LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-528373

Analysis of Scalable Data-Privatization Threading Algorithms for Hybrid MPI/OpenMP Parallelization of Molecular Dynamics

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

February 10, 2012

Journal of Supercomputing

Disclaimer

Analysis of Scalable Data-Privatization Threading Algorithms for Hybrid MPI/OpenMP Parallelization of Molecular Dynamics

Manaschai Kunaseth • David F. Richards • James N. Glosli • Rajiv K. Kalia • Aiichiro Nakano • Priya Vashishta

Abstract We propose and analyze threading algorithms for hybrid MPI/OpenMP parallelization of molecular-dynamics simulation, which are scalable on large multicore clusters. Two data-privatization thread scheduling algorithms via nucleation-growth allocation are introduced: (1) compact-volume allocation scheduling (CVAS); and (2) breadth-first allocation scheduling (BFAS). The algorithms combine fine-grain dynamic load balancing and minimal memory-footprint data privatization threading. We show that the computational costs of CVAS and BFAS are bounded by $\Theta(n^{5/3}p^{-2/3})$ and $\Theta(n)$, respectively, for p threads working on n particles on a multicore compute node. Memory consumption per node of both algorithms scales as $O(n+n^{2/3}p^{1/3})$, but CVAS has smaller pre-factors due to a geometric effect. Based on these analyses, we derive the selection criterion between the two algorithms in terms of the granularity, n/p. We observe that memory consumption is reduced by 75% for p = 16 and n = 8,192 compared to a naïve data privatization, while maintaining thread imbalance below 5%. We obtain a strong-scaling speedup of 14.4 with 16-way threading on a four quad-core AMD Opteron node. In addition, our MPI/OpenMP code achieves 2.58× and 2.16× speedups over the MPI-only implementation on 32,768 cores of BlueGene/P for 0.84 and 1.68 million particle systems, respectively.

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

M. Kunaseth • R. K. Kalia • A. Nakano • P. Vashishta
Collaboratory for Advanced Computing and Simulations, Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA
e-mail: {kunaseth, rkalia, anakano, priyav}@usc.edu

David F. Richards • James N. Glosli
Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
e-mail: {richards12,glosli}@llnl.gov

1. Introduction

Molecular dynamics (MD) simulation has become an important tool to study a broad range of scientific problems at the atomistic level [1-5]. Thanks to the rapid growth in computing power, large spatiotemporal scale MD simulations are widely available, enabling scientists to address more challenging problems [6-9]. However, recent improvements in computing power have been gained using multicore architectures instead of increased clock speed. This marks the end of the free-ride era, where legacy applications could obtain increased performance on a newer chip without substantial modification. Furthermore, the number of cores per chip is expected to grow continuously, which deepens the performance impact on legacy software.

As multicore chips become the standard of modern supercomputers, two significant issues have emerged: (1) performance of traditional parallel applications, which are solely based on the message passing interface (MPI), is expected to degrade substantially [10]; and (2) available memory per core tends to decrease because the number of cores per chip is growing considerably faster than the available memory [11]. 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 [5,12,13]. One of the most commonly used hierarchical parallelization frameworks is a hybrid message-passing/multithreading paradigm [14,15], which employs hybrid features of a distributed memory via message passing and a shared memory via multithreading.

The necessity of hierarchical parallelization has been further emphasized by the arrival of modern massive-scale multicore supercomputers such as “Sequoia”, the third generation of IBM BlueGene cluster with 1.6 million cores, which will be online at Lawrence Livermore National Laboratory (LLNL) in 2012. On such a gigantic symmetric multiprocessing (SMP) platform, MPI-only programming will not be an option for full-scale runs with up to 6.4 million concurrent threads. Especially for MD simulation on this class of clusters, hybrid parallelization based on MPI/threading schemes will likely replace the traditional MPI-only parallel MD.

However, efficiently integrating a multi-threading framework into an existing MPI-only code is challenging due to several reasons: (1) the highly overlapped memory layout in typical MD codes incurs a serious race condition in the computational kernel when multithreading is used; (2) naïve threading algorithms usually create significant memory and computation overheads, limiting the threading speedup for a large number of threads; and (3) the dynamic nature of MD requires low-overhead dynamic load balancing for threads to maintain good performance. Therefore, it is of significance to study and design an efficient threading algorithm particularly for a platform with such a large number of cores.

In this paper, we introduce two data-privatization thread scheduling algorithms based on a nucleation-growth concept to address these issues: (1) compact-volume allocation scheduling (CVAS); and (2) breadth-first allocation scheduling (BFAS). These two algorithms provide efficient threading schemes for MD by combining fine-grain dynamic load balancing and minimal memory-footprint threading. We also present an extensive theoretical analysis and comparisons of the proposed algorithms including: (1) the upper bound of load imbalance; (2) the computational complexity of the scheduling; and (3) memory consumption. These analyses provide insights into the algorithmic characteristics, leading to the identification of the relative advantage of CVAS and BFAS for different simulation parameters. Finally, we implement CVAS and BFAS algorithms in the MD code "ddcMD" [16] and demonstrate that the hybrid MPI/threading scheme outperforms an MPI-only scheme for strong scaling on large-scale MD problems.

This paper is organized as follows. Section 2 summarizes the hierarchy of parallel operations in ddcMD, followed by a detailed description of the proposed data-privatization algorithms in section 3. Theoretical analysis of the data-privatization algorithms is given in section 4. Section 5 evaluates the performance of the hybrid parallelization algorithm, and conclusions are drawn in section 6.

2. 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 $\phi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$, where $\mathbf{r}_i$ is the position of the i-th particle. 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 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 [17-19]. We focus on the highly efficient particle-particle/particle-mesh (PPPM) method [17]. 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 through a "screening parameter" α. With the appropriate choice of α, computational complexity for these methods is 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:

$$\Phi = \sum_{i<j} q_i q_j \frac{\mathrm{erfc}(\alpha r_{ij})}{r_{ij}},$$

where q_i is the charge of particle i, α denotes the screening parameter, r_{ij} is a separation between particles i and j, and $\mathrm{erfc}(x)$ denotes the complimentary error function,

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt\,.$$

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 [20].

2.1. Internode communication and load balancing in ddcMD

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. Improved surface-to-volume ratios in a homogeneous system are achieved if domain centers form a body-centered cubic, face-centered cubic, or hexagonal closed-packed lattice, which are common high-density arrangements 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 [21].

2.2. Intranode force computation

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 [22]. At present we choose an ordering specifically tailored to take advantage of the BlueGene "double-Hummer" single-instruction multiple-data (SIMD) operations [23]. However, we could just as easily reorder the data to account for non-uniform memory access (NUMA) or general-purpose graphics processing units (GPGPU) architectural details. We consistently find that the benefit of the regular memory access far outweighs the cost of particle reordering. 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 be the total number of cells in the system, and $\{C_k \mid 0 \le k < L\}$ 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, where $\mathrm{nn}^+(C_k)$ is a set of half the nearest-neighbor cells of C_k (see Fig. 1(b)). The symmetry of the forces from Newton's third law ($f_{ij} = -f_{ji}$) 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 $\mathbf{\Lambda} = \{\lambda_k \mid 0 \le k < L\}$. 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.

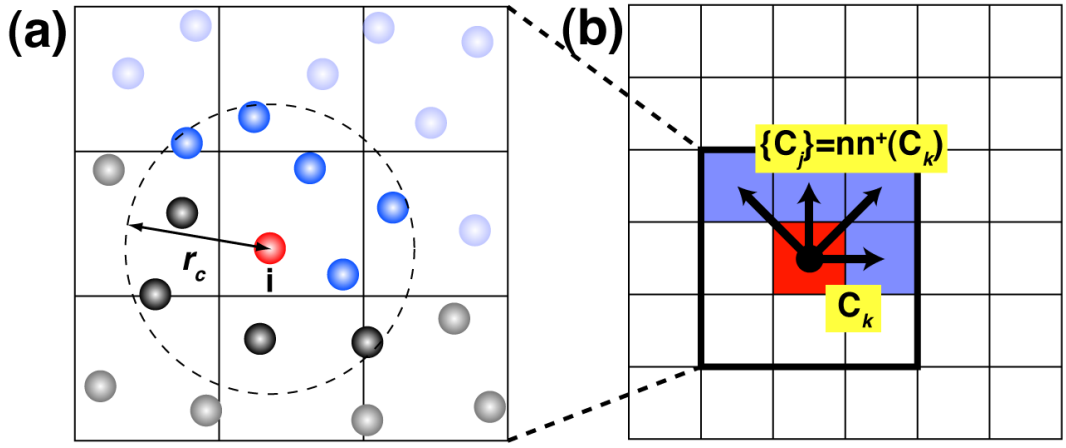

Fig. 1. (a) 2D schematic of the linked-list cell method for pair computation with the cell dimension r_c. Only forces exerted by particles within the cutoff radius (represented by a two-headed arrow) are computed for particle i. The blue particles represent particles in the top-half neighbor cells where the force is computed directly, while the forces on the gray particles are computed in other computation units. (b) 2D schematic of a single computation unit λ_k. The shaded cells C_j pointed by the arrows constitute the half neighbor cells, $nn^+(C_k)$.

