An Evaluation of Threaded Models for a Classical MD Proxy Application

Pietro Cicotti
San Diego Supercomputer Center
University of California, San Diego
pcicotti@sdsc.edu

Susan M. Mniszewski
Los Alamos National Laboratory
smm@lanl.gov

Laura Carrington
San Diego Supercomputer Center
University of California, San Diego
lcarring@sdsc.edu

***Abstract*—Exascale systems will have many-core nodes, less memory capacity per core than today's systems, and a large degree of performance variability between cores. All these conditions challenge bulk synchronous SPMD models in which execution is typically synchronous and communication is based on buffers and ghost regions.**

We explore the design of a multithreaded MD code to evaluate several tradeoffs that arise when converting an MPI application into a hybrid multithreaded application, to address the aforementioned constraints of future architectures.

Using OpenMP and PThreads, we implemented several variants of CoMD, a molecular dynamics proxy application. We found that in CoMD, duplicating some of the work to avoid race conditions is an easier and more scalable solution than using atomic updates; that data allocation and placement can be controlled to some extent with a hybrid MPI+threads approach, though an explicit NUMA API to control locality may be desirable; and finally that dynamically scheduling the work within a process can mitigate the impact of performance variability among cores and preserve most of the performance, especially when compared to bulk synchronous implementations such as the MPI reference.

Keywords—computer architecture, parallel processing, parallel programming, performance analysis, parallel algorithms, multithreading.

I. INTRODUCTION

At Exascale, the number of cores on a chip will likely be in the hundreds [1], with an expected 100x increase in parallelism on chip [2], and possibly thousands in a single shared-memory unit. One consequence is that the memory capacity per core will decrease, with density scaling and cost being the primary limitations of DRAM capacity. As a result, the memory capacity per core is expected to be 10 times less than it is in current systems [2]. Data movement will also be constrained, to conserve both performance and power. Together, all these conditions challenge the SPMD models in which communication is often synchronous, and based on buffers and ghost regions.

Another characteristic of Exascale systems will be performance variability among cores. Future Near-Threshold Voltage (NTV) processors will exhibit variability in performance and fluctuations in leakage and temperature [3], which combined with the dynamic management of power and thermal limits, along with error recovery delays, will create a computing environment with dynamically changing performance. This sort of variability will have a severe impact on coarse-grain bulk-synchronous execution models, which are prevalent in today's HPC applications.

These variability challenges are better addressed in multithreaded dynamic execution models, in which data and work can be easily shared and scheduled between processing elements [4-6]. Multithreaded execution models present several opportunities in sharing data and work. As an exploratory co-design effort, we take into account the aspects of future systems discussed above, as the motivation to embed multithreading and dynamic scheduling into an MPI application, and then compare different algorithmic formulations to evaluate features, such as atomic floating operations, and the ability to cope with variable core speed.

In this work, we consider multithreaded programming models for CoMD, a molecular dynamics proxy application [7]. We first characterize the performance of the MPI reference implementation, and then explore the challenges and the benefits of the multithreaded implementations. Finally, we compare how the variants adapt on a system in which core speed is not uniform, in an attempt to mimic the dynamic environment of future Exascale systems.

The primary contributions of this work are:

- a characterization of CoMD showing the memory footprint composition and the timing breakdown when scaling the number of cores,
- several multithreaded implementations and a comparison of how different algorithms (e.g., force symmetry vs. work duplication) and implementations (e.g. mutexes vs. atomic operations) affect the performance,
- and an evaluation of the ability of the different variants to adapt dynamically to resource-imbalance which we emulated by varying the duty cycle of processor cores.

The paper is organized as follows. Section II provides some context and a brief overview of related work. Section III describes the algorithm of CoMD, whereas details of the implementations are given in Section IV. The experimental results are presented in Section V. Finally, concluding remarks and future work are given in Section VI .

II. BACKGROUND AND RELATED WORK

Exascale systems present several challenges for both computer and software architects. Addressing these challenges requires a holistic approach. The Department of Energy (DoE) established three Co-Design centers to ensure

that its target workloads can take advantage of future architectures [8].

The Co-Design center for material in extreme environments (ExMatEx) [9] developed a number of proxy applications to explore new algorithms, and to provide architects and system developers with a *functional description* of their workloads.

Molecular Dynamics (MD) is widely used in material science, chemical physics, and the modeling of biomolecules. MD is a computer simulation technique for modeling the physical movement of interacting atoms by numerically solving Newton's equations of motion. Forces between atoms are defined by molecular mechanics force fields or potentials.

CoMD is a proxy application developed to represent classical molecular dynamics workloads [10, 11]. CoMD is based on the MD codes, ddcMD [12] and Scalable Parallel Short-range Molecular Dynamics (SPaSM) [13], which have been used to simulate phenomena at unprecedented size and time scale. CoMD is part of the Mantevo Suite 1.0 [14], which received the 2013 R&D 100 award. MPI and OpenMP versions of CoMD are publicly available [15]; OpenCL and CUDA implementations have been developed for GPUs along with a version for the Intel Xeon Phi processor.

Other researchers have explored several programming models using LULESH [16], a shock hydrodynamics proxy application. The comparison presented and evaluated several programming models on productivity, performance, and ease of optimization. While our paper is focused on a detailed characterization of performance, we address the comparison with other emerging and high level programming models in Section VI. In addition, we look at how the programming models will adapt to future Exascale environments.

