

The combined phases 3 and 4 are a notable exception to this process. The complexity of finding the neighbors of a given atom is a function of the number of candidate atoms in adjacent linked cells. Likewise, the complexity of force calculations on an atom depend on many factors, including the number of bonds involving the atom, whether it is charged, the number of other charged atoms, how many neighbors the atom has, and the atom index number. The atom index number is used to compute the force between a pair of atoms only once. When the lower indexed atom is processed, the force is computed and stored for both atoms. Thus, lower numbered atoms in general require more computation than higher indexed atoms. This variability is in addition to the already uneven amount of work due to potentially different numbers of neighbors. This variability in per-atom processing plays a role in load imbalance discussed later.

## III. EXPERIMENTAL WORKLOADS AND TEST SYSTEMS

To better understand the variety of simulations users execute on the MW engine, we surveyed the simulations available from the Molecular Workbench online repository [8] and consulted with the MW developers [9]. We found that simulations generally fall into one of three categories based on the type of force that dominates the simulation, namely Lennard-Jones, Coulombic, or bonded. In the vast majority of the simulations, Lennard-Jones forces predominated, but there were cases that strongly favored the other types of forces. We selected one case to represent each of the three categories (see Table I).

Benchmark	# of Atoms	# of Charged Atoms	# of Bonds	Dominant Computation Type
nanocar	989	0	2277	Bonds
salt	800	800	0	Ionic
Al-1000	1000	0	0	Lennard-Jones

TABLE I: Representative Benchmark Characteristics

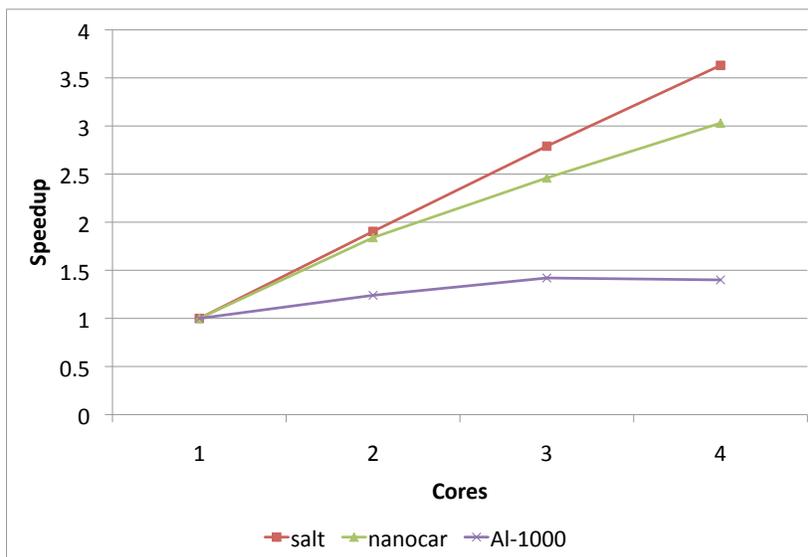


Fig. 1: Observed Speedup on an Intel Core i7 System

A typical MW case in the repository contained less than 100 atoms due to performance limitations of the serial code. A goal of this parallelization effort was acceptable parallel performance on simulations of approximately 1000 atoms. For this reason, we selected one published, large (989 atom) benchmark (nanocar) and increased the atom count of two other reference simulations by a factor of eight to ten. Each of the benchmarks had a working set size of about 25 MB.

The *salt* case is a simulation containing 400 sodium ions and 400 chlorine ions. There are no bonds in this simulation, but every atom is a charged ion, interacting with each other via Coulombic and potentially LJ forces.

The *nanocar* [4] test, on the other hand, emphasizes bonds. About half its atoms are bonded together to form a “nanoscale car” with the other half making up an immovable platform of gold on which the car “drives.” Because fixed-location atoms making up the platform do not interact with one another, this simulation has a lower effective atom count and requires far fewer Coulombic and LJ force computations than the other examples.

The last test case, *Al-1000*, is a densely packed

stationary block of 999 aluminum atoms hit by a single, fast-moving gold atom. This case has a large number of collisions and requires frequent neighbor list updates.