2.3. Hybrid MPI/OpenMP parallelization

The hybrid MPI/OpenMP parallelization of ddcMD is implemented by introducing a thread scheduler into the MPI-only ddcMD. Figure 2 shows the workflow of the hybrid MPI/OpenMP code using a master-slave approach [24][25]. The program repeats the following three computational phases: (1) the master thread performs initialization and internode communications using MPI; (2) the scheduler computes the scheduled work for each thread; and (3) the worker threads execute the workloads in an OpenMP parallel section. The program utilizes an explicit scheduler based on our proposed algorithms to distribute the workload prior to entering the pair computation section. The parallel threading section is initiated by `#pragma omp parallel` construct. 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. This approach also reduces the number of synchronizations and minimizes context switching compared to a real-time dynamic scheduling approach.

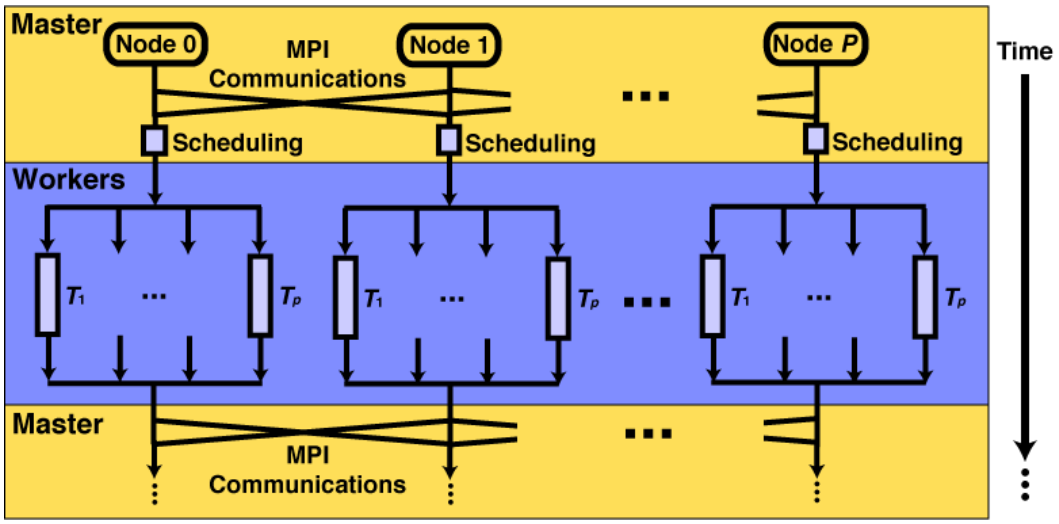

Fig. 2. Schematic workflow of a hybrid MPI/OpenMP scheme. The master thread schedules work for the worker threads before entering the parallel threading section. The master thread also handles internode communication, which is performed outside the threading region.

We parallelize the explicit pair-force computation kernel of ddcMD, which is the most computationally intensive kernel, at the thread level using OpenMP (see Fig. 3). Two major problems commonly associated with threading are: (1) a race condition among threads; and (2) thread-level load imbalance. The race condition occurs when multiple threads try to update the force of the same particle concurrently. Several techniques have been proposed previously to solve these problems:

- Duplicated pair-force computation—simple and scalable, but doubles computation (or more than double in many-body potentials). Due to its regular data-access pattern, it is usually used in GPGPU threading [26,27].
- Spatial decomposition coloring—scalable without increasing computation, but may cause considerable load imbalance [28].
- Dynamic scheduling—robust and suited for dynamic load balancing, but may incur fairly large overhead for context switching [29].
- Data privatization—no penalty on computation, but with excessive $\Theta(np)$ memory requirement per node and associated reduction sum cost [30].

We have designed CVAS and BFAS algorithms that combine a mutually exclusive scheduler with a reduced memory data-privatization scheme to address the race condition and fine-grain load balancing issues. The algorithms will be discussed in section 3.

Algorithm Pair-force computation kernel

```
1.  for 0 ≤ i < p
2.      T_i ← scheduleWork(i)
3.  end for
4.  omp parallel: nthreads = p
4.  i ← omp_get_thread_num()
5.  for each c_k:∀λ_k ∈ T_i
6.      for each c' ∈ nn+(c_k)
7.          ComputeForce(r_i,r_j):∀r_i ∈ c_k, ∀r_j ∈ c'
8.      end for
9.  end for
10. end omp parallel
```

Fig. 3. Pair-force computation kernel.

3. Data Privatization Scheduling Algorithms

Computation patterns in MD can lead to serious fine-grain race conditions. If the natural symmetry of the particle forces is exploited, multiple pair interactions involving a common particle (*e.g.,* interactions between particles *i*-*j* and *j*-*k*) possibly cause a write conflict in memory (*i.e.,* force array) when computed concurrently by two different threads. The simplest solution is not to exploit force symmetry, but this approach immediately doubles the computation (or more than double in case of many-body potentials). Hence, this approach is not a preferred solution in most circumstances.

To address these issues, we use a data privatization algorithm, which takes advantage of force symmetry while eliminating race conditions from the pair computations. However, a naïve implementation of data privatization consumes significantly more memory. This will be described in section 3.1. In section 3.2, the nucleation-growth allocation algorithm, an enhanced data privatization scheme with reduced memory consumption, will be discussed.

3.1. Naïve data privatization threading algorithm

A naïve data-privatization algorithm avoids write conflict in memory by replicating the entire write-shared data structure and allocating a private copy to each thread (Fig. 4). Clearly, the memory requirement for this redundant allocation scales as $\Theta(np)$. 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.

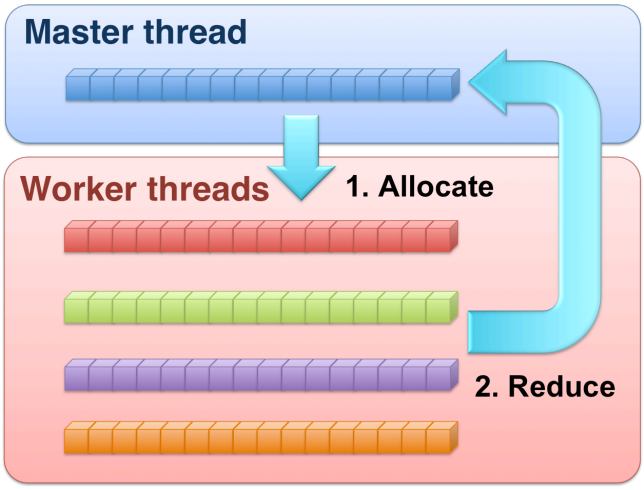

Fig. 4. Schematic of a memory layout for a naïve data privatization. To avoid data race conditions among threads, the entire output array is locally duplicated for each thread. These local arrays are privatized on the thread memory space and thus eliminate race conditions during the computation phase. This approach also incurs a reduction operation to sum the local results to the global array.

The naïve data privatization threading is simple and can be used to eliminate the race condition regardless of threading complexity in most situations. However, this naïve algorithm is usually not a suitable solution on a modern multicore platform because the algorithm allocates memory entirely without considering the actual work assigned to each thread, leading to much larger memory consumption when the number of threads is large. This drawback can be remedied if the data-access pattern is known ahead of the computation.

