Scalable Data-Privatization Threading 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]Departments of Computer Science, Physics, 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 load balancing spanning forest (LBSF) partitioning algorithm, which combines: 1) fine-grain dynamic load balancing; and 2) minimal memory-footprint data privatization via nucleation-growth allocation. 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 p = 16 threads working on n = 8,192 particles, while maintaining the average thread-level load-imbalance less than 5%. Strong-scaling speedup for the kernel is 14.4 with 16-way threading on a four quad-core AMD Opteron node. In addition, our MPI/OpenMP code shows 2.58× and 2.16× speedups over the MPI-only implementation, respectively, for 0.84 and 1.68 million particles systems on 32,768 cores of BlueGene/P.**

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 [6, 9-11].

Hybrid parallelization based on MPI/threading schemes will likely replace MPI-only parallel MD. However, efficiently integrating a multi-threading framework into an existing MPI-only code is difficult for several reasons: 1) highly overlapped memory layout in typical MD codes incurs serious race condition; 2) naïve threading algorithms usually create significant overhead, and limit 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 [12].

To address these issues, we have designed a load balancing spanning forest (LBSF) partitioning algorithm, which combines: 1) fine-grain dynamic load balancing; and 2) minimal memory-footprint data privatization via nucleation-growth allocation. 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, followed by the description and analysis of the proposed data-privatization algorithm in section III. Section IV evaluates the performance of the hybrid parallelization algorithm, and 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 MD simulations is the evaluation of

PDPTA'11, July 18-21, 2011, Las Vegas, Nevada, USA.

the potential energy function and associated forces. One model of great physical importance is the interaction between a collection of point charges, which is described by the long-range, pair-wise Coulomb field $1/r$ (r is the interparticle distance), requiring $O(N^2)$ operations to evaluate. Many methods exist to reduce this computational complexity [13-15]. We focus on the highly efficient particle-particle/particle-mesh (PPPM) method [13]. In PPPM the Coulomb potential is decomposed into two parts: A short-range part that converges quickly in real space and a long-range part that converges quickly in reciprocal space. The split of work between the short-range and long-range part is controlled though a "screening parameter" α. With the appropriate choice of α, computational complexity for these methods can be reduced to $O(N\log N)$.

Because the long-range part of the Coulomb potential can be threaded easily (as a parallel loop over many individual 1D fast Fourier transforms), this paper explores efficient parallelization of the more challenging short-range part of the Coulomb potential using OpenMP threading. The short-range part is a sum over pairs: $\phi = \sum_{i<j} q_i q_j \mathrm{erfc}(\alpha r_{ij})/r_{ij}$, 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 particular 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. An extensive comparison of MPI-only ddcMD with other pure MPI codes can be found in [16].

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 particle data between domains. The surface-to-volume ratio of the domains and the choice of potential set 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 controls 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 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 of atomic crystals.

In addition to setting the communication cost, the domain decomposition also controls 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 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 [17].

B. Intranode Operations

Once particles are assigned to domains and remote particles are communicated, the force calculation begins. Figure 1(a) 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.

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. At present we choose an ordering specifically tailored to take advantage of the BlueGene "double-Hummer" (SIMD) operations. However, we could just as easily reorder the data to account for NUMA or GPGPU architectural details. We consistently find that the benefit of the regular memory access far outweighs the cost of particle ordering. The threading techniques proposed here are specifically constructed to preserve these memory-ordering advantages.

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 $\{C_k \mid 0 \le k < L_x L_y L_z\}$ 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 $\lambda_k = \{(\mathbf{r}_i, \mathbf{r}_j) \mid \mathbf{r}_i \in C_k;\ \mathbf{r}_j \in \mathrm{nn}^+(C_k)\}$ for cell C_k is defined as a collection of pair-wise computations (see Fig. 1(b)), where $\mathrm{nn}^+(C_k)$ is a set of half the nearest-neighbor cells of C_k. Newton's third law 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. The set of all computation units on each node is denoted as $\mathbf{\Lambda} = \{\lambda_k \mid 0 \le k < L_x L_y L_z\}$. Since most of our analysis is performed at a node level, $n = N/P$ hereafter denotes the number of particles in

each node (P is the number of nodes), and p is the number of threads in each node.