III. CoMD

CoMD computes short-range forces between atoms placed in a face-centered-cubic (FCC) lattice. The forces are evaluated between pairs of atoms whose distance is within a cutoff distance. The resulting forces are subsequently used to update atoms' velocity and position via numerical integration.

There are three main computational phases in CoMD that update force, velocity, and the position of atoms. Velocity and position updates are embarrassingly parallel: each update depends on a single state variable and can be done on a single atom in isolation (i.e. the velocity of an atom is updated according to the force on that atom, and its position is updated according to its velocity).

The force computation is the most time consuming phase and it is therefore the focus of most optimizations. A brute force search for neighboring atoms requires N^2 distance calculations (N distances for each of the N atoms) to determine which atoms fall within the cutoff distance; such an approach is extremely inefficient. In order to identify suitable atom pairs, CoMD uses link-cells [17]. The space is partitioned by applying a regular rectangular decomposition. The partition has the largest number of cells such that each cell exceeds in size the cutoff distance, in every dimension; in this way, the neighbors of an atom need only be searched within 27 cells (the cell containing the atom and the 26 neighboring cells). By using link-cells, the computational complexity is reduced to linear in N, since the number of atoms per link cell is essentially bounded.

Figure 1 illustrates the phases described, the data flowing between them, and some of the high-level routines in CoMD. In addition, the figure shows that after the position of the atoms is updated, a global redistribution routine updates the content of the cells, both locally on a process and remotely via ghost cells exchange (more details about the communication pattern are provided in Section IV.A).

For the force computation two interatomic force models are available: the Lennard-Jones (LJ) two-body potential and the many-body Embedded-Atom Model (EAM) potential. The LJ potential is included for comparison and is a valid approximation for constant volume and uniform density. The EAM potential is a more accurate model of cohesion in simple metals, like copper, and includes the energetics necessary to model non-uniform density and free surfaces. In this paper we present results on the EAM potential, for sake of brevity, although similar results have been obtained on the LJ potential.

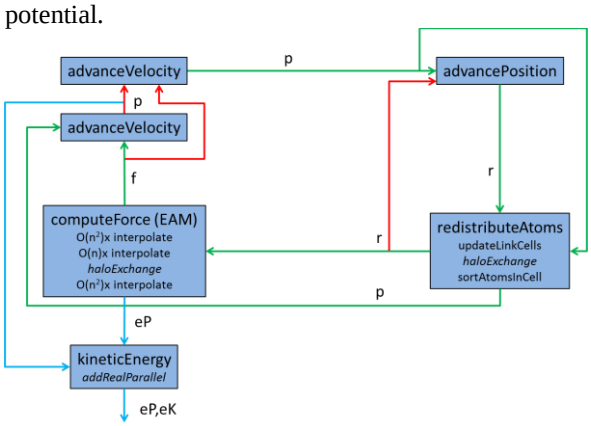

Figure 1: CoMD data-flow graph with EAM potentials. The variables p, f, r, eP, and eK represent momentum, force, position, potential energy and kinetic energy data. Green arrows indicate current timestep data-flow, red arrows indicate data-flow from one timestep to the next, and blue arrows indicate data transfers that occurr periodically.

Since the EAM potential cannot be calculated as the sum of pairwise potentials, the calculation requires three passes. The first pass computes the pairwise contribution and the electron density, the second pass computes the embedding energy and its derivative, and the final pass adds the embedding energy contribution to the force.

Forces are symmetric between 2 atoms and need only be calculated once. While this principle presents an opportunity for reducing the computational workload when the calculation is parallelized, as in most multithreaded approaches, race conditions arise if two concurrent pair evaluations try to simultaneously increment the forces and energies on the same atom. A tradeoff has to be made between managing race conditions and duplicating the force calculations.

IV. Execution Models

The reference version of CoMD is implemented using MPI and OpenMP. In this paper we evaluate the reference implementations of CoMD, as well as other variants that we

developed using OpenMP and PThreads. All the multithreaded versions are hybrid, in the sense that they preserve the MPI execution and inter-process communication model, although in this study we focus on single node performance. This section describes all the variants of CoMD that we evaluated, and their implementation.

A. MPI

The MPI version uses a 3-D spatial Cartesian domain decomposition to distribute the atoms across processors. The local domain is split into link cells assuming periodic boundaries. Atom data consists of position, velocity (momentum), force, and energy. Additional data is required for the EAM potential, such as the embedding energy and its derivative. All local atom data is stored as a structure of arrays (SoA).

A halo of link cells from replicated adjacent ranks is needed when computing local forces. Velocities and positions are advanced per link cell prior to the halo exchange. The local force computation is sequential and symmetric between atoms in each link cell and between atoms in neighboring link cells. Partial results for link cells are produced relative to the local domain atoms.

During the halo exchange, ghost atom data is communicated with neighbor ranks. Atom data is composed of updated halo link cells and migrating atoms. Atom data is exchanged per time step for LJ and EAM, while EAM requires an additional force exchange. Atom coordinates are shifted by the global size when crossing periodic boundaries. A halo exchange communication pattern consists of sends to 26 neighbor ranks in 6 messages in 3 steps: first between x-faces, then between y-faces, and finally between z-faces. This minimizes message count and maximizes message size, but requires that the message traffic be serialized, since steps must be processed in order.

