LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-457414

A Scalable Data-Privatization Threading Algorithm for Hybrid MPI/OpenMP Parallelization of Molecular Dynamics

M. Kunaseth, D. F. Richards, J. N Glosli, R. K. Kalia, A. Nakano, P. Vashishta

September 28, 2010

IEEE International Parallel & Distributed Processing Symposium
Anchorage, AK, United States
May 16, 2011 through May 20, 2011

Disclaimer

A Scalable Data-Privatization Threading Algorithm for Hybrid MPI/OpenMP Parallelization of Molecular Dynamics

Manaschai Kunaseth[1], David F. Richards[2], James N. Glosli[2],
Rajiv K. Kalia[1], Aiichiro Nakano[1], Priya Vashishta[1]

[1]Department of Computer Science, Department of Physics, Department of Material Science
University of Southern California, Los Angeles, CA 90089-0242, USA
(kunaseth, rkalia, anakano, priyav)@usc.edu

[2]Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
(richards12, glosli)@llnl.gov

***Abstract*—Calculation of the Coulomb potential in the molecular dynamics code ddcMD has been parallelized based on a hybrid MPI/OpenMP scheme. The explicit pair kernel of the particle-particle/particle-mesh algorithm is multi-threaded using OpenMP, while communication between multicore nodes is handled by MPI. We have designed a scalable data-privatization scheduling algorithm based on mutually exclusive workload partitioning, which combines: 1) fine-grain dynamic load balancing based on a greedy approach; and 2) minimal memory-footprint data privatization via memory locality-aware computation assignment. This algorithm reduces the memory requirement for thread-private data from $O(NP)$ to $O(N+P^{1/3}N^{2/3})$—amounting to 75% memory saving for 16 threads, while maintaining the average thread-level load-imbalance less than 5%. Strong-scaling speedup for the kernel is 14.43 with 16-way threading on a four quad-core AMD Opteron node.**

Keywords—Hybrid MPI/OpenMP Parallelization; Thread Scheduling; Memory Optimization; Load Balancing; Parallel Molecular Dynamics

I. INTRODUCTION

Molecular dynamics (MD) simulation is widely used to study material properties at the atomistic level. Large-scale MD simulations are beginning to address broad problems [1-6], but increasingly large computing power is needed to encompass even larger spatiotemporal scales. For example, Glosli *et al.* performed a massively parallel MD simulation involving 62 billion particles using the MD code ddcMD, which demonstrated excellent performance and scalability [7].

Due to shifting trends in computer architecture, improvements in computing power are now gained using multicore architectures instead of increased clock speed. Furthermore, the number of cores per chip is expected to continue to grow. As a consequence, the performance of traditional parallel applications which are solely based on the message passing interface (MPI), is expected to degrade substantially [8]. Hierarchical parallelization frameworks, which integrate several parallel methods to provide different levels of parallelism, have been proposed as a solution to this scalability problem on multicore platforms [5, 9, 10].

Two-level parallelization based on a hybrid MPI/threading scheme is anticipated to replace traditional MPI-only parallel MD. However, efficiently integrating a multi-threading framework into an existing MPI-only code is difficult for several reasons: 1) the highly overlapped memory layout in typical MD codes incurs serious thread-write contentions; 2) naïve threading algorithms for MD usually create significant overhead, thereby limiting the threading speedup for a large number of threads; and 3) dynamic nature of MD requires low-overhead dynamic load balancing for threads to maintain good performance [11].

To address these issues, we have designed a scalable data-privatization threading algorithm based on mutually exclusive workload partitioning, which combines: 1) fine-grain dynamic load balancing based on a greedy approach; and 2) minimal memory-footprint data privatization via memory locality-aware computation assignment. We have implemented this algorithm in ddcMD and demonstrated that the hybrid MPI/threading scheme outperforms MPI-only scheme in terms of the strong scaling of large-scale problems.

This paper is organized as follows. Section II summarizes the hierarchy of parallel operations in ddcMD. Section III describes the methodology and theoretical analysis of the data-privatization scheduler that combines two optimization techniques: Greedy load balancing and memory locality-aware work allocation. Section IV evaluates the performance of the hybrid parallelization algorithm against that of the traditional MPI-only parallel algorithm. Conclusions are drawn in section V.

II. Domain Decomposition Molecular Dynamics

Molecular dynamics simulation follows the phase-space trajectories of an N-particle system where the forces between particles are given by the gradient of a potential energy function $\phi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$. Positions and velocities of all particles are updated at each MD step by numerically integrating coupled ordinary differential equations. The dominant computation of most MD simulations is the evaluation of the potential energy function and associated forces. The computational intensity varies greatly (10^2 – 10^6 floating point operations per particle) with the physical model.

One model of great physical importance is the interaction between a collection of point charges. This interaction is a Coulomb field ($1/r$) which is long range and pair-wise, requiring $O(N^2)$ operations to evaluate. Simulations with potential evaluations requiring $O(N^2)$ operations per MD step are intractable for all but the smallest systems. However various methods exist to reduce the computational complexity. A common approach used by methods such as particle-particle/particle-mesh [12], particle mesh Ewald [13] and fast Fourier Poisson [14], is to decompose the Coulomb potential into two parts: A short-range explicit pair part that converges quickly in real space and a long-range part that converges quickly in reciprocal space. The computational split of work between the short-range and long-range part is controllable though a "screening parameter", which we call α. With the appropriate choice of α, computational complexity for these methods can be reduced to $O(N\log N)$.

In this paper we explore parallelization of the short-range part of the Coulomb potential using OpenMP threading. The short-range part is a sum over pairs with the form:

$$\phi = \sum_{i<j} q_i q_j \frac{\operatorname{erfc}\left(\alpha r_{ij}\right)}{r_{ij}}, \qquad (1)$$

where q_i is the charge of particle i and r_{ij} is the separation between particles i and j. Though this work is focused on this pair function, much of the work can be readily applied to other pair functions. In addition to this intranode parallelization, the ddcMD code is already parallelized across nodes using a particle-based domain decomposition implemented using MPI. Combining the existing MPI-based decomposition with the new intranode parallelization yields a hybrid MPI/OpenMP parallel code.