3.2. Nucleation-growth allocation algorithm

As explained in the previous section, the memory requirement of the naïve data-privatization algorithm is $\Theta(np)$. However, it is not necessary to allocate a complete copy of the force array for each thread, since only a subset of all computation units $\mathbf{\Lambda}$ is assigned to each thread. Therefore, we may 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. 5): (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; and (3) private force arrays from all threads are reduced into the global force array.

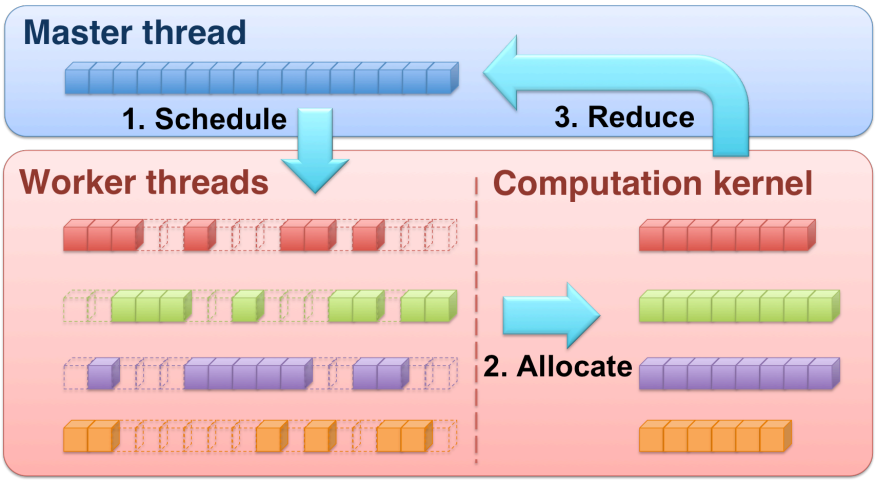

Fig. 5. Memory layout and three-step algorithm for the nucleation-growth allocation algorithm. Worker threads only allocate the essential portion of the force arrays, corresponding to their assigned workload.

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 $\mathrm{nn}^+(C_k)$ as shown Fig. 1(b), 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. This is achieved by minimizing the surface-to-volume ratio of particles assigned to each thread T_i. To achieve this, we implement two algorithms based on a nucleation-growth concept. This approach systematically divides the work as equally as possible with minimal memory overhead. These two algorithms are: (1) compact volume allocation scheduling (CVAS); and (2) breadth-first allocation scheduling (BFAS) algorithms. CVAS and BFAS employ a fine-grain load-balancing algorithm,

which will be explained in subsection 3.2.1. In subsections 3.2.2 and 3.2.3, we will discuss CVAS and BFAS algorithms in more details.

3.2.1. Thread-Level Load-Balancing Algorithm

We implement thread-level load balancing based on a greedy approach (*i.e.*, iteratively assigning 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. The computation time spent on λ_k is denoted as $\tau(\lambda_k)$. Thus, the computation time of each thread $\tau(T_i) = \Sigma_{\lambda \in Ti}\, \tau(\lambda)$ 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 $\mathbf{\Lambda}$. Each iteration selects the least-loaded thread $T_{\min} = \text{argmin}(\tau(T_i))$, and assigns λ_k to it. This original approach is a 2-approximation algorithm [31] and its pseudocode is shown in Fig. 6.

Algorithm Fine-Grain Load Balancing

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

Fig. 6. Thread load-balancing algorithm.

3.2.2. Compact-Volume Allocation Scheduling (CVAS) Algorithm

The CVAS algorithm consists of the following steps. First, we randomly assign a root computation unit λ_{root} to each thread. Next, the least-loaded thread $T_{\min}$ is identified. 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 then assign λ_{j*} to $T_{\min}$. The algorithm repeats until all computation units are assigned to threads. 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. The pseudocode of CVAS is shown in Fig. 7.

Algorithm Compact-volume allocation scheduling

1. Assign random root cell λ_{root} to each thread
2. **while** not all of the cells are assigned
3. Find the least loaded thread T_{min}
4. **if** nearest neighbor cells of T_{min} are already assigned to other threads
5. Randomly pick new root λ_{root*} for thread T_{min}
6. Assign λ_{root*} to T_{min}
7. Set λ_{root*} to be a new root of T_{min}
8. **else**
9. Find an unassigned cell λ_{j*} which has minimal distance to the centroid of T_{min}
10. Assign λ_{j*} to T_{min}
11. **end if**
12. **end while**

Fig. 7. CVAS algorithm.

3.2.3. Breadth-first allocation scheduling algorithm (BFAS)

One weakness of the CVAS algorithm is a fairly expensive scheduling cost. To improve this aspect, we propose a breadth-first allocation scheduling (BFAS) algorithm as an alternative. BFAS is designed to slightly increase memory consumption compared to CVAS but significantly reduce the scheduling cost. BFAS consists of the following steps. The initialization step randomly assigns a root computation unit λ_{root} to each thread. Then, an empty queue data structure Q_i is defined as a traversal queue for thread T_i. After that, each thread T_i adds all of the computation units surrounding λ_{root} into Q_i. The algorithm repeats the following steps until all of the computation units are assigned: (1) find the least-loaded thread T_{min}; (2) from the top of the queue Q_{Tmin}, find the computation unit λ_j that has not been assigned to any thread; (3) assign λ_j to T_{min} and update workload sum of each thread. Note that the BFAS algorithm proceeds similarly to a graph traversal using a breadth-first search (BFS) algorithm. The pseudocode of BFAS is shown in Fig. 8.

4. Theoretical Analysis

In this section, we present a comprehensive analysis of the two proposed algorithms, CVAS and BFAS. The analysis is important in order to understand the capabilities and limitations of the algorithms under different circumstances. We present an upper bound analysis for load imbalance profile (section 4.1), a lower

Algorithm Breadth-first allocation scheduling

```
1.  for each thread T_i
2.      Initialize queue Q_i = ∅ as a search queue
3.      Assign random root cell λ_root to T_i
4.      Enqueue all neighbor cells of λ_root to Q_i
5.  end for
6.  while not all of the cells are assigned
7.      Find the least loaded thread T_min
8.      if Q_Tmin is empty
9.          Randomly pick new root λ_root* for thread T_min
10.         Assign λ_root* to T_min
11.         Add all neighbor cells of λ_root* to Q_Tmin
12.     else
13.         do
14.             λ_j = dequeue(Q_Tmin)
15.         while λ_j is unassigned
16.         Assign λ_j to T_min
17.         Enqueue all neighbor cells of λ_j to Q_Tmin
18.     end if
19. end while
```

Fig. 8. BFAS algorithm.

bound analysis for memory consumption (section 4.2), and computational complexity analysis (section 4.3) of the proposed algorithms. Using the analysis from sections 4.2 and 4.3, we are able to identify the criterion to choose the most suitable algorithm for a particular system. This will be discussed in section 4.4.

4.1. Thread-level load imbalance analysis

The fine-grain load balancing algorithm presented in the previous section is simple yet provides an excellent load-balancing capability. In this section, we show that this approach has a well-defined upper bound for load imbalance. To quantify the load imbalance, we define a load-imbalance factor γ as the difference between the runtime of the slowest thread, $\max(\tau(T_i))$, and the average runtime, $\tau_{\text{average}} = (1/p)\Sigma_i \tau(T_i)$,

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

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

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

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

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

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

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

For the case where the density of particles is uniform, we can further assume that all computation units are equally expensive thus $\tau(\lambda) = \max(\tau(\lambda_k))$. In this case, the upper-bound load-imbalance factor is

$$\gamma_{\text{uniform}} \le \frac{\tau(\lambda)}{\tau_{\text{average}}} = \frac{p\tau(\lambda)}{L\tau(\lambda)} = \frac{p}{L}. \qquad (5)$$

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, $\tau(\lambda_k)$ remains highly correlated between the consecutive MD steps, since particle positions change slowly. 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.