B. MPI+OpenMP

There are two OpenMP implementations that we consider in this study: a reference provided by ExMatEx, and one that we implemented. Both are very simple and only differ in that the reference computes the forces between atoms in different cells twice, to avoid concurrent updates. In both, the outer loops in the force calculation are preceded by the *parallel for* directive to partition and assign the set of local cells to threads. A reduction aggregates the total energy from all the threads. In addition, our implementation uses the force symmetry optimization and avoids race conditions preceding updates to the force data with the *atomic* directive.

C. MPI+Pthreads

The PThreads version is similar in design to the OpenMP implementation: a master thread partitions and assigns the set of local cells to worker threads. Since the iteration space does not change between loops (all the outer loops iterate over the local cells), partition and mapping are static; the master thread sets the *body* function and signals the start of the parallel phase. As with OpenMP, for PThreads we compare two variants: one that uses force symmetry and one that doesn't.

PThreads is a low-level explicit model that gives more control than OpenMP on implementation details (of course at the cost of more lines of code and added complexity). In this work it enabled comparisons between locking mechanisms, partition strategies, and scheduling policies. In order to evaluate the cost of concurrent updates we implemented the update operation in different ways: guarded by a *mutex* or a *spinlock* (a non-yielding mutex), or as *pseudo-atomic* operation. The x86 instruction set does not include atomic floating point operations, and the atomic update is implemented using a 64bit *test_and_swap* on the value of the variable. First the variable is copied into a temporary, then a copy of the temporary is updated, and finally the copy and the temporary are used in the *test_and_swap* to replace the variable; if the operation does not succeed (the value has changed in the meantime) the entire sequence needs to be repeated.

A simple division of the work based on the cell ids may not always result in an optimal mapping. The ids of local cells are assigned using a simple enumeration in the three dimensions. The local space is scanned in all the dimensions by moving along one dimension until the boundaries are encountered, and then wrapping around and incrementing the next dimension, and so on. For example, if the number of cells is divisible by the number of threads, the space is partitioned along a single dimension (e.g. divided into identical planes with a certain thickness). In order to experiment with different partitions we implemented a variant that takes parameters to define how to partition the space within a process and assign cells to threads, similar to the way the space is decomposed and assigned to different MPI processes.

D. MPI+ Dynamic Pthreads

Finally, we developed a PThreads variant that implements a work-stealing policy. In this variant, a bitmap is used to represent the local cells. As before, each worker has a range of cells assigned, and starts working on its set of cells. As work progresses, threads flip bits in the bitmap accordingly, to indicate work that has been completed or is in progress. Then, when a thread runs out of work, it starts scanning the bitmap in search of more work, and when available cells are found, that work is claimed. After a complete scan the threads go to sleep waiting for a new parallel phase.

V. EVALUATION

In this Section we compare the reference MPI version of CoMD to the multithreaded versions developed. The comparison is limited to single node runs, to focus on multithreading and investigate the ability of CoMD to adapt to variability of core performance within a node.

For the experiments we used a dual processors node with Intel Xeon processors with 8 cores each (Sandy Bridge E5-2670), and running at 2.6 GHz. The node has 64GB of DDR3-1333 memory.

In most cases we used as benchmark problem the default size of a 20x20x20 lattice, as starting point on one core, and scaled the problem size accordingly. Since the computation is extremely regular, we only executed 30 timestep in weak scaling experiments, but we used the default 100 timestep in strong scaling to avoid runtimes that were too short. The few

exceptions to this configuration are noted in the respective sections.

A. Baseline Performance

As a first step of the evaluation we characterized the reference MPI implementation from a memory usage and basic performance perspective. Figure 2 shows the memory footprint when scaling from 1 to 64 cores, and shows also how the memory space is divided between data representing the simulated system and that is strictly necessary for the computation (i.e. local cells, interpolation tables, and global data), and data that is essentially overhead introduced in the implementation (i.e. halo cells, halo exchange, EAM exchange).

The first observation is that the serial version of the reference has the same exact structure of the parallel MPI version, and therefore has the same memory footprint. The only difference is that in the serial version send and receive primitives are replaced by memory copies. This is interesting as it shows that after all, ghost cells are useful also in the serial representation of the code as a way to simplify the computation on the boundaries. Nevertheless, the space overhead is high even on one core, and when scaling to 64 cores it approaches 90% of the total memory footprint. In weak scaling the space overhead does not change because the same memory footprint is allocated by each process, with exactly the same breakdown since all the data structures are equally allocated whether the core is sequential, running on 1 core, or more, and it is simply replicated as the core count increases. With multithreading, however, the overhead can be reduced significantly, by virtue of a better surface to volume ratio and because there is no need to facilitate communication between threads and replicating the data.

CoMD scales weakly with no loss of performance. As shown in Figure 3, it also scales strongly with great efficiency. It is therefore plausible to run at the higher core counts with great efficiency but exhibit the large memory overhead demonstrated. A multithreaded approach has the potential to mitigate that effect and reduce the space overhead.