For benchmarking purposes, each of the simulations was run with the graphical display set to the default size. There was no user interaction with the GUI. Each run simulated the molecular system’s behavior for 20 picoseconds at 1 or 2 femtoseconds per timestep.

Because MW is being used in the context of high school outreach, we assume the availability of four core systems in that setting in the near future and did most of our testing and simulation with that target in mind. As can be seen in Fig. 1, the speedup for the salt benchmark was quite good, yielding a 3.63x speedup on four cores. Likewise, the nanocar test benefitted from a 3.03x speedup. However, the Al-1000 benchmark scaled quite poorly, achieving a maximum speedup of 1.42x.

Since Al-1000 represents the majority of the simulations in the repository with its emphasis on Lennard-Jones forces, the extremely poor speedup seen there was alarming and precipitated an in-depth performance analysis.

#### IV. ANALYSIS OF LOAD BALANCING

Our investigation into the lack of parallel scalability began by examining whether or not MW was suffering from load imbalance. Specifically, we wanted to know if most threads were wasting processing power waiting at a barrier at the end of a phase while one thread took significantly more time to complete its portion of the work and reach the barrier. We turned to our collection of performance tools to answer that question.

##### A. *Observer Effects*

Initially, we inserted performance monitors into the code using the Java Application Monitor (JaMon). In theory, these would allow us to see how much time was spent in each phase by each thread. If each thread spent an equal amount of time in a phase, that would strongly indicate a well-balanced system. JaMon would also show how many threads were executing in parallel within a phase. If the average number of threads concurrently executing the phase was equal to the number of threads, that fact would be indicative of good load balancing.

Unfortunately, in order to allow multiple threads to update the performance counter variables safely, JaMon uses synchronized sections. We discovered that these synchronized updates to the performance monitors were serializing the overall performance of MW and drastically impacting the very behavior they were intended to measure.

This was the first of the so-called *observer effects* that we experienced. Likewise, using VisualVM and enabling the per-method cpu utilization instrumentation causes the Molecular Workbench simulation to run at roughly one quarter its normal speed. Much of the system's processing resources are devoted to TCP traffic between the application and the measurement tool. While instrumenting code in general can result in this magnitude of slowdown, in parallel applications structured like MW, it is particularly disruptive. For example, when testing four thread behavior on a four core system, some cores are competing with the instrumentation thread and the performance tool thread. If the worker threads on these cores are delayed, the entire system waits at a barrier for all threads to complete their portion of the current phase's work. This can mask issues with load imbalance, which has a similar signature.

##### B. *Insufficient Sampling Granularity*

Another phenomenon with an impact similar to load imbalance occurs when a thread arrives at the barrier later than other threads, not because it spent more time doing its work, but rather because it started later than other threads. A certain amount of skew is to be expected. However, if skew becomes excessive, particularly relative to the amount of time spent on the actual work, it

can decrease the effectiveness of the system and reduce speedup. In MW, this type of skewed launch could occur if there was contention for the work queue associated with the thread pool.

To decide if either load imbalance or skew was occurring, we gathered data using performance tools. VirtualVM has a graphical thread view displaying the state (running, sleeping, waiting, or blocked by a monitor) of all threads. However, it was sampling at a rate of one sample per second. VTune was able to sample on the order of 5 to 10 milliseconds apart. However, the typical work load in MW takes between 80 and 5000 microseconds to execute, with worst case queue imbalance occurring typically in less than 15 milliseconds. At the thread state sampling granularity of these tools, we were able to observe only the most severe imbalance. This sampling period also generated "false positives," or cases where extreme imbalance was displayed. In fact, these were artifacts. The tool sampled the thread state immediately before it changed, but continued to display the sampled state until the next sample.

The coarse sampling rate of these tools made it impossible to assess load imbalance or skew on the fine-grained level needed by MW.

##### C. *Identifying What Code Each Thread is Executing*

In the process of investigating load imbalance, it was observed that occasionally one thread would execute much longer than the others, or the master thread would execute for several times longer than usual. To explain this behavior, we needed to know what method the thread was executing.