A. Internode Operations

In typical parallel MD codes the first level of parallelism is obtained by decomposing the simulation volume into domains each of which is assigned to a compute core (*i.e.*, an MPI task). Because particles near domain boundaries interact with particles in nearby domains, internode communication is required to exchange remote particle data between domains. The surface to volume ratio of the domains and the choice of potential sets the balance of communication to computation.

The domain-decomposition strategy in ddcMD allows arbitrarily shaped domains that may even overlap spatially. Also, remote particle communication between nonadjacent domains is possible when the interaction length exceeds the domain size. A domain is defined only by the position of its center and the collection of particles that it "owns." Particles are initially assigned to the closest domain center, creating a set of domains that approximates a Voronoi tessellation. The choice of the domain centers will control the shape of this tessellation and hence the surface to volume ratio for each domain. The commonly used rectilinear domain decomposition employed by many parallel codes is clearly not optimal from this perspective. The best surface to volume ratio in a homogeneous system is achieved if domain centers form a bcc, fcc, or hcp lattice, which are common high-density packing arrangements of atomic crystals.

In addition to setting the communication cost, the domain decomposition can also control load imbalance. Because the domain centers in ddcMD are not required to form a lattice, simulations with a non-uniform spatial distribution of particles (*e.g.*, voids or cracks) can be load balanced by an appropriate non-uniform arrangement of domain centers. The flexible domain strategy of ddcMD allows for the migration of the particles between domains by shifting the domain centers. As any change in their positions affects both load balance and the overall ratio of computation to communication, shifting domain centers is a convenient way to optimize the overall efficiency of the simulation. Given an appropriate metric (such as overall time spent in MPI barriers) the domains can be shifted "on-the-fly" in order to maximize efficiency [15].

B. Intranode Operations

Once particles are assigned to domains and remote particles are communicated, the force calculation can begin. Figure 1 shows a schematic of the linked-list cell method used by ddcMD to compute pair interactions in $O(N)$ time. In this method, each simulation domain is divided into small cubic cells, and a linked-list data structure is used to organize particle data (*e.g.*, coordinates, velocities, type, and charge) in each cell. By traversing the linked list, one retrieves the information of all particles belonging to a cell, and thereby computes interparticle interactions. The dimension of the cells is determined by the cutoff length of the pair interaction, R_c.

The linked-list traversal introduces a highly irregular memory-access pattern, resulting in performance degradation. To alleviate this problem, we reorder the particles within each node at the beginning of every MD step so that the particles within the same cell are arranged contiguously in memory when the computation kernel is called. In our computing environment the benefit of the

regular memory access far outweighs the cost of particle ordering.

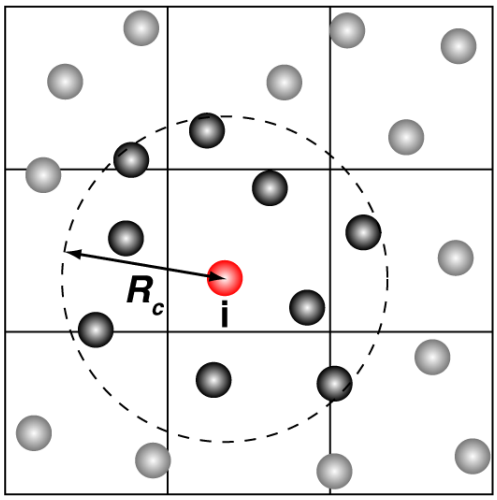

Figure 1. 2D schematic of the linked-list cell method for a pair computation with the cell dimension R_c. Only forces exerted by particles within the cutoff radius (represented by a two-headed arrow) are computed for particle i.

The computation within each node is described as follows. Let L_x, L_y, and L_z be the numbers of cells in the x, y, and z directions, respectively, and $\{C_k \mid 0 \le k < L_xL_yL_z\}$ be the set of cells within each domain. The computation within each node is divided into a collection of small chunks of work called a computation unit λ. A single computation unit λ_k corresponding to cell C_k is defined as a collection of pair-wise computations (see Fig. 2):

$$\lambda_k = \left\{ \left(\mathbf{r}_i, \mathbf{r}_j \right) \middle| \mathbf{r}_i \in C_k; \mathbf{r}_j \in \mathrm{nn}^+\left(C_k\right) \right\}, \tag{2}$$

where $\mathrm{nn}^+(C_k)$ is a set of half the nearest-neighbor cells of C_k. Due to Newton's third law, forces on a pair of particles are equal and opposite in direction. This allows us to halve the number of force evaluations and use $\mathrm{nn}^+(C_k)$ instead of the full set of nearest-neighbor cells, $\mathrm{nn}(C_k)$. The pairs in all computation units are unique, and thus the computation units are mutually exclusive:

$$\bigcap_{0 \le k < L_xL_yL_z} \lambda_k = \varnothing \tag{3}$$

The set of all computation units on each node is denoted as $\Lambda = \{\lambda_k \mid 0 \le k < L_xL_yL_z\}$. Hereafter, N denotes the number of particles in each node, and P is the number of threads in each node. Note that P is identical among all nodes.

We parallelize the explicit pair force computation kernel of ddcMD at the thread level using OpenMP. Two major problems commonly associated with threading are: 1) race condition among threads; and 2) thread-level load imbalance [8]. The race condition occurs when multiple threads try to update the force of the same particle concurrently. Several techniques have been proposed to solve these problems:

- Duplicated pair-force computation—simple and scalable, but doubles computation. Usually used in GPGPU threading [16, 17].
- Spatial decomposition coloring [18]—scalable without increasing computation, but can cause load imbalance.
- Mutually exclusive dynamic scheduling [19]—robust and suited for dynamic load balancing, but can incur considerable overhead for context switching.
- Data privatization—no penalty on computation, but with excessive $O(NP)$ memory requirement and associated reduction operation cost.