Finally, by looking at the time breakdown per function, we notice how even when strong scaling, the force computation dominates the computing time, even when scaling to multiple nodes (experiments scaling to 16k cores and with fewer atoms per core showed that the force computation is still responsible for 80% of the total running time). In the remainder of the paper, we focus on the force computation.

B. Performance Comparison

Next we compare the MPI version to hybrid MPI multithreaded versions. We developed multiple versions to explore the cost trade-offs between using force symmetry and computing forces twice. In the former case less work is done but with the overhead of dealing with race conditions (e.g. omp and pt); in the latter case the force computation is duplicated but there are no race conditions to deal with (e.g. ompd and ptd). Figure 4 and Figure 5 show the time spent computing forces in weak and strong scaling experiments. In these experiments, the PThreads version (pt) uses force symmetry and avoids race conditions by guarding the force

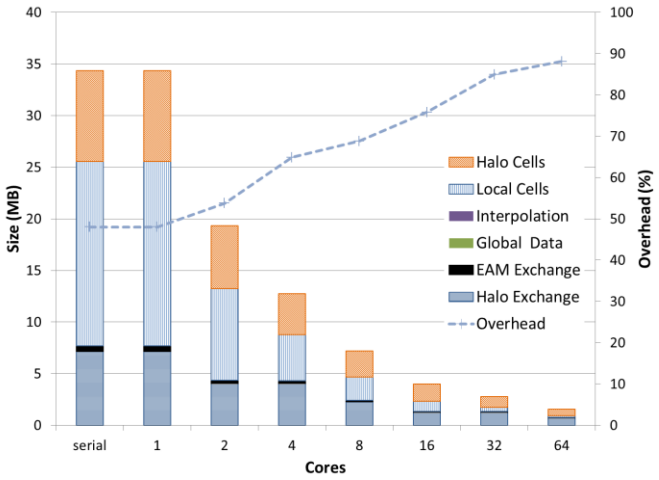

Figure 2: Break down of the memory footprint of each process, in strong scaling.

updates with locks (mutex).

The threaded versions achieve a lower performance than the reference MPI version, and that is primarily for the reasons discussed above: either the overhead of managing race conditions or the extra work done. By looking at the OpenMP versions we see that there is little difference between the two strategies, although it appears that as the number of threads increase, so does the overhead of the atomic updates and it can become a significant bottleneck on a larger number of cores. With PThreads the difference is even greater between the two versions; the atomic updates using mutexes incur a very high overhead and it gets significantly worse at 16 threads, when the threads involved communicate across sockets. Going from 8 to 16 cores, the multithreaded versions lose some performance due to the cost of inter-socket communication; this effect is addressed in Section V.D.

When duplicating the force calculation, the PThreads outperforms the OpenMP version. In PThreads the threads operate on an explicitly defined set of cells, and force duplication is necessary only in interactions between cells owned by different threads. This advantage diminishes in strong scaling, as the surface-to-volume ratio decreases and an increasingly large fraction of forces are computed twice.

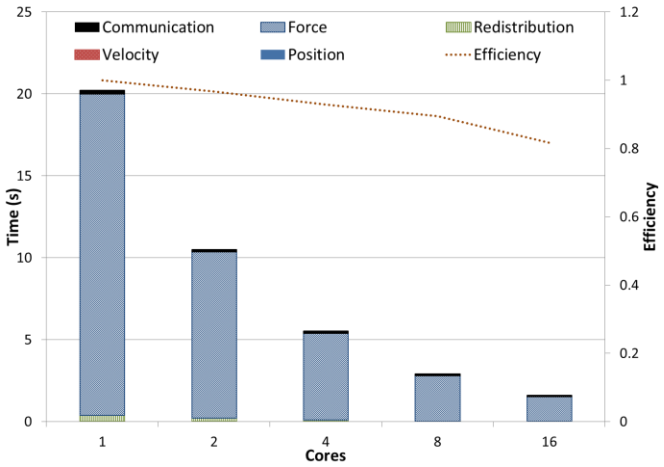

Figure 3: Strong scaling performance and timing breakdown.

In Section V.E we address this issue improving the surface-to-volume ratio.

C. The Overhead in Concurrent Updates

In order to further investigate the overhead in concurrent updates, we compared the performance of the PThreads using multiple implementations of the concurrent updates. Figure 6 shows the execution time for these versions and for reference the MPI version and the PThreads version with no force symmetry. In addition, we also compare to a version that ignores race conditions altogether (nolock), and although it produces incorrect results, it provides an additional comparison point showing what would be the performance of a zero-overhead atomic update.

The cost of locking is directly reflected in the computation time. Up to 4 cores, the performance of *nolock* and *pdt* is on par with the reference, then coherency and sharing costs create a performance gap (this effect is addressed in Section V.D). The other PThreads versions can be ranked by the cost of the mutual exclusion mechanism used to implement the atomic updates, with mutexes being the worse, and the *pseudo-atomic* instruction implementation being the best. In the latter case, the overall overhead of the atomic instruction ranges from 23% to 27% of the force computation time (with a 25% average) when compared to the *nolock* version. Even assuming a zero-overhead atomic instruction shows no tangible benefit with respect to not using force symmetry. After all, in the PThreads implementation only a fraction of the forces are actually computed twice, and just preserving cache coherency seems to have a high enough cost to offset the benefit of doing less work. Duplicating some of the work and avoiding concurrent updates entirely appears to be an easier and more scalable solution.