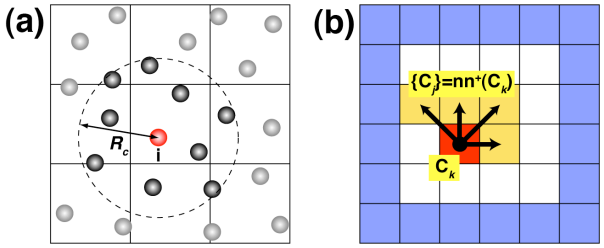

Figure 1. (a) 2D schematic of the linked-list cell method for pair computation with the cell dimension R_c. Only forces exerted by particles within the cutoff radius (represented by a two-headed arrow) are computed for particle i. (b) 2D schematic of a single computation unit λ_k. The shaded cells C_j pointed by the arrows constitute the half neighbor cells, $\mathrm{nn}^+(C_k)$.

We 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 [18, 19].
- Spatial decomposition coloring [20]—scalable without increasing computation, but can cause load imbalance.
- Mutually exclusive dynamic scheduling [21, 22]—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 per node and associated reduction sum cost [22].

This paper focuses on hybrid MPI/OpenMP parallelization on Sequoia, the third generation of BlueGene, which will be online in 2011-2012 at Lawrence Livermore National Laboratory (LLNL). On this SMP platform called Sequoia, MPI-only programming will not be an option for full-scale runs with 3.2 million concurrent threads.

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. 2(a)). 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.

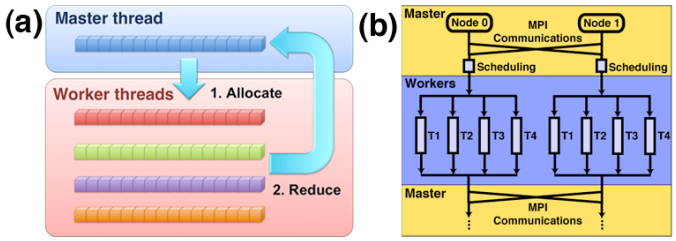

Figure 2. (a) Schematic of a memory layout for a traditional data privatization. (b) Schematic workflow of our hybrid MPI/OpenMP scheme.

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 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 2(b) shows the workflow of the hybrid MPI/OpenMP code. 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 section III-A. In section III-B, the load-balancing scheduler is further enhanced by introducing a nucleation-growth allocation algorithm.

A. *Thread-Level Load-Balancing 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 $T_i \subseteq \mathbf{\Lambda}$ denote a mutually exclusive subset of computation units assigned to the i-th thread, $\cap_{k=0,\ldots,LxLyLz}\lambda_k = \varnothing$. The computation time spent on λ_k is denoted as $\tau(\lambda_k)$. Thus, the computation time of each thread $\tau(T_i)$ is a sum of all computation units assigned to thread T_i. The algorithm initializes T_i to be empty, and loops over λ_k in Λ. Each iteration selects the least-loaded thread $T_{\min} = \mathrm{argmin}(\tau(T_i))$, and assigns λ_k to it. This approach is a 2-approximation algorithm [23] and its pseudo code is shown in Fig. 3.

Algorithm Fine-Grain Load Balancing

1. **for** $0 \le i < p$ **do**
2. $T_i \leftarrow \varnothing$
3. **end do**
4. **for each** λ_k **in** Λ **do**
5. $T_{\min} \leftarrow \text{argmin}_{0 \le i < p}(\tau(T_i))$
6. $T_{\min} \leftarrow T_{\min} \cup \lambda_k$
7. **end do**

Figure 3. Thread load-balancing algorithm.

The 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. 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,

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

By definition, $\gamma = 0$ when the loads are perfectly balanced. Since $\min(\tau(T_i)) \le \tau_{\text{average}}$,

$$\gamma \le \frac{\max(\tau(T_i)) - \min(\tau(T_i))}{\tau_{\text{average}}}. \quad (2)$$

In our load-balancing algorithm, the workload of $T_{\min}$ is increased at most by $\max(\tau(\lambda_k))$ at each iteration. This procedure guarantees that the variance of the workloads among all threads is limited by

$$\max(\tau(T_i)) - \min(\tau(T_i)) \le \max(\tau(\lambda_k)). \quad (3)$$

Substituting Eq. (3) in Eq. (2) provides an upper limit for the load-imbalance factor,

$$\gamma \le \frac{\max(\tau(\lambda_k))}{\tau_{\text{average}}}. \quad (4)$$

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. Fortunately, since particle positions change slowly $\tau(\lambda_k)$ remains highly correlated between the consecutive MD steps. Therefore, we use $\tau(\lambda_k)$ measured in the previous MD step as an estimator of $\tau(\lambda_k)$. This automatically takes into account any local variations in the cost of the potential evaluation. 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.