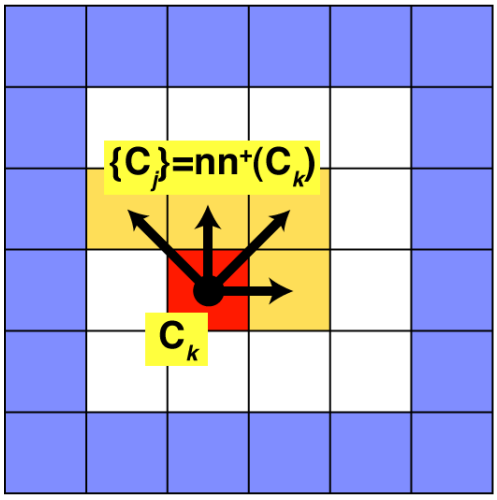

Figure 2. 2D schematic of a single computation unit λ_k. The shaded cells C_j pointed by the arrows constitute the half neighbor cells, $\mathrm{nn}^+(C_k)$.

We have designed an algorithm that combines a mutually exclusive scheduler with a reduced memory data-privatization scheme to address all of these issues. In section III, we describe a traditional data-privatization algorithm and its problems, followed by a discussion of our solutions to these problems in subsections III-A and III-B.

III. Data-Privatization Scheduling Algorithm

A traditional data-privatization algorithm avoids write conflicts by replicating the entire write-shared data structure and allocating a private copy to each thread (Fig. 3). The memory requirement for this redundant allocation scales as $O(NP)$. Each thread computes forces for each of its computation units and stores the force values in its private array instead of the global array. This allows each thread to compute forces independently without a critical section. After the force computation for each MD step is completed, the private force arrays are reduced to obtain the global forces. The reduction operation can be performed in $O(N\log P)$ time using a hypercube algorithm. Note that read conflicts do not cause any problem in this context.

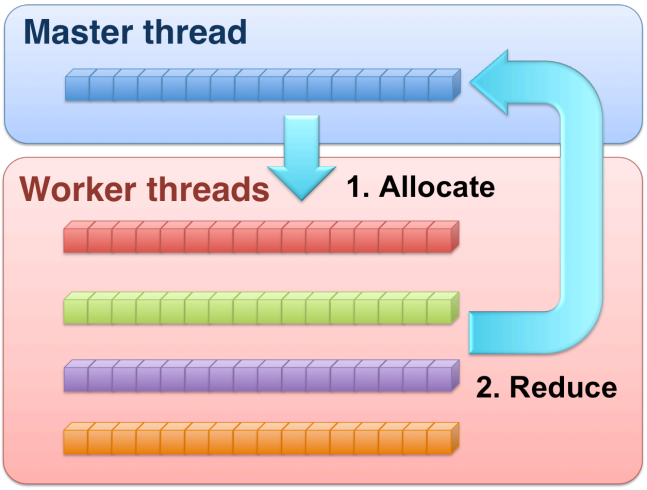

Figure 3. Schematic of a memory layout for a traditional data privatization.

To reduce the redundant memory requirement, we have developed a low-overhead approach that provides excellent load balancing while imposing minimal interference on the worker threads. Our algorithm utilizes a greedy scheduler to distribute the workload before entering the pair computation, *i.e.*, parallel section. In this approach, the scheduling cannot interfere with the worker threads since the scheduling is already completed before the worker threads are started. Because the schedule is recomputed every MD step (or perhaps every few MD steps) there is adequate flexibility to adapt load balancing to the changing dynamics of the simulation.

The hybrid MPI/OpenMP parallelization of ddcMD is implemented by introducing the thread scheduler into the MPI-only ddcMD. Figure 4 shows the workflow of the hybrid MPI/OpenMP code using the data-privatization scheduler. The program repeats the following computational phases: First, the master thread performs initialization and internode communications using MPI; the scheduler computes the scheduled workload for each thread; and the worker threads execute the workloads in an OpenMP parallel section.

Since the scheduling is performed frequently, the load-balancing algorithm needs to be simple yet provide sufficient load-balancing capability. Therefore, we have adopted a greedy approach for the load-balancing scheduler, which is discussed and analyzed in subsection III-A. In subsection III-B, the load-balancing scheduler is further enhanced by introducing the minimal memory-footprint data-privatization scheme.

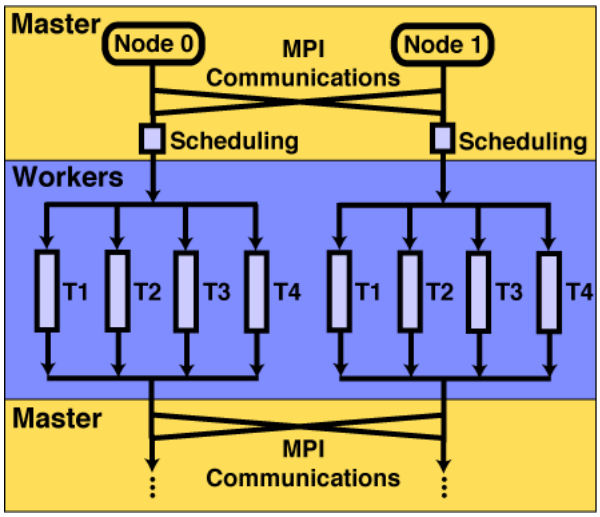

Figure 4. Schematic workflow of the hybrid MPI/OpenMP scheme. In each MD step, internode MPI communications and thread scheduling are performed by the master thread prior to the force computations, which are computed by worker threads in an OpenMP parallel section.

A. Thread-Level Load-Balancing Algorithm Based on a Greedy Algorithm

We implement thread-level load balancing based on a simple greedy approach, *i.e.*, iteratively assign a computation unit to the least-loaded thread, until all computation units are assigned.

Let $T_i \subseteq \Lambda$ denote a mutually exclusive subset of computation units assigned to the i-th thread, where

$$\bigcap_{0 \le i < P} T_i = \varnothing \quad . \tag{4}$$

The computation time spent on λ_k is denoted as $\tau(\lambda_k)$. Thus, the computation time of each thread is

$$\tau(T_i) = \sum_{\lambda_k \in T_i} \tau(\lambda_k) \quad . \tag{5}$$

The algorithm initializes T_i to be empty, and loops over λ_k in Λ. Each iteration selects the least-loaded thread $T_{\min} = \text{argmin}(\tau(T_i))$, and assigns λ_k to it. Figure 5 shows the pseudo code of this algorithm.