4.2. Memory consumption

Limited available memory per core becomes an important constraint for threading algorithm design. Our nucleation-growth scheduling algorithms tend to distribute workload equally among threads while minimizing the surface area of

the assigned workload in the physical space. Here, let us assume that the number density of particles of the system, ρ, is uniform. The memory required for storing particle forces for each thread m_t is proportional to the volume V_t occupied by the particles required for the force computation of each thread:

$$m_t = \rho V_t. \tag{6}$$

The particles within volume V_t consist of two groups: (1) main particles, *i.e.*, the particles that are directly assigned to each thread; and (2) surface particles, *i.e.*, the particles that are not directly assigned to a particular thread but are required due to their interaction with the main particles of each thread. We define the *main volume* Ω as a volume occupied by the main particles and the *shell volume* ω as a volume occupied by the surface particles. Thus, V_t is written as

$$V_t = \Omega + \omega. \tag{7}$$

From Eqs. (6) and (7), we will be able to calculate the memory consumption if the main and shell volumes are known. Fortunately, both CVAS and BFAS algorithms tend to form a particular geometry, which can be estimated analytically.

CVAS and BFAS algorithms are different only during the workload assignment phase. CVAS chooses the closest unassigned cells to its centroid because it minimizes the change of its centroid after assignment. This results in a spherical shape of the main volume Ω created by CVAS (Fig. 9(b)). On the other hand, BFAS chooses unassigned cells layer-by-layer starting with the closest ones to its root cell. Since the cell has cubic geometry, BFAS volumes tend to form a cubic shape (Fig. 9(a)). These considerations on the CVAS and BFAS volume geometry do not take account the following facts: (1) the other threads might prevent the algorithms from obtaining the optimal cells; and (2) discretization might cause sub-optimal geometry for the algorithms (see Figs. 9(c) and (d)). Since these factors always increase the memory consumption from the optimal shape, an assumption of perfect CVAS or BFAS geometry implies the best-scenario (*i.e.,* lower-bound) memory consumption for both algorithms. We perform this analysis in detail in the following subsections.

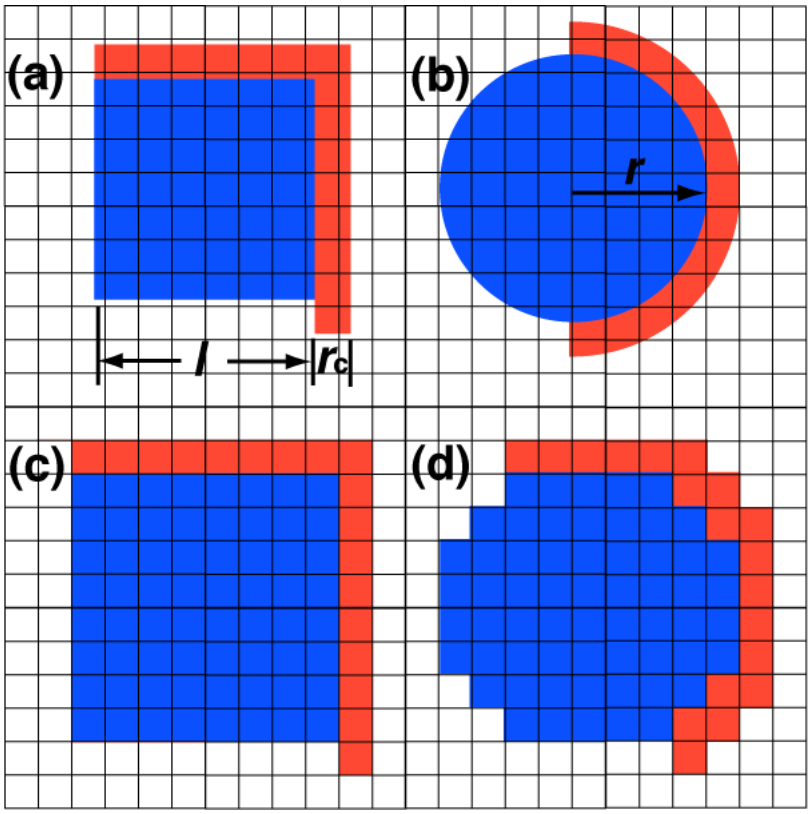

Fig. 9. Geometric effects on surface-to-volume ratio. (a) and (b) show the shell volume of cubic and spherical shapes with the same size of main volumes. (c) and (d) show the discretized version, which indicates that the spherical shape consumes less volume than the cubic shape. The square grid represents cells in which particles reside. The blue area refers to the main volume while the red area shows the shell volume. Variable l is defined as the side length of main volume created by BFAS while r denotes the radius of CVAS. The surface thickness is r_c.

4.2.1. Lower-bound memory consumption of CVAS algorithm

In the best case, Ω of CVAS forms a spherical shape. In this situation, the main volume obtained by each thread is

$$\Omega_{\mathrm{CVAS}} = \frac{4}{3}\pi r^3, \tag{8}$$

or

$$r = \left(\frac{3}{4\pi}\Omega_{\mathrm{CVAS}}\right)^{1/3}, \tag{9}$$

where r is the radius of main volume's sphere; see Fig. 8(b). The shell volume can be calculated from half of the shell volume of the main volume (due to force symmetry) with the shell radius equal to r_c:

$$\begin{aligned}\omega_{\mathrm{CVAS}} &= \frac{2}{3}\pi((r+r_{\mathrm{c}})^3 - r^3) \\ &= \frac{2}{3}\pi(r^3 + 3r^2 r_{\mathrm{c}} + 3r{r_{\mathrm{c}}}^2 + {r_{\mathrm{c}}}^3 - r^3) \\ &= 2\pi r^2 r_{\mathrm{c}} + 2\pi r{r_{\mathrm{c}}}^2 + \frac{2}{3}\pi{r_{\mathrm{c}}}^3\end{aligned} \quad (10)$$

Substitution of Eq. (10) with r from Eq. (9) yields

$$\omega_{\mathrm{CVAS}} = 2\pi\left(\frac{3}{4\pi}\Omega\right)^{2/3} r_{\mathrm{c}} + 2\pi\left(\frac{3}{4\pi}\Omega\right)^{1/3} {r_{\mathrm{c}}}^2 + \frac{2}{3}\pi{r_{\mathrm{c}}}^3, \quad (11)$$

where $\Omega = \Omega_{\mathrm{CVAS}}$. Let the dimensionless quantity β be

$$\beta = \frac{\Omega^{1/3}}{r_{\mathrm{c}}}, \quad (12)$$

or,

$$r_{\mathrm{c}} = \frac{\Omega^{1/3}}{\beta}. \quad (13)$$

Substituting Eq. (11) with r_{c} from Eq. (13), we obtain

$$\begin{aligned}\omega_{\mathrm{CVAS}} &= 2\pi\left(\frac{3}{4\pi}\Omega\right)^{2/3}\left(\Omega^{1/3}\beta^{-1}\right) \\ &\quad + 2\pi\left(\frac{3}{4\pi}\Omega\right)^{1/3}\left(\Omega^{2/3}\beta^{-2}\right) + \frac{2}{3}\pi\left(\Omega\beta^{-3}\right) \\ &= 2\pi\left(\frac{3}{4\pi}\right)^{2/3}\Omega\beta^{-1} + 2\pi\left(\frac{3}{4\pi}\right)^{1/3}\Omega\beta^{-2} + \frac{2}{3}\pi\Omega\beta^{-3} \\ &= \Omega\left(2\pi\left(\frac{3}{4\pi}\right)^{2/3}\beta^{-1} + 2\pi\left(\frac{3}{4\pi}\right)^{1/3}\beta^{-2} + \frac{2}{3}\pi\beta^{-3}\right) \\ &\approx \Omega\left(2.4180\beta^{-1} + 3.8978\beta^{-2} + 2.0944\beta^{-3}\right)\end{aligned} \quad (14)$$

Therefore, the total volume required by each thread in CVAS is

$$\begin{aligned}V_{\mathrm{t}}^{\mathrm{CVAS}} &= \Omega_{\mathrm{CVAS}} + \omega_{\mathrm{CVAS}} \\ &= \Omega_{\mathrm{CVAS}}\left(1 + 2.4180\beta^{-1} + 3.8978\beta^{-2} + 2.0944\beta^{-3}\right)\end{aligned} \quad (15)$$