D. The Penalty of Non Uniform Memory Access

In the previous Sections we observed in several occasions that there is a performance penalty in using both processors. Each processor controls the four DRAM channels connected to the socket, and that form a NUMA node. To share data and implement the coherency protocol, the memory controllers communicate via the Quick Path Interconnect (QPI). In contrast, when all the threads of a process execute on a single processor there is no traffic due to coherency, or data sharing across the two NUMA nodes; trivially, for a pure MPI implementation that is always the case.

A simple way to avoid this issue is to execute a process per socket. The speedup achieved varies depending on the size of the problem and the version of the code, but in all cases there is a significant improvement. Table 1 shows the speedup achieved when running with one process per processor (PTD Socket) instead of with one process per node (PTD Node). For both problem sizes there is a substantial performance improvement. Nevertheless, on 16 cores the PThreads implementation is still slower than the MPI reference (approximately a 10% increase in time).

Alternatively, to avoid traffic due sharing, the memory allocation and data initialization phase should be carefully designed to ensure that data structures are mapped to NUMA minimizing intra-socket traffic, which is the case if threads access only data in the local NUMA node.

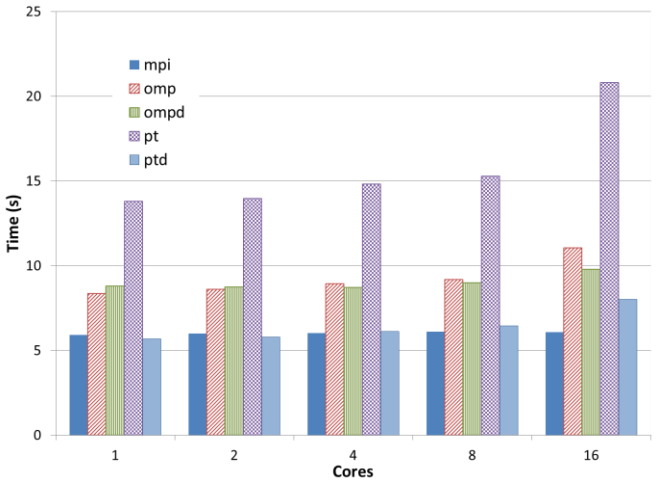

Figure 4: Weak scaling comparison between MPI, OpenMP, and PThreads versions.

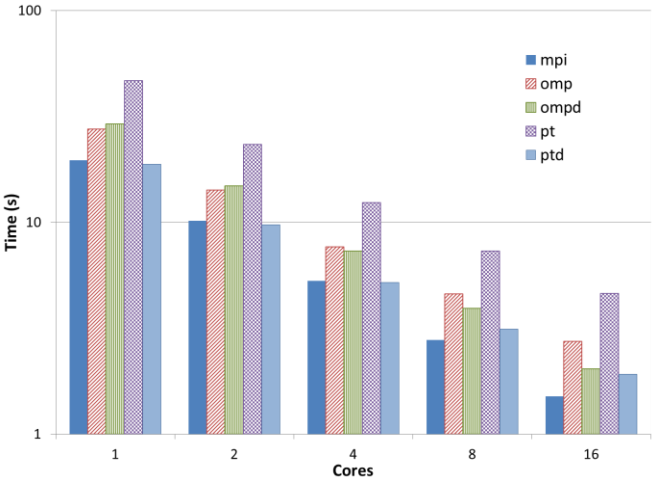

Figure 5: Strong scaling comparison between MPI, OpenMP, and PThreads versions.

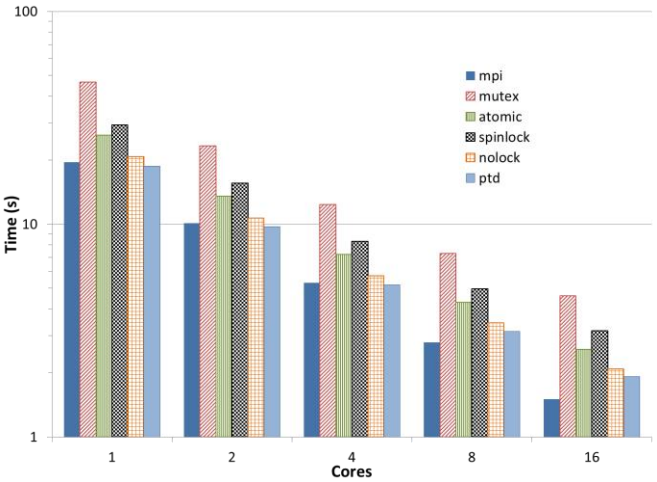

Figure 6: Strong scaling comparison between PThreads versions using different implementations of the atomic updates.

However, this approach adds great complexity to the code, especially if work is scheduled dynamically; in this case a static placement would not be a viable solution.

Table 1: Performance comparison between one process (16 treads) and two processes (8 threads per process) on 16 cores.