Algorithm Greedy Load Balancing
1. **for** $0 \leqslant i < P$ **do**
2. $\quad T_i \leftarrow \varnothing$
3. **end do**
4. **for each** λ_k **in** Λ **do**
5. $\quad T_{\min} \leftarrow \arg\min_{0 \le i < P}\left(\tau(T_i)\right)$
6. $\quad T_{\min} \leftarrow T_{\min} \cup \lambda_k$
7. **end do**

Figure 5. Greedy load-balancing algorithm.

The greedy algorithm is simple yet provides an excellent load-balancing capability. As shown below, this approach has a well-defined upper bound on the load imbalance. For a perfectly load-balanced system, the computation time of every thread is equal to the average computation time,

$$\tau_{\text{average}} = \frac{1}{P}\sum_{i=0}^{P-1}\tau(T_i) \quad (6)$$

To quantify the load imbalance, we define a load-imbalance factor γ as the difference between the runtime of the slowest thread and the average runtime,

$$\gamma = \frac{\max(\tau(T_i)) - \tau_{\text{average}}}{\tau_{\text{average}}} = \frac{\max(\tau(T_i))}{\tau_{\text{average}}} - 1 \quad (7)$$

By definition, $\gamma = 0$ when the loads are perfectly balanced. Considering

$$\min(\tau(T_i)) \le \tau_{\text{average}}, \quad (8)$$

thus,

$$\max(\tau(T_i)) - \tau_{\text{average}} \le \max(\tau(T_i)) - \min(\tau(T_i)). \quad (9)$$

Substituting Eq. (9) into Eq. (7) yields

$$\gamma \le \frac{\max(\tau(T_i)) - \min(\tau(T_i))}{\tau_{\text{average}}}. \quad (10)$$

Eq. (10) shows that reducing the difference of the maximum and minimum workloads minimizes the load-imbalance factor.

In our greedy algorithm, the workload of $T_{\min}$ is increased by $\tau(\lambda_k)$ at each iteration. Therefore, the maximum workload that can be increased is $\max(\tau(\lambda_k))$. This procedure guarantees that the variance of the workloads among all threads is limited by

$$\max(\tau(T_i)) - \min(\tau(T_i)) \le \max(\tau(\lambda_k)). \quad (11)$$

Applying the transitive relation to inequalities Eq. (9) and Eq. (11), we obtain

$$\max(\tau(T_i)) - \tau_{\text{average}} \le \max(\tau(\lambda_k)). \quad (12)$$

Substituting Eq. (12) in Eq. (7) provides an upper limit for the load-imbalance factor

$$\gamma \le \frac{\max(\tau(\lambda_k))}{\tau_{\text{average}}}. \quad (13)$$

Performance of this load-balance scheduling algorithm depends critically on the knowledge of time spent on each computation unit $\tau(\lambda_k)$. Since the runtime of the computation units are unknown to the scheduler prior to the actual computation, the scheduler has to accurately estimate the workload of each computation unit. Due to the gradual change of the particle positions between consecutive MD steps, $\tau(\lambda_k)$ remains highly correlated between the consecutive MD steps. Therefore, we use $\tau(\lambda_k)$ measured in the previous MD step as an estimator of $\tau(\lambda_k)$. For the first step as well as steps when the cell structure changes significantly (*e.g.,* redistribution of the domain centers), the workload of cell $\tau(\lambda_k)$ is estimated by counting the number of pairs in λ_k,

$$\tau(\lambda_k) \approx \sum_{C_j \in \text{nn}^+(C_k)} \text{n}(C_k)\,\text{n}(C_j), \quad (14)$$

where $\text{n}(C_k)$ refer to the number of particles in cell C_k.

B. Memory Locality-Aware Workload Distribution Algorithm

As mentioned before, the memory requirement of the data-privatization algorithm is $O(NP)$. However, since only a small subset of Λ is assigned to each thread it is not strictly necessary to allocate a complete copy of the force array on each thread. Therefore, we allocate only the necessary portion of the global force array corresponding to the computation units assigned to each thread as a private force array. This idea is embodied in a simple three-step algorithm (Fig. 6): First, the scheduler assigns computation units to threads and then determines which subset of the global data is required by each thread. Second, each thread allocates its private memory as determined by the scheduler. Finally, private force arrays from all threads are reduced into the global force array.

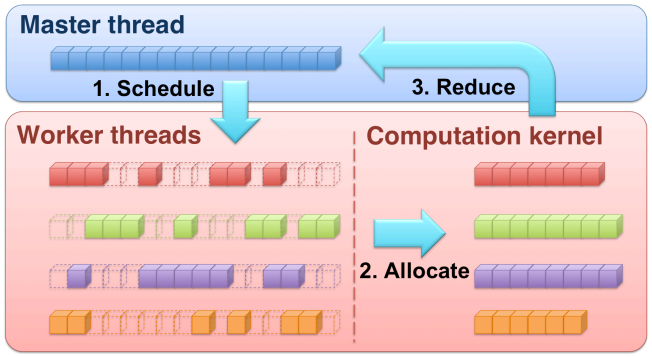

Figure 6. Memory layout and three-step algorithm for memory locality-aware scheduling algorithm.

To do this, we create a mapping table between the global force-array index of each particle and its thread-array index in a thread memory space. Fortunately, ddcMD sorts the particle data based on the cell they reside in, and thus only the mapping from the first global particle index of each cell to the first local particle index is required. The local

ordering within each cell is identical in both the global and private arrays.

It should be noted that assigning computation unit λ_k to thread T_i requires memory allocation more than the memory for the particles in C_k. Since each computation unit computes the pair forces of particles in cell C_k and half of its neighbor cells $\mathrm{nn}^+(C_k)$ as shown in Fig. 2, the force data of particles in $\mathrm{nn}^+(C_k)$ need to be allocated as well. In order to minimize the memory requirement of each thread, the computation units assigned to it must be spatially proximate, so that the union of their neighbor-cell sets has a minimal size. For this purpose, it is essential to minimize the surface-to-volume ratio of T_i.