4.2.2. Lower-bound memory consumption of BFAS algorithm

In the best case, Ω_{BFAS} forms a cubic shape. In this situation, the main volume obtained by each thread is

$$\Omega_{\mathrm{BFAS}} = l^3, \tag{16}$$

or

$$l = \Omega_{\mathrm{BFAS}}{}^{1/3}, \tag{17}$$

where l is the side length of the main volume; see Fig. 8(a). The shell volume can be calculated from the half of the cubic shell of the main volume with shell thickness equal to r_c:

$$\begin{aligned} \omega_{\mathrm{BFAS}} &= \frac{1}{2}((l+2r_c)^3 - l^3) \\ &= \frac{1}{2}(l^3 + 6l^2 r_c + 12 l r_c{}^2 + 8 r_c{}^3 - l^3) \\ &= 3l^2 r_c + 6 l r_c{}^2 + 4 r_c{}^3 \end{aligned} . \tag{18}$$

Substitution of Eq. (18) with l from Eq. (17) yields

$$\omega_{\mathrm{BFAS}} = 3\Omega^{2/3} r_c + 6\Omega^{1/3} r_c{}^2 + 4 r_c{}^3, \tag{19}$$

where $\Omega = \Omega_{\mathrm{BFAS}}$. Substituting r_c from Eq. (19) with β from Eq. (13), we obtain

$$\begin{aligned} \omega_{\mathrm{BFAS}} &= 3\Omega^{2/3}(\Omega^{1/3}\beta^{-1}) + 6\Omega^{1/3}(\Omega^{2/3}\beta^{-2}) + 4\Omega\beta^{-3} \\ &= \Omega(3\beta^{-1} + 6\beta^{-2} + 4\beta^{-3}) \end{aligned} . \tag{20}$$

Therefore, the total volume requires by each thread of BFAS is

$$\begin{aligned} V_{\mathrm{t}}^{\mathrm{BFAS}} &= \Omega_{\mathrm{BFAS}} + \omega_{\mathrm{BFAS}} \\ &= \Omega_{\mathrm{BFAS}}\left(1 + 3\beta^{-1} + 6\beta^{-2} + 4\beta^{-3}\right) \end{aligned} . \tag{21}$$

4.2.3. Memory scaling comparison between CVAS and BFAS

Using Eqs. (15) and (21), we can compare how the memory consumption of CVAS and BFAS scales. Let us assume that $\Omega = \Omega_{\mathrm{CVAS}} = \Omega_{\mathrm{BFAS}}$ for direct comparison. We substitute β from Eq. (13) into Eq. (15) to obtain

$$\frac{V_t^{CVAS}}{\Omega} = 1 + 2.4180\Omega^{-1/3} r_c + 3.8978\Omega^{-2/3} r_c^{\,2} + 2.0944\Omega^{-1} r_c^{\,3} \qquad (22)$$

Since each cell has volume of r_c^3, we express a main volume in terms of a dimensionless unit Ω* where

$$\Omega^* = \frac{\Omega}{r_c^3}. \qquad (23)$$

Substitution of Ω* from Eq. (23) into the right-hand side of Eq. (22) yields

$$\frac{V_t^{CVAS}}{\Omega} = 1 + 2.4180\Omega^{*-1/3} + 3.8978\Omega^{*-2/3} + 2.0944\Omega^{*-1} \qquad (24)$$

Similarly, using the same analysis on BFAS volume in Eq. (21), we obtain

$$\frac{V_t^{BFAS}}{\Omega} = 1 + 3\Omega^{*-1/3} + 6\Omega^{*-2/3} + 4\Omega^{*-1}. \qquad (25)$$

Figure 10 shows the scaling ratio of V_t/Ω as a function of Ω* for both algorithms. The result indicates a decreasing function of the ratio for both CVAS and BFAS. The ratio decreases sharply up to Ω* ~ 6 for both algorithms. For Ω* > 6, the ratio decreases more slowly. We observe that CVAS algorithm consumes less memory compared to the BFAS algorithm due to the geometric effect on the surface-to-volume ratio of spheres and cubes as seen in Figs. 9 (c) and (d). The memory consumption difference between CVAS and BFAS is significant for small Ω* and diminishes for larger Ω*. BFAS uses 18% more memory for Ω*=50, and less than 10% more memory when Ω*>269. This suggests that for a very large system, the relative memory difference between the algorithms is insignificant.

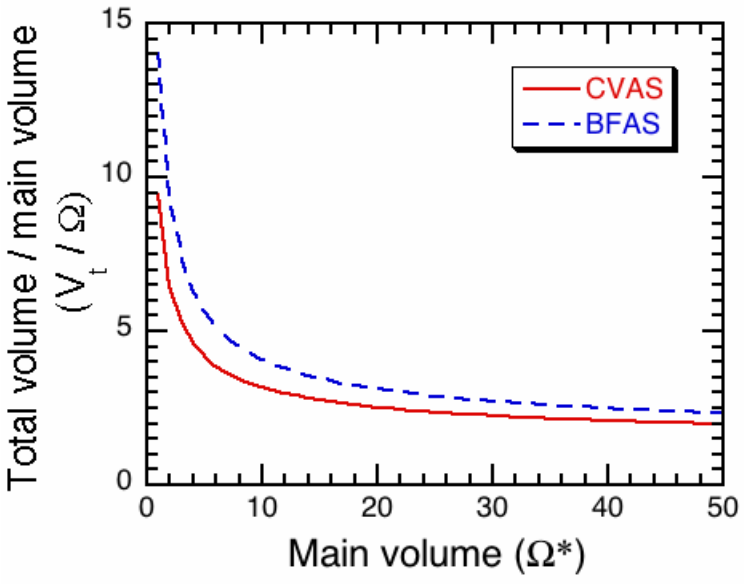

Fig. 10. The lower-bound memory consumption of CVAS and BFAS algorithms.

4.2.4. Asymptotic memory consumption of CVAS and BFAS

In this subsection, the asymptotic memory consumption of CVAS and BFAS will be analyzed. The memory consumption of CVAS and BFAS from Eqs. (15) and (21) can be expressed as

$$\begin{aligned} m_t &= \rho V_t \\ &= \rho\Omega\left(1 + a\beta^{-1} + b\beta^{-2} + c\beta^{-3}\right), \end{aligned} \tag{26}$$

where $a, b, c, \in \Re$ are pre-factors. Substitution of β from Eq. (13) into Eq. (26) yields

$$\begin{aligned} m_t &= \rho\Omega\left(1 + ar_c\Omega^{-1/3} + br_c^2\Omega^{-2/3} + cr_c^3\Omega^{-1}\right) \\ &= \rho\left(\Omega + a\Omega^{2/3} + b\Omega^{1/3} + c\right) \end{aligned} \tag{27}$$

.

Let n be the number of particles on a node and p be the number of threads per node. On average, each thread has n/p particles in the main volume Ω. From the assumption that the system has uniform density, then

$$\Omega = \frac{n}{p\rho}. \tag{28}$$

Substitution of Eq. (28) into Eq. (27) yields

$$
\begin{aligned}
m_t &= \rho\left(\frac{n}{p\rho} + a\left(\frac{n}{p}\right)^{2/3} + b\left(\frac{n}{p}\right)^{1/3} + c\right) \\
&= \frac{n}{p} + a\left(\frac{n}{p}\right)^{2/3} + b\left(\frac{n}{p}\right)^{1/3} + c \\
&= O\left(\frac{n}{p} + \left(\frac{n}{p}\right)^{2/3}\right)
\end{aligned}
\qquad (29)
$$

,

as the asymptotic memory consumption per thread or $O(n+n^{2/3}p^{1/3})$ per node. This asymptotic memory consumption of CVAS and BFAS is the same as the memory consumption of a traditional MPI-only scheme.

4.3. Computational complexity

Since the dynamic nature of MD leads to variation of density profile over time, the scheduling needs to be repeated in order to maintain its quality as the simulation progresses. In this section, the computational complexity of CVAS and BFAS are discussed.