Lattice	MPI	PTD Node	PTD Socket	Speedup
88x44x44	7.95	9.99	8.75	1.14
22x22x22	1.87	2.55	2.04	1.25

E. The Impact of the Surface-to-Volume Ratio

The PThreads version that achieved the best performance used force duplication (ptd). The amount of work in this version is comparable to the amount of work done in the MPI version. In both cases, the forces are computed twice on the boundaries of the space assigned to a process or a thread. However, while the MPI decomposition is controlled by parameters (the processor grid that defines the space decomposition is specified at startup), the mapping of cells to threads is not. In ptd, cells are assigned to threads by id ranges, which results in sub-optimal mapping. The results of controlling the mapping to threads are shown in Table 2; the effect of the improved mapping is reported as 1.05 times speedup. The ptd version with *blocked* decomposition and mapping is comparable in performance to the MPI version, with just a 5% increase in computation time. That 5% overhead is essentially the cost of managing threads, scheduling the work, and maintain coherency within the processor.

Table 2: Performance comparison between simple and blocked mapping, on 16 cores.

Lattice	MPI	PTD Simple	PTD Blocked	Speedup
88x44x44	7.95	8.75	8.34	1.05
22x22x22	1.87	2.04	1.95	1.05

F. Adapting to Performance Variability

As we head towards Exascale technology, factors such as feature size scaling, voltage scaling, power capping, and heat management will create imbalance in core speed, even within the same chip. It is the anticipation of this imbalance that has spawned a large growth in programming models and runtime systems that adapt to this imbalance through dynamic scheduling. To explore this capability we extended the ptd version with dynamic scheduling.

The dynamic scheduling is implemented using a map of cells, to keep track of the progress made by each thread, and to give faster threads the opportunity to *steal* work from slower threads. The granularity of the map determines the size of the chunks of work managed by the map, which is defined in number of cells to update.

To simulate uneven core speed we use clock modulation. With clock modulation, it is possible to arbitrarily slow down one or more processor cores by reducing the duty cycle [18]. First, we evaluate the cost of dynamic scheduling as a function of the granularity with the expectation that finer granularity enables better balancing. Next we investigate the effect of granularity and its ability to adapt to changes in core speed, and finally we compare the ability of the different versions (e.g. MPI, OpenMP, PThreads) to adapt to core speed changes.

The first experiment was in determining the overhead of dynamically scheduling work and how that overhead varies with varying work granularity. With dynamic scheduling, threads actively acquire a portion of the cells to update and maintain a shared data structure representing the status of the cells (i.e. the map of cells); access to the map is synchronized in the case of race conditions. The ptd version with static and dynamic scheduling was run on 16 cores with increasingly coarse granularity. The overhead was calculated as the increase in the runtime of the dynamic scheduling version versus the static scheduling version. The overhead is shown in Table 3. As expected, the overhead depends on the granularity, but a comparison across the two problem sizes indicate that it is not just an issue of a constant cost in accessing the scheduling structures, otherwise in the large problem size the overhead should be a small fraction of the computing time as seen in the smaller problem size. What we observe is the combination of that and the effect of increased work due to smaller blocks of work with higher surface-to-volume ratio than in the static version.

The overhead is also dependent on the number of operations required by updates (although not shown here, with the LJ potentials, which is less compute intensive, we measured higher overheads than in the EAM potentials). More complex potentials with a larger number of terms would experience a lower overhead.

Table 3: Analysis of overhead as a function of granularity in work decomposition, on 16 cores.

Lattice	Cells	Overhead (%)
88x44x44	128	22.5
88x44x44	256	21.5
88x44x44	512	6.8
88x44x44	1024	0.6
22x22x22	1	32.5
22x22x22	8	20.2
22x22x22	16	16.8
22x22x22	32	15.3
22x22x22	64	4.4

Finer granularity enables a more balanced distribution of the work, but it also incurs higher overhead in scheduling a larger number of small chunks of work. In the next experiment we looked at how the granularity affects the code's ability to adapt to imbalances in the core speeds. For the entire execution, one of the 16 cores was configured to run with a reduced duty cycle. We explored a duty cycle range from 100% (no variation) down to a 25% duty cycle. In the experiment we also examine the effects at different granularity. Figure 7 illustrates how the force time is affected by core speed imbalance and how granularity affects the ability to rebalance the workload. By looking at the timing of the force calculation (bars), we see that the time increases when the duty cycle decreases, but very slowly. Looking at the slowdown (lines indicating a speedup lower than 1), we see that even at the coarsest granularity the slowdown is small for a running one core at 25% of its speed. Finer granularities are more effective and enable a better load balancing, but also incur higher overheads; we observe that the higher overhead is not compensated by a better balancing

and the coarsest granularity always results in faster execution.

In the next experiment we examine the ability of the different versions to adapt to core frequency changes. We compared three versions of the code: MPI, OpenMP (executing with *dynamic* scheduling), and PThreads. The OpenMP version was executed enabling dynamic scheduling to adaptively assign portions of the iteration space to threads; the chunk size was to set 64 which corresponds to the work granularity in the ptd. Figure 8 shows both the time (bars) and the slowdown (lines) of the force calculation. The MPI reference implementation is marginally faster than the ptd if the cores speed is about the same, but as the difference exceeds the 12.5% threshold, the PThreads version is faster, and the gap grows proportionally to the difference in core speed between the slow core and the others. As seen in the speedup, the MPI reference does not rebalance the workload and is synchronous, and therefore it reflects the speed of the slowest core. The OpenMP version is inferior in performance compared to the PThreads version; however, the OpenMP version does balance the work although not as effectively as the PThreads version.