To maintain a small surface-to-volume ratio when assigning a new computation unit to each thread, we modify the greedy scheduling algorithm introduced in section III-A. The pseudo-code of this algorithm is given in Fig. 7. First, we randomly assign a single computation unit to each thread. Then, the iteration begins by selecting the least-loaded thread $T_{\min}$. From the surrounding volume of $T_{\min}$, we select the computation unit that has the minimum distance to the centroid of $T_{\min}$, and add it to $T_{\min}$. The algorithm repeats until all computation units are assigned. If all of the surrounding computation units of $T_{\min}$ are already assigned, $T_{\min}$ randomly chooses a new unassigned computation unit as a new cluster's initial seed and continue to grow from that point.

Algorithm Memory Locality-Aware Load-Balancing Scheduler

1. $i \leftarrow 0$
2. **while** $i < P$ **do**
3. **repeat**
4. $\lambda_{rnd} \leftarrow \mathrm{random}(\Lambda)$
5. **until** $\lambda_{rnd} \notin T_{j(<i)}$
6. $T_i \leftarrow \lambda_{rnd}$
7. $i \leftarrow i+1$
8. **end do**
9. **while** $\bigcup_{0 \le i < P} T_i \neq \Lambda$ **do**
10. $T_{\min} \leftarrow \arg\min_{0 \le i < P}\left(\tau(T_i)\right)$
11. $j^* \leftarrow \arg\min_{C_j \in \mathrm{nn}(T_{\min})}\left(\left\|\mathrm{centroid}(T_{\min}) - C_j\right\|\right)$
12. $T_{\min} \leftarrow T_{\min} \cup \lambda_{j^*}$
13. **end do**

Figure 7. Modified scheduling algorithm combining load balancing and memory locality-aware algorithms.

Figure 8 shows a 2D example of memory locality-aware and load-balanced data-privatization scheduling. Figure 8(a) shows non-uniform particle distribution. Here, we assume that the workload in each cell is proportional to the number of particles in the cell. Figure 8(b) illustrates the result of scheduling. Most computation units on the lower left corner are assigned to T_1, while the rest are assigned to T_2. The load-imbalance factor γ in this example is 0.0435.

The memory requirement of this algorithm can be analyzed as follows. The memory for each thread comes from two sources: 1) memory for the actually assigned computation units M_λ; and 2) memory for the surface cells neighboring the assigned computation units M_s. The amount of memory requirement for the first source is

$$M_\lambda = O\left(\frac{N}{P}\right), \tag{15}$$

whereas that for the second source is

$$M_s = O\left(\left(\frac{N}{P}\right)^{2/3}\right). \tag{16}$$

Hence, the memory requirement for one thread is

$$M_\lambda + M_s = O\left(\frac{N}{P} + \left(\frac{N}{P}\right)^{2/3}\right), \tag{17}$$

and the memory footprint of P threads on a node is

$$\begin{aligned} P(M_\lambda + M_s) &= O\left(P\frac{N}{P} + P\left(\frac{N}{P}\right)^{2/3}\right) \\ &= O\left(N + P^{1/3}N^{2/3}\right) \end{aligned}. \tag{18}$$

Thus, the asymptotic memory requirement for each node is $O(N+P^{1/3}N^{2/3})$, which is much smaller than the $O(NP)$ memory requirement of the traditional data-privatization algorithm.

Although this algorithm reduces the memory footprint significantly, it poses a difficulty in utilizing the hypercube reduction. This difficulty arises from the fact that the partial private force arrays are not aligned with each other. Nevertheless, the cost of linear reduction is reduced to $O(N+P^{1/3}N^{2/3})$ as a consequence of the reduced memory footprint. In fact, for a given P, the computation time of the partial linear reduction could be less than that of the hypercube reduction when N is large such that $O(P^{1/3}N^{2/3}) < O(N\log P)$.

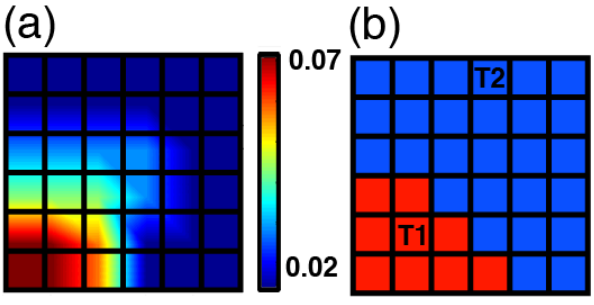

Figure 8. 2D illustration of memory locality-aware algorithm. (a) Spatial particle distribution where the normalized particle density is color-coded. (b) The corresponding computation-unit assignment to two threads originating at T_1 and T_2 cells.

IV. Performance Evaluation

In this section, we perform performance measurements and analysis for the algorithm described in the previous section. Section IV-A measures the load-imbalance factor of our memory locality-aware scheduling. Section IV-B measures the memory requirement reduction achieved by our approach and confirms the $O(N+P^{1/3}N^{2/3})$ memory requirement for each node. Section IV-C demonstrates that the scheduling cost can be reduced without affecting the quality of load balancing. Section IV-D compares the performance of the hybrid MPI/OpenMP scheme with that of the MPI-only scheme.

A. Thread-Level Load Balancing

We have performed a load-balancing test for the scheduling algorithm on a dual six-core AMD Opteron 2.3 GHz with $N = 8{,}192$ (Fig. 9). The actual measurement of the load-imbalance factor γ is plotted along with its estimator introduced in section III-A and the theoretical bound given by Eq. (13) as a function of P. The results show that $\gamma_{\text{estimated}}$ and γ_{actual} are close, and are below the theoretical bound. Also, $\gamma_{\text{theoretical}}$, $\gamma_{\text{estimated}}$ and γ_{actual} are increasing functions of P. This result indicates the severity of the load imbalance for a highly multi-threaded environment and highlights the importance of the fine-grain load balancing.

We have also observed that the performance fluctuates slightly depending on the initial cell selection of the memory locality-aware algorithm. While the random initial cell selection tends to provide robust performance compared to deterministic selection, it is possible to use some optimization techniques (*e.g.*, reinforcement learning) to dynamically optimize the initial cell selection at runtime. For more irregular applications, it is conceivable to combine the light-overhead thread-level load balancing in this paper with a high quality node-level load balancer such as a hypergraph-based approach [20].