B. Load Balancing Spanning Forest Partitioning 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 necessary to allocate a complete copy of the force array for 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 three-step algorithm (Fig. 4): 1) the scheduler assigns computation units to threads and then determines which subset of the global data each thread requires; 2) each thread allocates its private memory as determined by the scheduler; 3) private force arrays from all threads are reduced into the global force array.

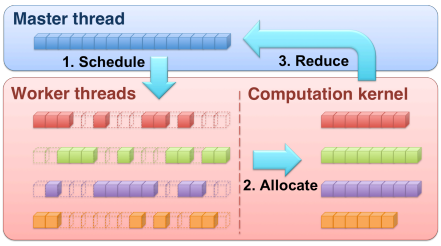

Figure 4. Memory layout and three-step algorithm for scheduling algorithm.

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. Since ddcMD sorts the particle data based on the cell they reside in, 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 $\text{nn}^+(C_k)$ as shown in Fig. 1(b), the force data of particles in $\text{nn}^+(C_k)$ 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. This is achieved by minimizing the surface-to-volume ratio of T_i. For this purpose, we implement the LBSF algorithm (Fig. 5) by modifying algorithm shown in section III-A. First, we randomly assign a root 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 a computation unit λ_{j^*} 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 root and continue to grow from that point.

Figure 6 shows a 2D example of the LBSF partitioning. Figure 6(a) shows non-uniform particle distribution, where the workload in each cell is assumed to be proportional to the number of particles in the cell. Figure 6(b) illustrates the result of a partitioning by scheduler. 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 0.044.

Algorithm LBSF Partitioning

1. $i \leftarrow 0$
2. **while** $i < p$ **do**
3. **repeat**
4. $\lambda_{root} \leftarrow$ random($\mathbf{\Lambda}$)
5. **until** $\lambda_{root} \notin T_{j(<i)}$
6. $T_i \leftarrow \lambda_{root}$
7. $i \leftarrow i + 1$
8. **end do**
9. **while** $\bigcup_{0<i<p} T_i \neq \Lambda$ **do**
10. $T_{min} \leftarrow \text{argmin}_{0<i<p}(\tau(T_i))$
11. $j^* \leftarrow \text{argmin}_{Cj \in \text{nn}(T\text{min})}(||\text{centroid}(T_{min})\text{-}C_j||)$
12. $T_{min} \leftarrow T_{min} \cup \lambda_{j^*}$
13. **end do**

Figure 5. LBSF partitioning algorithm.

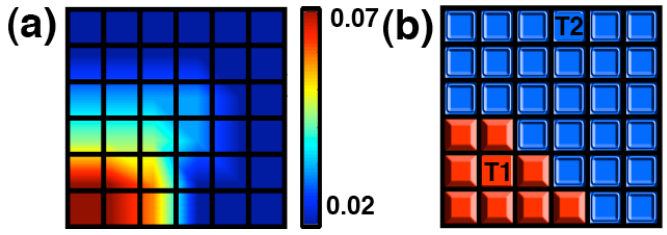

Figure 6. 2D illustration of the LBSF partitioning algorithm. (a) Spatial particle distribution where the normalized particle density is color-coded. (b) The corresponding computation-unit assignment to two threads originating at T_1 and T_2 cells.

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 one thread is

$$\begin{aligned} M_\lambda &= O(n/p) \\ M_s &= O(n/p)^{2/3}, \end{aligned} \tag{5}$$

and the memory footprint of p threads on a single node is

$$\begin{aligned} M_{\text{node}} &= p\left(M_\lambda + M_s\right) \\ &= O(p(n/p) + p(n/p)^{2/3}). \\ &= O(n + p^{1/3}n^{2/3}) \end{aligned} \tag{6}$$

The asymptotic memory requirement for each node is thus $O(n+p^{1/3}n^{2/3})$, which is much smaller than the $O(np)$ memory requirement of traditional data-privatization.