For the CVAS algorithm, distances from the centroid to all the surrounding cells (*i.e.,* intermediate surface cells of the thread) need to be calculated to find the closest cell to its centroid. Focusing on the scheduling cost of one thread T_i, there is only one cell in the thread volume in the first step, which is the root cell. The number of distance computations scales as the surface area of one cell. In the second assignment phase of T_i, there are two cells in T_i, thus the number of distance computations scales as the surface area of two cells. In the last assignment step of T_i, T_i has L/p assigned cells, thus the number of distance computations scales as the surface area of L/p cells. Since the surface area scales as the 2/3 power of the volume, the number of distance computations of one thread is

$$
t_{\text{thread}} = 1 + (2)^{2/3} + (3)^{2/3} + \ldots + (\frac{L}{p})^{2/3} = \sum_{k=1}^{L/p} k^{2/3} \qquad (30)
$$

.

For p threads in one node, the cost is

$$
t_{\text{node}} = p t_{\text{thread}} = p \sum_{k=1}^{L/p} k^{2/3} \qquad (31)
$$

.

For large L/p, the summation can be approximated by integration. Therefore, the computation cost is approximated as

$$t_{\text{node}} = p\sum_{k=1}^{L/p} k^{2/3} \sim p\int_{0}^{L/p} k^{2/3}\,dk = p\left[k^{5/3}\right]_{k=0}^{L/p} = \frac{3}{2}p\left(\frac{L}{p}\right)^{5/3} = \Theta(L^{5/3}p^{-2/3}) \qquad (32)$$

When particles are uniformly distributed, the number of particles for each node n scales as the number of cells L. Therefore,

$$t_{\text{node}} = \Theta(\rho n^{5/3}p^{-2/3}) = \Theta(L^{5/3}p^{-2/3}) \qquad (33)$$

In the case of the BFAS algorithm, the scheduling cost of BFAS is identical to that of BFS graph traversal on the work assignment phase. The complexity of a generic BFS traversal is $\Theta(V+E)$, where V is the number of vertices and E is the number of edges. In the BFAS algorithm, the number of vertices is equivalent to the number of cells L and the number of edges is equal to $27L$. Although multiple BFS searches run simultaneously in BFAS, the total number of vertex traversals is L steps. Each traversal consumes a constant amount of neighbor-cell lookups. Thus, the running time of the BFAS algorithm scales as $\Theta(L) = \Theta(\rho n) = \Theta(n)$.

4.4. Selection criterion for CVAS and BFAS

We have analyzed the CVAS and BFAS algorithms in the earlier sections. However, for a particular simulation, it is still unclear which algorithm achieves better performance. In this section, we discuss the relative benefit of CVAS and BFAS based on the granularity of the simulation, n/p. This can be used as a selection criterion to choose the better algorithm for a different simulation setup.

We begin by focusing on memory consumption. According to the memory analysis in section 4.2, CVAS has the optimal geometry that minimizes the surface-to-volume ratio, while BFAS geometry is not optimal. Nevertheless, Fig. 9 indicates that the difference in memory consumption is notable only when the main volume (*i.e.,* granularity) is small. This difference diminishes when the size of the main volume increases. Asymptotically, the difference becomes negligible. In summary for memory consumption, CVAS consumes less memory for small granularity while the difference is small for large granularity.

Next, we consider the scheduling cost. Section 4.3 shows that the computational complexity of BFAS is linear while that of CVAS is superlinear in

terms of n. Clearly, BFAS is better than CVAS in terms of computation cost especially for large n/p. However, for small n/p, the computation costs of CVAS and BFAS are similar.

In conclusion, CVAS algorithm consumes considerably less memory than BFAS with a comparable scheduling cost for small n/p. In contrast, BFAS algorithm consumes asymptotically the same memory with much less scheduling cost when n/p is large. Therefore, CVAS and BFAS are suitable for small and large granularities, respectively. This is summarized in Table 1. In production simulations, n/p usually ranges from 100 – 10,000 particles, corresponding to the transition from small to large granularity.

Table 1: Selection criterion between CVAS and BFAS in term of granularity.

Granularity n/p	Advantage algorithm		
	Memory	Computation	Overall
Small	CVAS	Small difference	CVAS
Large	Identical	BFAS	BFAS

5. Performance Evaluation

In this section, we measure the performance of the CVAS and BFAS algorithms. Section 5.1 shows benchmarks of the scheduling cost of CVAS and BFAS algorithms. Section 5.2 measures the load-imbalance factor of our scheduler, and section 5.3 measures the memory-footprint reduction for CVAS, confirming its $O(n+p^{1/3}n^{2/3})$ memory scaling. Section 5.4 demonstrates the reduction of the scheduling cost of CVAS without affecting the quality of load balancing, followed by strong-scaling comparison of the hybrid MPI/OpenMP and MPI-only schemes in section 5.5 for CVAS.

Performance evaluations in this section are performed on three testing platforms: (1) Dawn, a BlueGene/P cluster at LLNL; (2) Hera, a quad 4-core AMD Opteron cluster at LLNL; and (3) HPCC, a dual 6-core Intel Xeon cluster at the University of Southern California (USC). The LLNL-Dawn cluster is used for large-scale testing, while LLNL-Hera and USC-HPCC provide higher thread count environments for AMD and Intel architectures, respectively. Detailed specifications for each cluster are provided in Table 2.

Table 2: Specifications of testing clusters.

	Platforms		
	LLNL-Dawn	LLNL-Hera	USC-HPCC
Processor	IBM PowerPC 450	AMD Opteron 8356	Intel Xeon X5650
Clock Speed	850 MHz	2.3 GHz	2.66 GHz
Number of nodes	36,864	864	256
Cores per node	4	16	12
Memory per node	4 GB	32 GB	24 GB
Network	IBM 3D torus	Infiniband DDR	Myrinet 10G

5.1. Scheduling cost

We have measured the average scheduling time for a 128,000-particle system over 1,000 MD steps on 64 BlueGene/P nodes. Figure 11 shows the average scheduling cost when the scheduling is performed every 15 steps. We observe that CVAS spends 8.1, 1.5, and 9.8% more time on scheduling than BFAS for the number of threads p = 2, 3, and 4, respectively. Note that for p = 1, we simply schedule all the work to one thread.

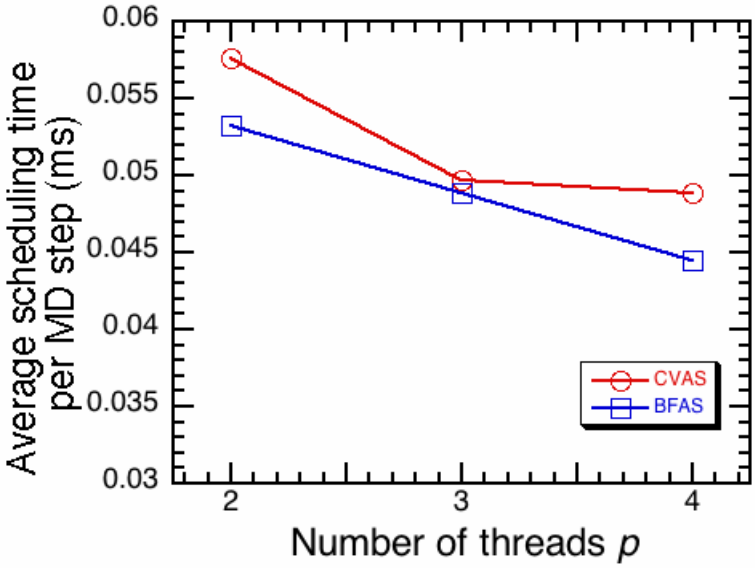

Fig. 11. Scheduling cost for the CVAS and BFAS algorithms for 128,000-particle system on 64 BlueGene/P nodes. The circles and squares are the average scheduling times of CVAS and BFAS, respectively.

5.2. Thread-level load balancing

We perform a load-balancing test for CVAS on a dual six-core 2.3 GHz AMD Opteron with n = 8,192. In Fig. 12, the measured load-imbalance factor γ is plotted as a function of p, along with its estimator introduced in section 3.2.1 and the theoretical bound, Eq. (5). 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 performance fluctuates slightly depending on the selection of root nodes in CVAS algorithm. While random root selection tends to provide robust performance compared to deterministic selection, it is possible to use some optimization techniques 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 [32].