In these cases, the available data from VisualVM consisted of the thread state information indicating that the thread was in an active, running state, and the call stack information reporting which methods were consuming the largest share of execution time. However, using VisualVM, we could see no way to determine, for a given moment in time, what code a particular thread was executing. Shark's Java Time Profile view did provide timestamped call stack traces. However, it would either allow for all threads on a single core to be traced over time, or a single thread as it moved between all cores to be traced. The desired information, specifically what code each thread was executing, was available, but an easy way to compare across the threads was not apparent. A simple way to see what method a thread was executing at a given moment for all threads would be tremendously helpful.

#### V. IMPROVING MEMORY PERFORMANCE

After extensive study of MW's performance, we eliminated load balancing, skew, or parallelization overhead as the major source of scalability issues with the AI-1000

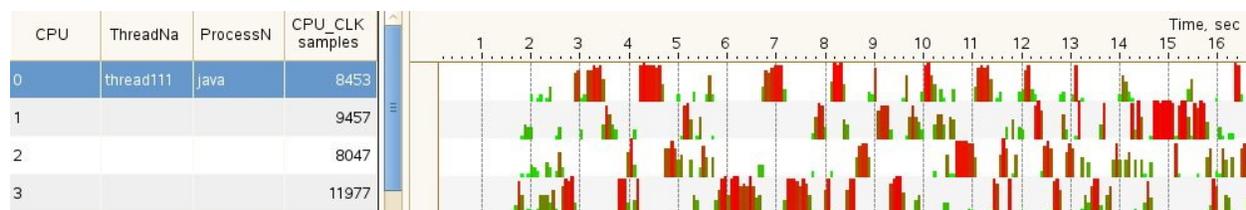


Fig. 2: Worker Thread to Core Affinity Without Pinning. The thread moves frequently between all four cores. Red indicates a heavy load, green a lighter load

benchmark. Prior experience with irregular applications led us to suspect that the performance limiter for MW was the memory subsystem. This suspicion led to an investigation of the memory performance of MW and a search for ways to improve it.

### A. Data Packing

With irregular scientific applications, inspector/executor strategies can often dynamically reorder data so as to improve the spatial locality and consequently the memory performance.

To see if this approach would improve MW performance, we examined the different data access patterns for each of the three types of force computations. Lennard-Jones calculations access atoms that are physically adjacent in simulation space, though not necessarily near one another in memory. Coulombic calculations access atoms in a linear fashion, taking advantage of spatial memory locality if most atoms are charged. Bond computations exhibit an irregular atom usage pattern, accessing atoms based on their participation in bonds rather than in index order.

Seeing that the worst observed performance was on the highly irregular LJ computations, we attempted to do data reordering. Specifically, we attempted to put atoms that were physically proximate in the simulation into adjacent memory locations. MW stores data about each atom in an array of objects and accesses these atom objects irregularly. We wanted to see if we could do runtime reordering of the array to improve spatial locality. We created a new array, then populated it with objects that were created by rapidly successive calls to `new()`. Due to the way the Java memory manager selects the actual memory locations for data, we were unsure if this approach was feasible. We did not see appreciable performance improvement, but needed to confirm whether or not the reordering was behaving as hoped.

We used VTune to measure the mid-level and last-level cache miss rates using the hardware performance monitoring unit and saw no significant improvement. This was a strong indicator that the objects were not being reordered and packed in memory. However, it could also be that the original ordering was nearly ideal.

With existing tools, it is difficult to determine to what degree data is packed in Java.

It would be very informative if there was a heap viewer that would show the actual data addresses of objects in Java. Various hacks can be used, like printing out the `(Object).toString()`, but these are awkward and somewhat VM specific. The Eclipse debugger provides access to objects, but does not provide their memory addresses. There are heap viewers but they tend to show how many objects of each type are on the heap, how many were created within some time or generation, etc. The heap viewers do not show the relative spatial locality of the objects, which is what is needed to identify false sharing or optimize true sharing via data reordering or object member ordering. While Java provides limited opportunity to resolve these types of problems, if high cache miss rates are seen, it can be beneficial at least to understand the memory address pattern.