Although this algorithm reduces the memory footprint significantly, it poses a minor difficulty in the reduction sum. This difficulty arises from the fact that the partial private force arrays are not aligned with each other. Nevertheless, the cost of linear reduction sum 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 sum could be less than that of a hypercube reduction when n is large such that $O(p^{1/3}n^{2/3}) < O(n\log p)$.

IV. Performance Evaluation

In this section, we perform performance measurements and analysis of the algorithm described in the previous section. Section IV-A measures the load-imbalance factor of our scheduler, and section IV-B measures the memory-footprint reduction by our approach, confirming its $O(n+p^{1/3}n^{2/3})$ scaling. Section IV-C demonstrates the reduction of the scheduling cost without affecting the quality of load balancing, followed by strong-scaling comparison of the hybrid MPI/OpenMP and MPI-only schemes.

A. Thread-Level Load Balancing

We perform a load-balancing test for the scheduling algorithm on a dual six-core AMD Opteron 2.3 GHz with n = 8,192 (Fig. 7(a)). The actual measurement of the load-imbalance factor γ is plotted as a function of p, along with its estimator introduced in section III-A and the theoretical bound, Eq. (4). The results show that $\gamma_{\text{estimated}}$ and γ_{actual} are close, and are below the theoretical bound. γ is an increasing function of p, which indicates the severity of the load imbalance for a highly multi-threaded environment and highlights the importance of the fine-grain load balancing.

We also observe that the performance fluctuates slightly depending on a selection of root nodes in LBSF algorithm. While the random root 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 [24].

B. Memory Footprint

To test the memory efficiency of the proposed method, we perform simulations on a four quad-core AMD Opteron 2.3 GHz machine with a 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 7(b) 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 for 16 threads is reduced by 65%, 72%, and 75%, respectively, for n = 8,192, 16,000, and 31,250 compared with the traditional $O(np)$ memory per-node requirement. In Fig. 7(b), the dashed curves show the reduction of memory requirement per thread estimated as

$$m = ap^{-1} + bp^{-2/3}, \tag{7}$$

where the first term represents the memory scaling from actual assigned cells and the second term represents scaling from

surface cells of each thread, see Eq. (5). The regression curves fit the measurements well, indicating that the memory requirement is accurately modeled by $O(n+p^{1/3}n^{2/3})$.

We also measure the computation time spent for the reduction sum of the private force arrays to obtain the global force array. Figure 7(c) shows the reduction-sum 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,

$$t_{\text{reduction}} = ap^{1/3} + b, \tag{8}$$

which fit all cases well.

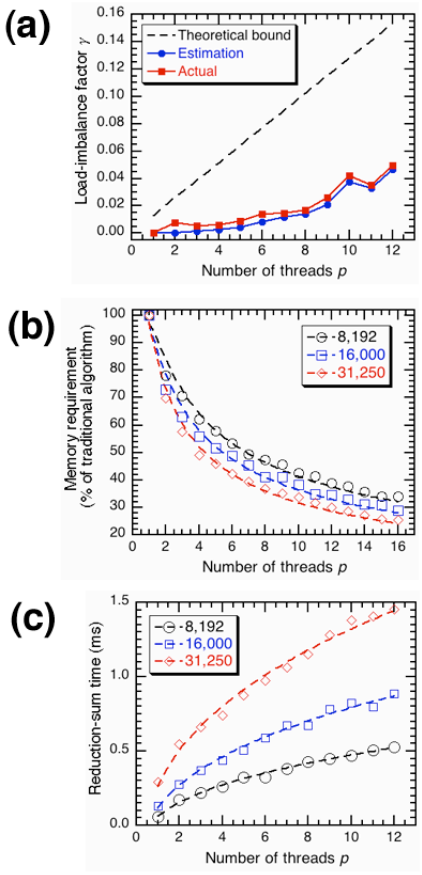

Figure 7. (a) Load-imbalance factor γ as a function of p from theoretical bound, scheduler estimation, and actual measurement. (b) Average memory consumption for the private force arrays as a function of p using LBSF compared to the conventional method. Numbers in the legend denote n. (c) Average reduction-sum time of the LBSF scheduling as a function of p.

C. *Strong-Scaling Performance*

We measure the performance of the combined nucleation-growth allocation and load-balancing algorithms. Figure 8(a) shows the thread-level strong-scaling speedup on a four quad-core AMD Opteron 2.3 GHz. 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 up to 65% for n = 8,192, while still maintaining excellent strong scalability.