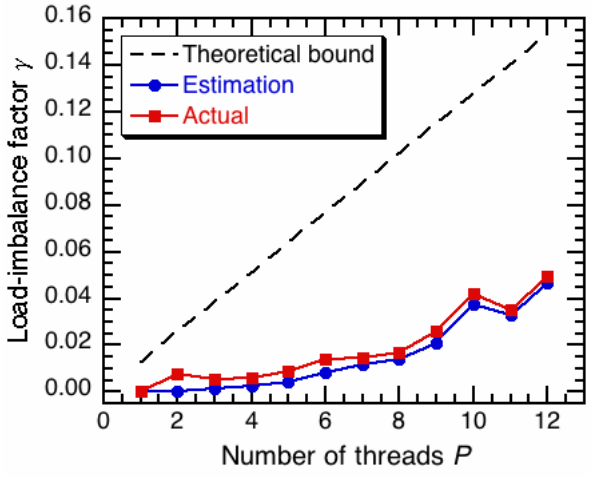

Figure 9. Load-imbalance factor γ as a function of P from theoretical bound, scheduler estimation, and actual measurement.

B. Memory Footprint

To test the memory efficiency of the proposed method, we perform a simulations on a four quad-core AMD Opteron 2.3 GHz machine with the fixed number of particles N = 8,192, 16,000, and 31,250. We measure the memory allocation size for 100 MD steps while varying the number of threads P from 1 to 16. Figure 10 shows the average memory allocation size of the force array as a function of the number of threads for the proposed algorithm compared to that of a traditional data-privatization algorithm. The results show that the memory requirement is reduced by 65%, 72%, and 75% for 16 threads for N = 8,192, 16,000, and 31,250, respectively, compared with the traditional $O(NP)$ memory requirement. In Fig. 10, the dashed curves show the reduction of memory requirement per thread,

$$m = aP^{-1} + bP^{-2/3} \tag{19}$$

where the first term represents the memory scaling from actual assigned cells and the second term represents scaling from surface cells of each thread (Eq. 17). The regression curves fit the measurements well, indicating that the memory requirement is accurately modeled by $O(N+P^{1/3}N^{2/3})$ per node or $O(N/P+(N/P)^{2/3})$ per thread.

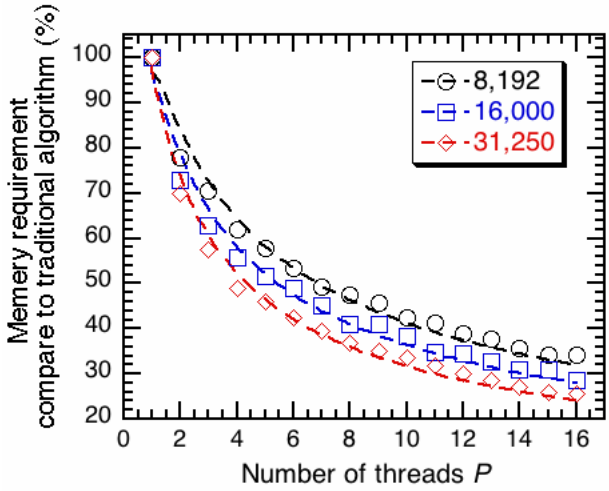

Figure 10. Average memory consumption for the private force arrays using our memory-saving strategy compared to the conventional entire-allocation method. Numbers in the legend denote the number of particles N.

We also measure the computation time spent for the reduction of the private force arrays to obtain the global force array. Figure 11 shows the reduction-operation time as a function of the number of threads P for N = 8,192, 16,000, and 31,250 particles. Here, dashed curves represent the regression,

$$t = aP^{1/3} + b \tag{20}$$

The curves fit well in all cases. This validates our analysis of the reduction-operation cost that it follows $O(P^{1/3})$ scaling.

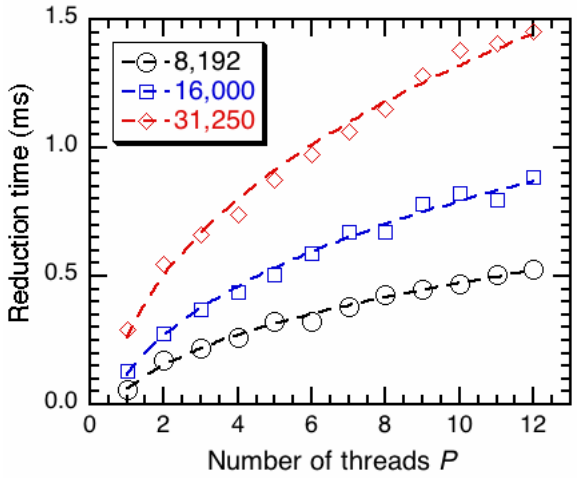

Figure 11. Average reduction-operation time of the memory locality-aware scheduling as a function of the number of threads. Numbers in the legend denote N.

C. Scheduling Cost

Though our scheduling algorithm has successfully reduced the memory footprint compared to the traditional data-privatization threading, one might be concerned with the cost of the schedule calculation at each MD step. To quantify the computational overhead of the scheduling algorithm, we have measured and analyzed the scheduling costs. Figure 12 shows the measured scheduling costs when the scheduling is performed every 1 and 15 steps for N=8,192 and 16,000 varying the number of threads P from 1 to 12 on dual six-core AMD Opteron 2.3 GHz. The figure shows that the scheduling cost increases linearly from 0.8% for P = 1 to approximately 5% for P =12 when the scheduling is performed at every MD step. The longer scheduling interval of 15 MD steps shows a similar behavior but with approximately 15-fold smaller scheduling cost (less than 0.1 to 0.4%).