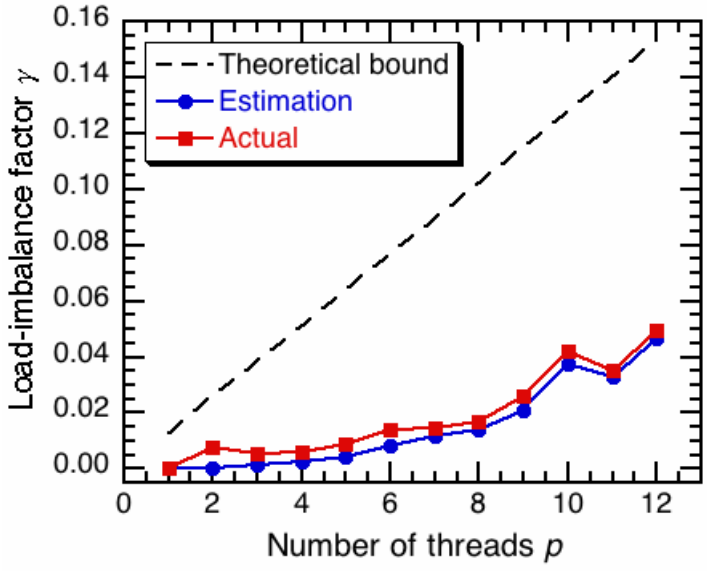

Fig. 12. The load-imbalance factor γ of CVAS as a function of p from theoretical bound, scheduler estimation, and actual measurement.

5.3. Memory consumption

To test the memory efficiency of the CVAS algorithm, we perform simulations on a four quad-core 2.3 GHz AMD Opteron 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 12 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 naïve 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 naïve $\Theta(np)$ memory per-node requirement. In Fig. 13, the dashed curves show the reduction of memory requirement per thread estimated as

$$m = ap^{-1} + bp^{-2/3} \tag{34}$$

where the first term represents the memory scaling from main cells and the second term represents scaling from surface cells of each thread, see Eq. (29). In Eq. (34), a and b are fitting parameters. The regression curves fit the measurements well, indicating that the memory requirement is accurately modeled by $O(n+p^{1/3}n^{2/3})$.

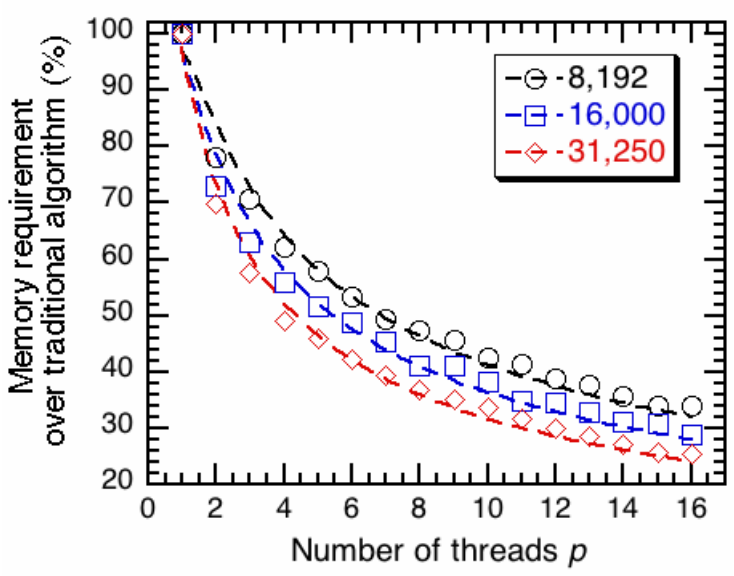

Fig. 13. Average memory consumption for the private force arrays as a function of p using CVAS compared to the naïve method. Numbers in the legend denote n.

5.4. Reduction-sum operation cost

We also measure the computation time spent for the reduction sum of the private force arrays to obtain the global force array. Figure 14 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,

$$t_{\text{reduction}} = ap^{1/3} + b \tag{35}$$

where a and b are fitting parameters, which fit all cases well.

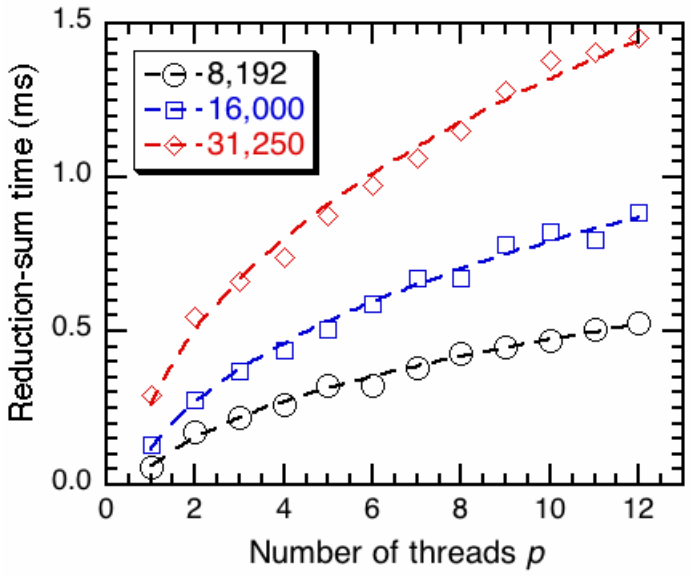

Fig. 14. Average reduction-sum operation time of the CVAS as a function of p. Numbers in the legend denote n.

5.5. Strong-scaling benchmark

In strong-scaling benchmarks of CVAS, where the problem size is fixed, Figure 15(a) shows the thread-level strong-scaling speedup on a four quad-core 2.3 GHz AMD Opteron. The algorithm achieves a speedup of 14.43 on 16 threads, *i.e.*, the strong-scaling multi-threading parallel efficiency is 0.90. CVAS 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 850 MHz PowerPC 450 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 15 (b) and (c) show the running time of 843,750 and 1,687,500 particles systems, respectively, 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 even slower. The MPI/OpenMP code shows 2.58× and 2.16×

speedups 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. The larger running time of

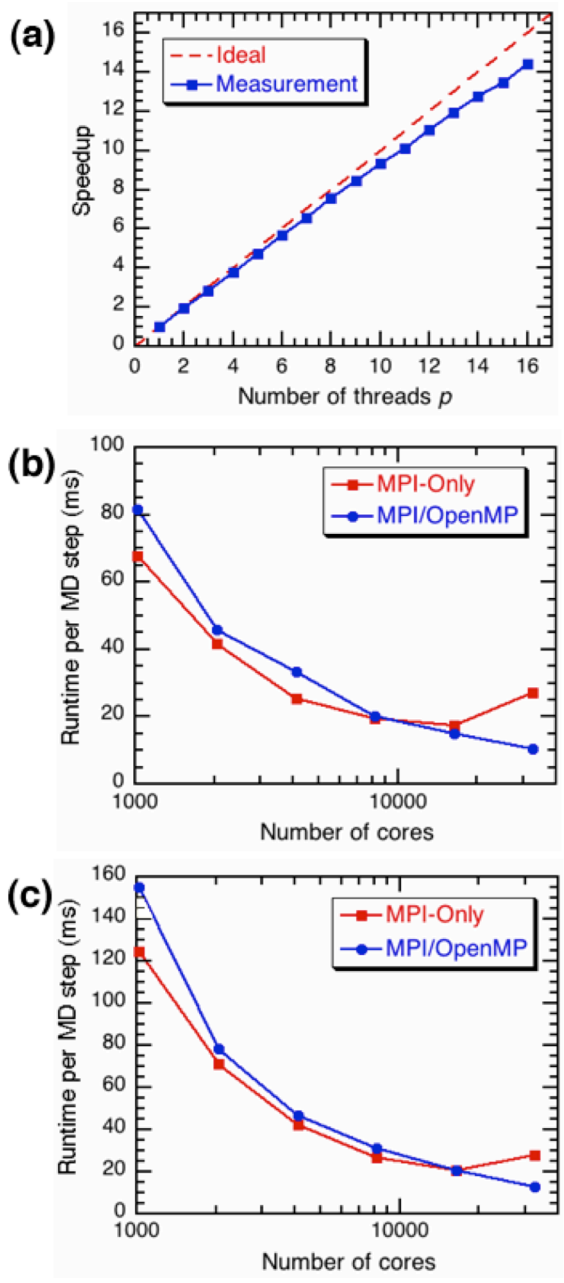

Fig. 15. (a) Thread-level strong scalability of the parallel section on a four quad-core AMD Opteron 2.3 GHz with fixed problem size at n = 8,192 particles. Total running time per MD steps on 1,024 – 32,768 Power PC 450 850 MHz cores of BlueGene/P for a fixed problem size at N = 0.84-million particles (b) and 1.68-million particles (c).

the hybrid MPI/OpenMP code compared with that of the MPI-only code for small number of nodes in Figs. 15(b) and (c) can be understood as a result of the Amdahl's law. Namely, only the pair kernel of the ddcMD code 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 associated with the spatial-decomposition domain is larger for the MPI-only code compared to that of the hybrid MPI/OpenMP code; and (2) the aggregated 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.