Next, we compare the strong-scaling performance of the hybrid MPI/OpenMP and MPI-only schemes for large-scale problems on BlueGene/P at LLNL. 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 P = 8,192 nodes, which is equivalent to 32,768 MPI tasks in the MPI-only case and 32,768 threads for hybrid MPI/OpenMP. Figures 8 (b) and (c) show the running time of 843,750 and 1,687,500 particles systems 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 becomes slower. The MPI/OpenMP code shows 2.58× and 2.16× speedup over the MPI-only implementation for N = 0.84 and 1.68 million, respectively, when using 32,768 cores. Note that the crossover granularity of the two schemes is $n/p \sim 100$ particles/core for both cases.

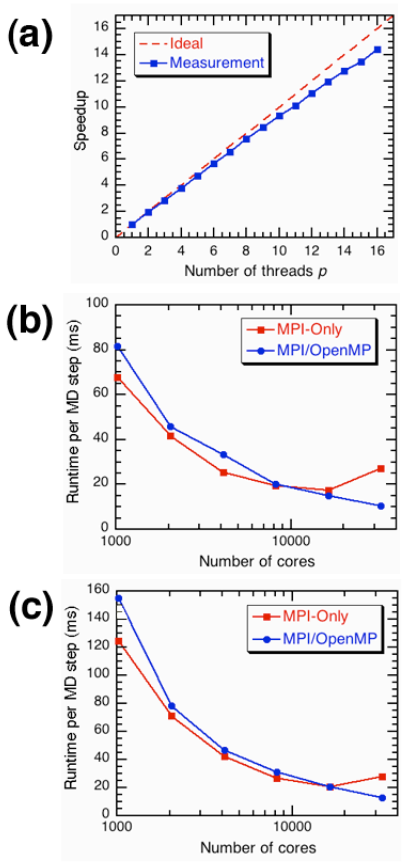

Figure 8. (a) Thread-level strong scalability of the parallel section on a four quad-core AMD Opteron 2.3 GHz with fixed problem size at n = 8,192 particles. (b) Total running time per MD steps of 1,024 – 32,768 Power PC 450 850 MHz cores on BG/P for a fixed problem size at N = 0.84-million particles and (c) 1.68-million particles.

This result indicates that the Amdahl's law limits the performance of the MPI/OpenMP code when using small number of nodes. 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. Eventually, the hybrid MPI/OpenMP code performs better than the MPI-only code after 8,192 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.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.

V. Conclusions

Our LBSF partitioning algorithm successfully overcomes the disadvantages of the traditional data-privatization threading with minimal overhead. The LBSF scheduling guarantees a bounded load imbalance 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-grain large-scale 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-469312). The work at USC was partially supported by DOE BES/EFRC/SciDAC/SciDAC-e/INCITE and NSF PetaApps/EMT.

References

[1] J. C. Phillips, *et al.*, "NAMD: Biomolecular simulations on thousands of processors," in *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, 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] D. E. Shaw, *et al.*, "Millisecond-scale molecular dynamics simulations on Anton," in *Supercomputing*, Portland, Oregon, 2009, pp. 1-11.

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

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

[8] S. R. Alam, *et al.*, "Impact of multicores on large-scale molecular dynamics simulations," in *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, 2009.

[11] R. Rabenseifner, *et al.*, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," *Proceedings of the Parallel, Distributed and Network-Based Processing,* pp. 427-436, 2009.

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

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

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

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

[16] D. F. Richards, *et al.*, "Beyond homogeneous decomposition: scaling long-range forces on Massively Parallel Systems," in *Supercomputing*, Portland, Oregon, 2009, pp. 1-12.

[17] J.-L. Fattbert, *et al.*, unpublished.

[18] 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.

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

[20] C. Hu, *et al.*, "Efficient parallel implementation of molecular dynamics with embedded atom method on multi-core platforms," in *International Conference on Parallel Processing Workshops*, 2009, pp. 121-129.

[21] 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.

[22] K. Madduri, *et al.*, "Memory-efficient optimization of Gyrokinetic particle-to-grid interpolation for multicore processors," in *Supercomputing*, Portland, Oregon, 2009, pp. 1-12.

[23] J. Kleinberg and E. Tardos, *Algorithm Design*, 2 ed.: Pearson Education, Inc., 2005.

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