To conclude, we tested with all the cores at variable speed. A process controls the duty cycle of each core and periodically wakes up and changes the duty cycle of all the cores, by one step (6.25%), randomly choosing the direction (increment or decrement). The range of the duty cycle is set from a user selected minimum duty cycle, to 100%. Initially all cores execute at the speed in the middle of the range; then, every second the process wakes up and changes the duty cycle of all cores; since the changes are random, the average tends to stay in the middle of the range. For the experiments we selected ranges with the minimum at 87.5%, 75%, 50%, and 6.25%.

The OpenMP version executed with two different scheduling policies: dynamic (static chunk size of 64 cells, which is labelled omd) and guided (starting from a chunk of size 64 and then decreasing the size to fine tune the balancing, which is labelled omg). The results are shown in Figure 9, together with the average and average minimum duty cycle, showing an approximation of the ideal and the worst case slowdown.

With all cores running at lower speeds it is no surprise that the performance degrades significantly. The omg version had ideal balancing for the first two settings, but it degraded rapidly at lower duty cycle settings, whereas omd has a more consistent behavior and ends achieving the best balance

VI. Conclusions and Future Work

Exascale systems will have many-core nodes but the increase in core count will not be matched by the increase in memory capacity. In addition, the performance of individual cores will vary, creating load imbalance even in regular problems. In this paper, we explore some the benefits and challenges in implementing a multithreaded molecular dynamics proxy application to address the aforementioned issues. We focused on tradeoffs between atomic operations and work duplication, and between work granularity load balance. In addition, we tried to address the performance issues that may arise in sharing data within an address space.

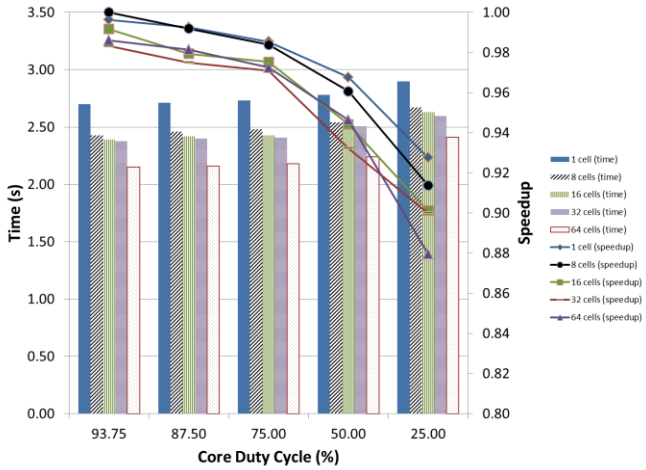

Figure 7: Time and speedup in the force calculation at different granularities for a 22x22x22 lattice on 16 cores. The duty cycle is changed only on one core.

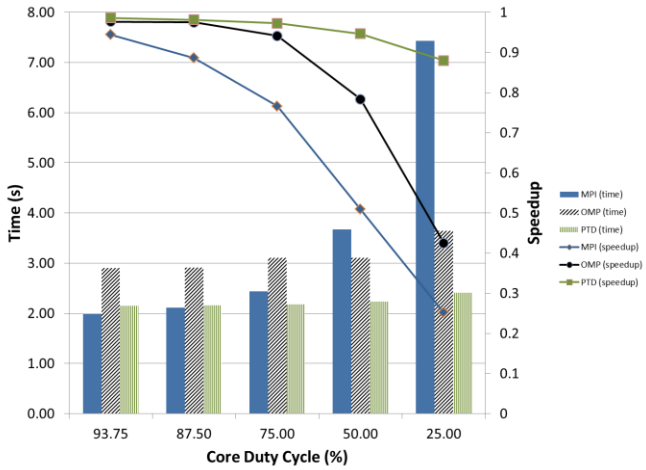

Figure 8: Time and speedup in the force calculation for different versions of the code for a 22x22x22 lattice on 16 cores. The duty cycle is changed only on one core.

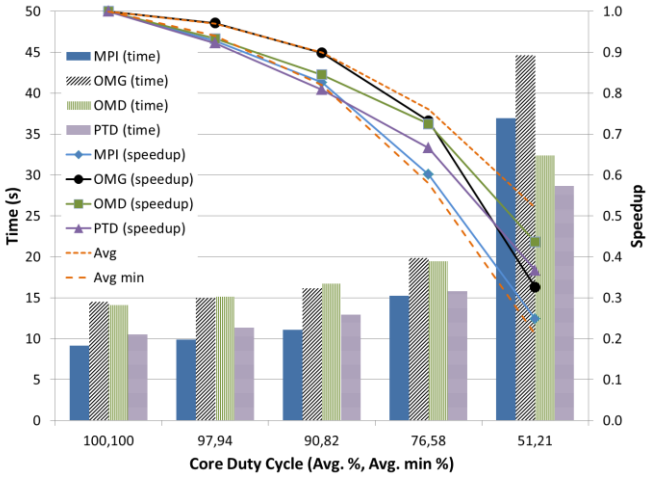

Figure 9: Time and speedup in the force calculation for different versions of the code for a 22x22x22 lattice on 16 cores (500 timesteps). The duty cycle is changed periodically for all the cores.