### B. Direct Cache Management

We next pursued the possibility of achieving better performance via temporal locality rather than spatial locality. In this approach, threads that need access to a particular subset of data are executed on the cores that have already cached that data.

Using the `java.util.concurrent` API, we created for each core a `FixedThreadPool` containing a single thread. By assigning work to the pool, it would be executed by the corresponding thread. Different tasks and computations using the same or similar subsets of the simulation data could thus be directed to the same thread.

This approach encountered two issues. First, the Java runtime, in concert with the underlying operating system, can migrate a thread between various cores in the system. This is particularly frequent when threads encounter synchronization operations, as this can cause the thread to park or be put to sleep. When it awakes, the scheduler will place it on a core based on the system load and some degree of affinity with the previously assigned core.

We used VTune to plot the thread to core affinity of a workload running in Molecular Workbench. In Figure 2, a single worker thread is shown as it executes on the four cores of the system. Red indicates that the thread is heavily loading the core, while green indicates a lighter

Processor Type	Procs x Cores	$L_1$ Data Cache	$L_2$ Cache	$L_3$ Cache	Memory
Intel Core i7 920	1x4	32 kB	256 kB	1 x (8 MB shared/4 cores)	6 GB
Intel Xeon E5450	2x4	32 kB	256 kB	4 x (6 MB shared/2 cores)	16 GB
Intel Xeon X7560	4x8	32 kB	256 kB	4 x (24 MB shared/8 cores)	192 GB

TABLE II: Test Machines and Their Memory Hierarchies

Number of Cores Used	Topology	Runtime (sec)
4	one core per processor	172.2
4	4 cores on one processor	154.7
4	OS scheduled	147.3
8	OS scheduled	164.3
8	two cores per processor	132.0
8	8 cores on one processor	103.7
32	OS scheduled	100.2

TABLE III: Differences in runtime with the same number of cores but different topologies

load. As can be seen, even in a four core system, the degree of thread affinity was quite low. In many cases, the thread visited every core in the system in less than one second.

Since we are using a thread as a proxy for a set of caches, it is critical that the thread stay bound to a particular core. Pure Java does not provide any thread to core binding API, such as is available with `sched_setaffinity` in Linux. To experimentally determine if thread binding would resolve this issue, we evaluated the impact of thread binding by writing a simple C function wrapper around `sched_setaffinity` and using the Java Native Interface to access it from Java.

Table III shows the impact of thread pinning. The column labeled “topology” describes where the threads were bound. An entry of “OS Scheduled” indicates that the full core count was made available and the OS was unrestricted as to where to schedule a thread. Other entries indicate the topology of the cores that were made available via the `cpu mask` given to `sched_setaffinity`.

To better understand the efficacy of thread binding, we ran experiments on the quad-core system, but also used an eight core and a thirty-two core system. Details of these machines’ memory and processing capabilities are given in Table II.

As shown in Table III, with low core counts, more flexibility regarding on which core to assign a thread results in better performance, as the OS can avoid cores loaded with other tasks. However, once sufficient cores are made available to meet the computational load, pinning provides an advantage. As can be seen, running 8 threads on a single 8 core processor with a shared last level cache performs comparably to running on 32 cores with no restrictions on thread migration.

Aside from Java’s lack of a thread pinning API, there are other weaknesses to cache management through thread pinning. This approach can be disrupted by cache

pollution. If small chunks of memory are allocated throughout the memory space, they can quickly force out the very data this approach is attempting to keep in the caches. This is often the case in Java, where many small objects can be created and discarded in a relative short time, but live until the next garbage collection. Using the VisualVM live allocated objects view, we were able to see that over 50% of our live memory was being used by one type of temporary object, a simple convenience class that wraps together three floating point values. Unfortunately, this view does not provide any information as to which thread or method was creating these objects. This particular class was ubiquitous throughout the code, representing three dimensional forces, placements, and velocities. The impact these objects were having on the cache performance was difficult to judge, as the degree of cache pollution depended on how many and which cores created the objects. Knowing which thread was using what portion of the heap would have provided insight into caching behavior and how to increase cache hits.