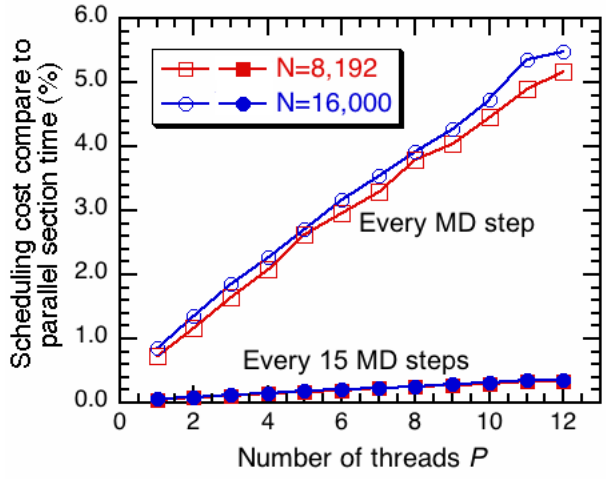

Figure 12. Computational overhead of our scheduling algorithm for N = 8,192 and 16,000 when the scheduling is performed every 1 and 15 steps.

We have found that skipping the scheduling for 15 MD steps does not degrade the quality of load balancing. In Fig. 13, the load-imbalance factor γ is plotted as a function of P when the scheduling is performed every 1 and 15 steps for 8,192-particle system. The result indicates that there is no significant variation between two different scheduling frequencies. This result can be explained by realizing that the particle trajectories change slowly over the course of simulation. Hence, the workload within each computation unit is stable for some period of time. It is therefore not necessary to execute the scheduling calculation at every MD step. This makes the cost of scheduling negligible.

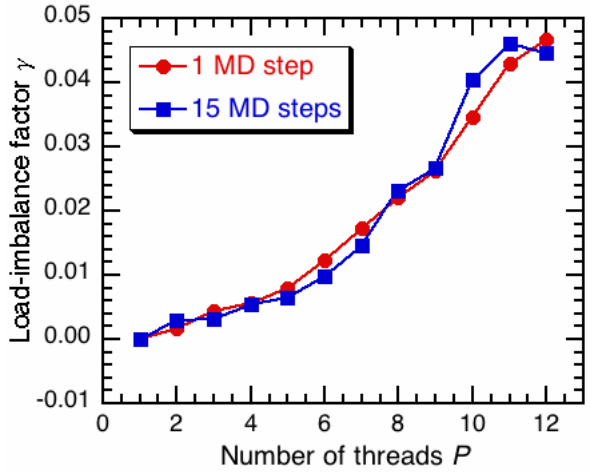

Figure 13. Load-imbalance factor as a function of P for the scheduling period of 1 and 15 MD steps.

D. Strong-Scaling Performance

We have measured the performance of the combined memory locality-aware and load-balancing algorithms. Figure 14 shows the thread-level strong-scaling speedup up to 16 threads on a four quad-core AMD Opteron 2.3 GHz for N = 8,192. The algorithm achieves a speedup of 14.43 on 16 threads, *i.e.*, the strong-scaling multi-threading parallel efficiency is 0.90. As shown in section IV-B, the combined algorithm reduces the memory consumption on the force array up to 65% for N = 8,192, while still maintaining an excellent strong scalability.

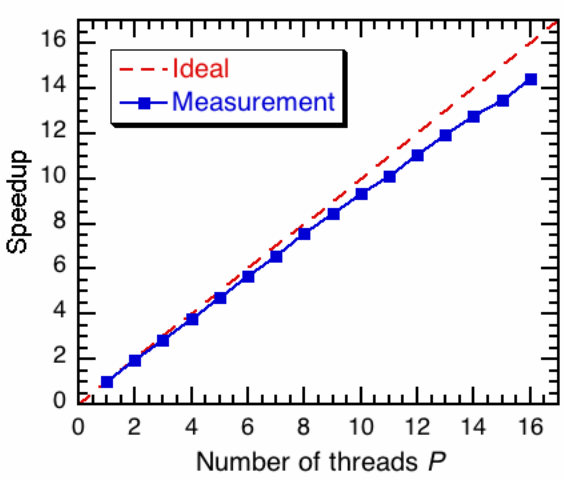

Figure 14. Thread-level strong scalability of the parallel section on a four quad-core AMD Opteron 2.3 GHz. The problem size is fixed as N = 8,192 particles.

Next, we compare the strong-scaling performance of the hybrid MPI/OpenMP and MPI-only schemes for large-scale problems on BlueGene/P at the Lawrence Livermore National Laboratory. One BlueGene/P node consists of four PowerPC 450 850 MHz processors. The MPI-only implementation treats each core as a separate task, while the hybrid MPI/OpenMP implementation has one MPI task per node, which spawns four worker threads for the force computation. The test is performed on 512 BlueGene/P nodes, which is equivalent to 2,048 MPI tasks in the MPI-only case and 512×4 threads for hybrid MPI/OpenMP. The problem size is fixed as 221,184 particles.

The test result in Fig. 15 shows that the speed of the MPI-only implementation outperforms that of hybrid MPI/OpenMP on 128 cores by a factor of 1.39. This result is expected, since the performance of the MPI/OpenMP code is limited by Amdahl's law. Namely, only the pair kernel is parallelized, while the rest of the program is sequential in the thread level. This disadvantage of the MPI/OpenMP code diminishes as the number of cores increases from 256, 512, to 1,024. Eventually, the hybrid MPI/OpenMP code performs better than the MPI-only code on 2,048 cores. The main factors underlying this result are: 1) the surface-to-volume ratio of the MPI-only code is larger than that of the hybrid MPI/OpenMP code; and 2) the communication latency for each node of the MPI-only code is four times larger than that of the hybrid MPI/OpenMP code.

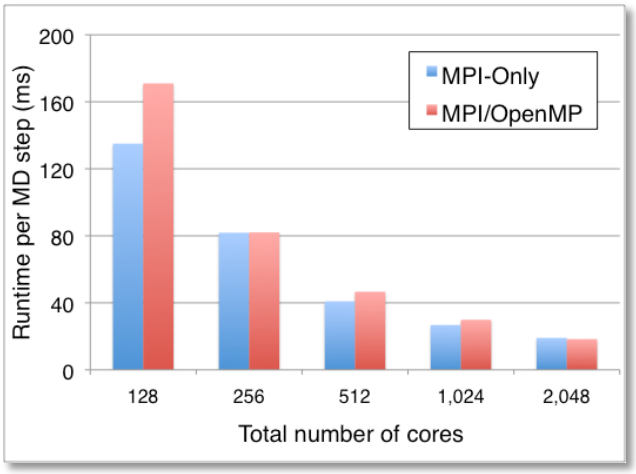

Figure 15. Total running time per MD step for a fixed problem size at N = 221,184 particles. The benchmark is performed on 32 to 512 BlueGene/P nodes (4 cores/node).