6. Summary

We have designed two new data-privatization algorithms, CVAS and BFAS, based on a nucleation-growth allocation, which have minimal memory footprint. Both algorithms have been thoroughly analyzed and compared. The memory consumption per thread is found to scale as $O(n+p^{1/3}n^{2/3})$ for both CVAS and BFAS. However, geometric effects cause BFAS to consume more memory compared to CVAS. Our analysis shows that the computational costs of CVAS and BFAS are bounded by $\Theta(n^{5/3}p^{-2/3})$ and $\Theta(n)$, respectively. These analyses identify an optimal range for CVAS and BFAS in terms of granularity, n/p: CVAS is the better choice for smaller granularity while BFAS is preferred for larger granularity. Massively parallel MD benchmarks have demonstrated significant performance benefits of the hybrid MPI/OpenMP parallelization for fine-grain large-scale applications. Namely, 2.58× speedup has been observed for 0.8-million particle simulation on 32,768 cores of BlueGene/P.

Acknowledgements 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-JRNL-528373). The work at USC was partially supported by DOE BES/EFRC/SciDAC/SciDAC-e/INCITE and NSF PetaApps/CDI.

References

1. Phillips JC, Zheng G, Kumar S, Kale' LV (2002) NAMD: Biomolecular simulations on thousands of processors. In: Supercomputing, Los Alamitos, CA
2. Bowers KJ, Dror RO, Shaw DE (2007) Zonal methods for the parallel execution of range-limited N-body simulations. J Comput Phys 221 (1):303-329

3. Hess B, Kutzner C, van der Spoel D, Lindahl E (2008) GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. J Chem Theory Comput 4 (3):435-447
4. Shaw DE, Dror RO, Salmon JK, Grossman JP, Mackenzie KM, Bank JA, Young C, Deneroff MM, Batson B, Bowers KJ, Chow E, Eastwood MP, Ierardi DJ, Klepeis JL, Kuskin JS, Larson RH, Lindorff-Larsen K, Maragakis P, Moraes MA, Piana S, Shan Y, Towles B (2009) Millisecond-scale molecular dynamics simulations on Anton. In: Supercomputing, Portland, OR
5. Nomura K, Dursun H, Seymour R, Wang W, Kalia RK, Nakano A, Vashishta P, Shimojo F, Yang LH (2009) A metascalable computing framework for large spatiotemporal-scale atomistic simulations. In: International Parallel and Distributed Processing Symposium
6. Kushima A, Lin X, Li J, Eapen J, Mauro JC, Qian X, Diep P, Yip S (2009) Computing the viscosity of supercooled liquids. The Journal of Chemical Physics 130:224504: 224501-224503
7. Wang W, Clark R, Nakano A, Kalia RK, Vashishta P (2009) Multi-Million Atom Molecular Dynamics Study of Combustion Mechanism of Aluminum Nanoparticle. In: Material Research Society Symposium Proceeding
8. Streitz FH, Glosli JN, Patel MV, Chan B, Yates RK, de Supinski BR, Sexton J, Gunnels JA (2006) Simulating solidification in metals at high pressure: The drive to petascale computing. J Phys: Conf Ser 46:254-267
9. Brown WM, Kohlmeyer A, Plimpton SJ, Tharrington AN (2012) Implementing molecular dynamics on hybrid high performance computers - Particle-particle particle-mesh. Comput Phys Commun 183 (3):449-459
10. Alam SR, Agarwal PK, Hampton SS, Ong H, Vetter JS (2008) Impact of multicores on large-scale molecular dynamics simulations. In: International Parallel and Distributed Processing Symposium, Miami, FL
11. Fuller SH, Millett LI (2011) Computing performance: game over or next level? Computer 44 (1):31-38
12. Peng L, Kunaseth M, Dursun H, Nomura K, Wang W, Kalia RK, Nakano A, Vashishta P (2009) 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, NV
13. Chorley MJ, Walker DW, Guest MF (2009) Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters. Int J High Perform C 23 (3):196-211
14. Rabenseifner R, Hager G, Jost G (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. Euromicro Workshop P:427-436
15. Osthoff C, Grunmann P, Boito F, Kassick R, Pilla L, Navaux P, Schepke C, Panetta J, Maillard N, Silva Dias PL, Walko R (2011) Improving Performance on Atmospheric Models through a Hybrid OpenMP/MPI Implementation. In: International Symposium on Parallel and Distributed Processing with Applications
16. Glosli JN, Richards DF, Caspersen KJ, Rudd RE, Gunnels JA, Streitz FH (2007) Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability. In: Supercomputing, Reno, NV
17. York D, Yang W (1994) The fast Fourier Poisson method for calculating Ewald sums. The Journal of Chemical Physics 101 (4):3298-3300
18. Hockney R, Eastwood J (1981) Computer simulation using particles. McGraw-Hill, New York
19. Darden T, York D, Pedersen L (1993) Particle mesh Ewald: An N log(N) method for Ewald sums in large systems. The Journal of Chemical Physics 98 (12):10089-10092
20. Richards DF, Glosli JN, Chan B, Dorr MR, Draeger EW, Fattebert J-L, Krauss WD, Spelce T, Streitz FH, Surh MP, Gunnels JA (2009) Beyond homogeneous decomposition: scaling long-range forces on massively parallel systems. In: Supercomputing, Portland, OR
21. Fattebert J-L, Richards DF, Glosli JN (2012) Dynamic load balancing algorithm for molecular dynamics based on Voronoi cells domain decompositions. Comput Phys Commun 183 (12):2608-2615
22. Mellor-Crummey J, Whalley D, Kennedy K (2001) Improving memory hierarchy performance for irregular applications using data and computation reorderings. International Journal of Parallel Programming 29 (3):217-247
23. Peng L, Kunaseth M, Dursun H, Nomura K, Wang WQ, Kalia RK, Nakano A, Vashishta P (2011) Exploiting hierarchical parallelisms for molecular dynamics simulation on multicore clusters. J Supercomput 57 (1):20-33

24. Penmatsa S, Chronopoulos AT, Karonis NT, Toonen B (2007) Implementation of distributed loop scheduling schemes on the TeraGrid. In: International Parallel and Distributed Processing Long Beach, CA
25. Ciorba FM, Andronikos T, Riakiotakis AT, Papakonstantinou G (2006) Dynamic multi-phase scheduling for heterogeneous clusters. In: International Parallel and Distributed Processing Rhodes, Greece
26. Sunarso A, Tsuji T, Chono S (2010) GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows. J Comput Phys 229 (15):5486-5497
27. Yang J, Wang Y, Chen Y (2007) GPU accelerated molecular dynamics simulation of thermal conductivities. J Comput Phys 221 (2):799-804
28. Hu C, Liu Y, Li J (2009) Efficient parallel implementation of molecular dynamics with embedded atom method on multi-core platforms. In: International Conference on Parallel Processing Workshops
29. Holmes DW, Williams JR, Tilke P (2010) An events based algorithm for distributing concurrent tasks on multi-core architectures. Comput Phys Commun 181 (2):341-354
30. Madduri K, Williams S, Ethier S, Oliker L, Shalf J, Strohmaier E, Yelicky K (2009) Memory-efficient optimization of Gyrokinetic particle-to-grid interpolation for multicore processors. In: Supercomputing, Portland, OR
31. Kleinberg J, Tardos E (2005) Algorithm Design. 2 edn. Pearson Education, Inc.,
32. Catalyurek UV, Boman EG, Devine KD, Bozdag D, Heaphy R, Riesen L (2007) A Hypergraph-based dynamic load balancing for adaptive scientific computations. In: International Parallel and Distributed Processing Symposium