### C. System Topology Discovery

Another difficulty we encountered during our thread pinning efforts is one of mapping between virtual, logical, and physical processors. In the multicore era, it is becoming increasingly difficult to visualize the cache and core topology of even a two socket system. Some virtual cores share a last level cache while others do not. Some virtual cores are in fact just different hardware thread domains on the same hardware core, thereby potentially sharing lower level caches but also sharing other resources such as the execution pipeline.

For example, one of our test machines had a single processor with four hardware cores, each with two HyperThreading® domains, and a single shared last level cache (LLC). Another had two processors, each with four cores and no multithreading. Each pair of cores shared an LLC and had different memory access speeds

to other caches, depending on whether they shared data at the LLC, socket, or system level. The last machine had four processors, each with eight hardware cores running two logical cores via Hyperthreading®, for a total of 64 virtual processors. Here again, three levels of cache sharing are possible. It was difficult to determine which virtual processors shared a cache and which were primary threads or secondary threads on the same core.

A tool or API that aided in deciphering the core and cache topology of the underlying hardware would have been helpful. One such API is provided by the *hwloc* project [10]. The *hwloc* API is designed for use by parallel applications to gather information for use in such tasks as assigning threads to cores. It presents information about the system as a general-purpose tree of resources. This resource information should be made available as a key part of thread information presented in performance tools. For example, VTune’s thread to cpu graph, as shown in Figure 2, could be annotated with this information. Without this type of topological insight, it is very difficult to understand observed performance or discover computation or data distribution problems such as pinning two threads to the same physical core inadvertently.

## VI. RELATED WORK

The performance analysis study presented in this paper relates to various approaches for parallelizing and optimizing Java programs and other Java parallelization case studies.

### A. Java Parallelization Case Studies

Other parallel case studies in Java have focused on regular applications, whereas the molecular dynamics application we analyze is an irregular application. In general, they also rely on modifications to the Java virtual machine, rather than focus on application-level code. Li et al. [11] parallelized an image processing application using a master/worker, single work queue strategy similar to the one we use. They investigated the cache behavior as the number of threads increased and found that blocking the image data resulted in a 5-10% performance improvement. Li et al. also investigated improving performance through thread affinity (i.e., pinning threads to specific processors), but did not find that it had an effect on performance. However, their regular application experienced much less thread switching than we saw in the irregular application (see Figure 2 and in [11] see Figure 7), therefore it is possible that thread affinity scheduling might have more impact in irregular applications.

Gravvanis et al. [12] parallelize a preconditioner for sparse matrices that have arrow-type non-zero structure,

which is somewhat irregular, but not as irregular as general molecular dynamics simulations. They also found thread migration to be a problem. Another issue they encountered was the need to write their own barrier to enable finer-granularity parallelism. We worked around a similar issue in MW by fusing some of the loops in the force computation to increase the granularity.

Data locality improvements in Java have also been investigated within the context of garbage collectors [13]. Data locality improvements in the garbage collector would be complementary to any application-level data reordering. Other work [14] has developed techniques for profiling regular memory accesses in Java and then modifying the array layout for the bytecode. The techniques they present such as array interleaving would be complementary to any inspector/executor strategies that reorder the data at the application level.

### B. Performance Analysis of Java Applications

Mytkowicz et al. analyzed the accuracy of Java code profilers [15] and found that the different tools are inconsistent in identifying hot methods or sections of code. This is due to sampling the call stack primarily at yield points in the code and a lack of random sampling. These effects can compound with the sampling granularity issues we observed and make it difficult to find load imbalances

## VII. CONCLUSIONS

In this paper, we relate our experiences parallelizing a molecular dynamics application written in Java. For some workloads, the parallelized application enjoyed good scalability. On a quad-core machine, we were able to achieve a refresh rate as high as 32 updates per second on benchmarks with nearly one thousand atoms. Unfortunately, one common case exhibited very little speedup, prompting an in-depth analysis of performance. We used several popular performance analysis tools to examine two common problem areas for irregular scientific applications, namely load imbalance and memory behavior.