In CoMD, it is tempting to try and reduce the amount of computation done (by virtue of force symmetry) and using some form of synchronized updates, but the overhead of the synchronization mechanisms limits both the performance and the scalability of this approach; we observed significant overheads even when allowing unsynchronized concurrent updates suggesting that even hardware floating atomic operations will be inferior to a non-synchronized approach. Future work will explore a scheduling system to automatically avoid race conditions while using force symmetry.

Sharing data even between threads has a cost and the cost depends on the *distance* of the computing cores, for example as we observed by distributing 16 threads on two sockets. The relative *distance* between cores will increase with deeper on-chip memory hierarchies; as a simple way to cope with traffic between NUMA node we placed one MPI process per processor; alternative solutions would be APIs to explicitly control the placement of data structures to NUMA nodes, or a runtime system that improves locality by integrating data placement with scheduling.

As the speed of cores varies, adaptive work scheduling limits the potential performance loss by preventing cores to be idle while there is work available. As the difference in cores speed increases these techniques are necessary. However, striking an optimal balance between overhead and load balance (controlled for example by the granularity of the work chunks) is not trivial. Even an adapting policy like the *guided* scheduling in OpenMP seemed to fail compared to a simpler *dynamic* scheduling with fixed size chunks of work. Especially when all the cores speed varies, achieving a good balance does require fine grain work decomposition but that will negatively affect the overhead. When the difference between the average core speed and the minimum is not large, there amount of overhead for load balancing that can be tolerated is very small and more effective work sharing mechanisms should be considered. Currently, we are investigating a CoMD version that uses the OCR runtime system [19, 20]. This study will explore new algorithms and architectural features using the runtime system and an architectural simulator, and take advantage of features of the runtime system, such as automatic load balancing and customizable scheduling policies.

Finally, while this study touches on some key aspects of multithreading on future architectures, further investigation is required to gain a deeper understanding on the issues touched, to improve the quality of the prototypes implemented (especially the scheduling and work distribution system which is admittedly a primitive implementation), and to continue the study in other directions. Other directions that will be investigated include data movement on node and between nodes, and scaling to larger number of cores and of nodes.

ACKNOWLEDGMENT

The authors would like to thank Jim Belak and David Richards for their comments and suggestions.

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717.

DISCLAIMER

REFERENCES

[1] P. Kogge, “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems,” CSE Dept. Tech. Report TR-2008-13

[2] Architectures and Technologies for Extreme Scale Computing, 2009.

[3] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, “Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits,” Proceedings of the IEEE, vol. 98, no. 2, pp. 253-266, 2010.

[4] L. Dagum, and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” Computational Science & Engineering, IEEE, vol. 5, no. 1, pp. 46-55, 1998.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: an efficient multithreaded runtime system,” SIGPLAN Not., vol. 30, no. 8, pp. 207-216, 1995.

[6] R. A. Alfieri, “An efficient kernel-based implementation of POSIX threads,” in Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1, Boston, Massachusetts, 1994, pp. 5-5.

[7] ExMatEx. "CoMD," http://www.exmatex.org/comd.html.

[8] "Scientific Discovery through Advanced Computing (SciDAC) Co-Design," http://science.energy.gov/ascr/research/scidac/co-design/.

[9] "DoE Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx)," http://www.exmatex.org.

[10] J. Mohd-Yusof, CoDesign Molecular Dynamics (CoMD) Proxy App, LA-UR-12-21782, Los-Alamos National Lab, 2012.

[11] S. Mniszewski, and JamalMohd-Yusof, “CoDesign Molecular Dynamics (CoMD) Proxy Application,” in PGI OpenACC WorkShop, 2013.

[12] D. F. Richards, J. N. Glosli, B. Chan, M. R. Dorr, E. W. Draeger, J.-L. Fattebert, W. D. Krauss, T. Spelce, F. H. Streitz, M. P. Surh, and J. A. Gunnels, “Beyond homogeneous decomposition: scaling long-range forces on Massively Parallel Systems,” in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, 2009, pp. 1-12.

[13] T. C. Germann, K. Kadau, and S. Swaminarayan, “369 Tflop-s molecular dynamics simulations on the petaflop hybrid supercomputer ‘Roadrunner’,” Concurr. Comput. : Pract. Exper., vol. 21, no. 17, pp. 2143-2159, 2009.

[14] M. A. Heroux, D. W. Doefler, P. S. Crozier, J. Willenbring, M., C. H. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, Improving Performance via Mini-applications, Sandia National Laboratory, 2009.

[15] ExMatEx. "CoMD Software," https://github.com/exmatex/CoMD.

[16] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, “Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application,” in Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 919-932.

[17] R. W. Hockney, S. P. Goel, and J. W. Eastwood, “Quiet high-resolution computer models of a plasma,” Journal of Computational Physics, vol. 14, no. 2, pp. 148-158, 2//, 1974.

[18] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," Software Controlled Clock Modulation, 2014.

[19] "Open Community Runtime," https://01.org/open-community-runtime.

[20] "Traleika Glacier X-Stack Program," https://xstackwiki.modelado.org/Traleika_Glacier.