To verify this hypothesis, we have further performed a preliminary study on a massive strong-scaling comparison of hybrid MPI/OpenMP and MPI-only schemes. We increase the number of BlueGene/P nodes to 8,192 nodes (32,768 cores). Figure 16 shows the running time of 0.84-million particle system for the total number of cores ranging from 1,024 to 32,768. The result indicates that the hybrid scheme performs better when the core count is larger than 8,192. On the other hand, the MPI-only scheme gradually stops gaining benefit from the increased number of cores and become slower when using 32,768 cores. Note that the crossover point of the two schemes in terms of the granularity in Figs. 15 and 16 are similar at ~ 100 particles/core. This result confirms the assertion that the MPI/OpenMP model (or similar hybrid schemes) will be required to achieve better strong-scaling performance on large-scale multicore architectures.

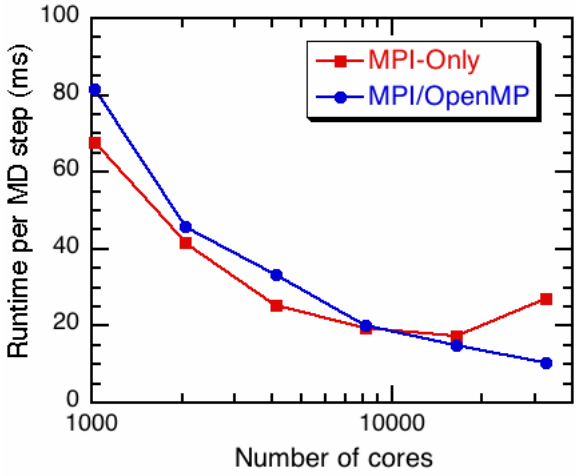

Figure 16. Total running time per MD steps for a fixed problem size at N = 843,750 particles on 1,024 – 32,768 Power PC 450 850 MHz cores.

V. Conclusions

We have demonstrated that our memory locality-aware scheduling algorithm successfully overcomes the disadvantages of the traditional data-privatization threading with minimal overhead. The scheduling algorithm guarantees a bounded load-imbalance factor while reducing the memory requirement from $O(NP)$ to $O(N+P^{1/3}N^{2/3})$. The cost of scheduling can be eliminated without the loss of load-balancing quality by reducing the scheduling frequency. Also, benchmarks of the massively parallel MD simulations suggest significant performance benefits of the hybrid MPI/OpenMP scheme for fine-grained large-scale strong-scaling applications.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-457414). The work at USC was partially supported by DOE BES/EFRC/SciDAC/SciDAC-e and NSF PetaApps/EMT/CRI.

References

[1] J. C. Phillips, *et al.*, "NAMD: Biomolecular simulations on thousands of processors," in *Proceedings of Supercomputing*, Los Alamitos, CA, 2002.

[2] F. H. Streitz, *et al.*, "Simulating solidification in metals at high pressure: The drive to petascale computing," *SciDAC 2006: Scientific Discovery Through Advanced Computing,* vol. 46, pp. 254-267, 2006.

[3] K. J. Bowers, *et al.*, "Zonal methods for the parallel execution of range-limited N-body simulations," *Journal of Computational Physics,* vol. 221, pp. 303-329, Jan 20 2007.

[4] B. Hess, *et al.*, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation,* vol. 4, pp. 435-447, 2008.

[5] K. Nomura, *et al.*, "A metascalable computing framework for large spatiotemporal-scale atomistic simulations," in *Proceedings of the International Parallel and Distributed Processing Symposium,* IEEE, 2009.

[6] D. E. Shaw, *et al.*, "Millisecond-scale molecular dynamics simulations on Anton," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, Oregon, 2009.

[7] J. N. Glosli, *et al.*, "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Proceedings of Supercomputing*, Reno, Nevada, 2007, pp. 1-11.

[8] S. R. Alam, *et al.*, "Impact of multicores on large-scale molecular dynamics simulations," in *Proceedings of the International Parallel and Distributed Processing Symposium*, Miami, Florida USA, 2008.

[9] L. Peng, *et al.*, "A scalable hierarchical parallelization framework for molecular dynamics simulation on multicore clusters " in *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, 2009.

[10] M. J. Chorley, *et al.*, "Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters," *International Journal of High Performance Computing Applications,* vol. 23, pp. 196-211, Aug 2009.

[11] C. Long, *et al.*, "Dynamic load balancing on single- and multi-GPU systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2010, pp. 1-12.

[12] R. Hockney and J. Eastwood, *Computer simulation using particles*. New York: McGraw-Hill, 1981.

[13] T. Darden, *et al.*, "Particle mesh Ewald: An N log(N) method for Ewald sums in large systems," *Journal of Chemical Physics,* vol. 98, pp. 10089-10092, 1993.

[14] D. York and W. Yang, "The fast Fourier Poisson method for calculating Ewald sums," *Journal of Chemical Physics,* vol. 101, pp. 3298-3300, 1994.

[15] J-L. Fattbert, *et al.*, In preparation.

[16] A. Sunarso, *et al.*, "GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows," *Journal of Computational Physics,* vol. 229, pp. 5486-5497, 2010.

[17] J. Yang, *et al.*, "GPU accelerated molecular dynamics simulation of thermal conductivities," *Journal of Computational Physics,* vol. 221, pp. 799-804, 2007.

[18] C. Hu, *et al.*, "Efficient parallel implementation of molecular dynamics with embedded atom method on multi-core platforms," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, 2009.

[19] D. W. Holmes, *et al.*, "An events based algorithm for distributing concurrent tasks on multi-core architectures," *Computer Physics Communications,* vol. 181, pp. 341-354, 2010.

[20] U. V. Catalyurek, *et al.*, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2007, pp. 1-11.