We found that visualizing load imbalance is very difficult with existing tools. These tools lack sufficiently fine granularity to expose small imbalances. They also impact the performance of the program under test to a significant degree. When load imbalance is found, it is difficult to correlate the imbalance back to particular code.

We also attempted to improve the memory performance of Molecular Workbench. In this effort, we confronted several challenges posed by the Java virtual machine environment. We could not explicitly position data into specific memory locations. We also attempted to improve temporal locality in caches by assigning

related tasks to use the same thread throughout program execution. In this effort, we were blocked by Java's inability to bind threads to cores. Our experiments with pinning threads through use of the JNI showed that exploiting the cache hierarchy can have a large impact on performance.

We find that better tools are needed to help performance programmers deal with the new degrees of multicore complexity. Future tools need to build on their single-threaded strengths, but do so in a less timing-intrusive manner. They need to enable the user to be aware of the system topology and any potential performance impacts, positive and negative, this topology might impose. Also, they need to present previously available data in new ways, combining data about threads, cores, and caches in intuitive ways. Information about all cores in a system, the code executing on them, and their impact on the memory subsystem, needs to be delivered to the programmer in a unified and comprehensible manner.

#### ACKNOWLEDGEMENTS

The authors thank Christie Williams and Jeshua Bratman of Colorado State University for their earlier work on Molecular Workbench parallelization. We also thank Dr. Qian Xie and Dr. Robert Tinker of the Concord Group for insightful consultations and help with this project. Special thanks are extended to Intel Corporation for the use of its Manycore Testing Laboratory. This work is partially sponsored by NSF CAREER grant CCF 0746693.

#### REFERENCES

- [1] A. Bhatlele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kalé, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *Proceedings of the IEEE International Conference on Parallel and Distributed Processing (IPDPS)*, 2008.
- [2] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 02 2008.
- [3] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles, "Millisecond-scale molecular dynamics simulations on Anton," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. New York, NY, USA: ACM, 2009, pp. 1–11.
- [4] R. Tinker and Q. Xie, "Applying computational science to education: The molecular workbench paradigm," *Computing in Science and Engineering*, vol. 10, no. 5, pp. 24–27, 2008.
- [5] C. Xie, "Molecular dynamics for everyone: A brief introduction to the Molecular Workbench software," The Concord Consortium, 2007.
- [6] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. New York, NY, USA: McGraw-Hill, 1981.
- [7] T. Darden, D. York, and L. Pedersen, "Particle Mesh Ewald: An  $n \log(n)$  method for Ewald sums in large systems," *J. Chem. Phys.*, vol. 98, no. 12, pp. 10 089–10 092, 1993.
- [8] B. Tinker, B. Berenfeld, and C. Xie, "Molecular workbench," <http://mw.concord.org/modeler/>.
- [9] C. Xie, personal communication, June 2009.
- [10] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa Italie, 02 2010. [Online]. Available: <http://hal.inria.fr/inria-00429889/PDF/main.pdf>
- [11] W. Li, E. Li, R. Meng, T. Wang, and C. Dulong, "Performance analysis of Java concurrent programming: a case study of video mining system," in *The 8th International Workshop on Java for Parallel and Distributed Computing*, April 2006.
- [12] G. A. Gravvanis, V. N. Eptropou, and K. M. Giannoutakis, "On the performance of parallel approximate inverse preconditioning using Java multithreading techniques," *Applied Mathematics and Computation*, vol. 190, no. 1, pp. 255 – 270, 2007.
- [13] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "The garbage collection advantage: improving program locality," in *In Proceedings of OOPSLA*, vol. 39. New York, NY, USA: ACM, 2004, pp. 69–80.
- [14] F. Li, P. Agrawal, G. Eberhardt, E. Manavoglu, S. Ugurel, and M. Kandemir, "Improving memory performance of embedded Java applications by dynamic layout modifications," in *The 6th International Workshop on Java for Parallel and Distributed Computing*, April 2004.
- [15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," *SIGPLAN Not.*, vol. 45, no. 6, pp. 187–197, 2010.