Computational Particle Mechanics manuscript No.
(will be inserted by the editor)

Comparison between Pure MPI and Hybrid MPI-OpenMP Parallelism for Discrete Element Method (DEM) of Ellipsoidal and Poly-ellipsoidal Particles

Beichuan Yan · Richard A. Regueiro

Received: date / Accepted: date

Abstract Parallel computing of 3D Discrete Element Method (DEM) simulations can be achieved in different modes, and two of them are pure MPI and hybrid MPI-OpenMP. The hybrid MPI-OpenMP mode allows flexibly-combined mapping schemes on contemporary multiprocessing supercomputers. This paper profiles computational components and floating point operation features of complex-shaped 3D DEM, develops a space decomposition based MPI parallelism and various thread-based OpenMP parallelism, and carries out performance comparison and analysis from intranode to internode scales across four orders of magnitude of problem size (namely, number of particles). The influences of memory/cache hierarchy, processes/threads pinning, variation of hybrid MPI-OpenMP mapping scheme, ellipsoid vs poly-ellipsoid are carefully examined. It is found that OpenMP is able to achieve high efficiency in interparticle contact detection, but the unparallelizable code prevents it from achieving the same high efficiency for overall performance; pure MPI achieves not only lower computational granularity (thus higher spatial locality of particles) but also lower communication granularity (thus faster MPI transmission) than hybrid MPI-OpenMP using the same computational resources; the cache miss rate is sensitive to the memory consumption shrinkage per processor, and the last level cache (LLC) contributes most significantly to the strong superlinear speedup among all of the three cache levels of modern microprocessors; in hybrid MPI-OpenMPI mode, as the number of MPI processes increases (and the number of threads per MPI processes decreases accordingly), the total execution time decreases, until the maximum performance is obtained at pure MPI mode; the processes/threads pinning on NUMA architectures improves performance significantly when there are multiple threads per process, whereas the improvement becomes less pronounced when the number of threads per process decreases; both the communication time and computation time increase substantially from ellipsoids to poly-ellipsoids. Overall, pure MPI outperforms hybrid MPI-OpenMP in 3D DEM modeling of ellipsoidal and poly-ellipsoidal particles.

Keywords parallelism · discrete element · ellipsoidal · MPI · OpenMP · granularity

1 INTRODUCTION

Parallel computing of 3D DEM has become an indispensable trend in numerical simulations of granular materials that involve a large number of particles, and it is particularly needed for modeling complex-shaped particles such as axisymmetric ellipsoids (Ng, 1994, 2004), true ellipsoids (Yan et al, 2010), poly-ellipsoids (Peters et al, 2009),

Beichuan Yan
Department of Civil, Environmental, and Architectural Engineering, University of Colorado Boulder
Tel.: +1-303-588-9566
Fax: +1-303-492-7317
E-mail: beichuan.yan@colorado.edu

Richard A. Regueiro
Department of Civil, Environmental, and Architectural Engineering, University of Colorado Boulder
E-mail: richard.regueiro@colorado.edu

superellipsoids (Wellmann et al, 2008; Delaney et al, 2010), superquadrics (Williams and Pentland, 1992) or asymmetrical particles constructed by non-uniform rational Basis-Splines (NURBS) (Lim and Andrade, 2014), because of their expensive computational cost in interparticle contact resolution. A serial 3D DEM code is typically limited to simulating from a few to approximately ten thousand particles (i.e., 21 x 21 x 21 particle array in 3D space), owing to the following factors: non-linear and history-dependent interparticle contact mechanical models, complicated contact geometry resolution between particle pairs, normally adopted explicit time integration scheme, and associated small time step. However, a parallel 3D DEM is able to expand the simulation scale or problem size (in terms of number of particles) to 10^6 or 10^7 on contemporary supercomputers.

There has been considerable interest in developing and utilizing parallel DEM codes in recent years (Henty, 2000; Baugh Jr and Konduri, 2001; Washington and Meegoda, 2003; Maknickas et al, 2006). Vedachalam and Virdee (2011) used LAMMPS (large-scale atomic and molecular massively parallel simulator, developed by Sandia National Laboratories) and LIGGGHTS (LAMMPS improved for general granular and granular heat transfer simulations) to study the motion of snow particles, wherein the snow grains are assumed to be spherical particles of 5 mm diameter. An empirical coefficient of restitution (ratio of rebound velocity to impact velocity) is adopted rather than the strict Hertzian nonlinear contact model, while Mindlin's history-dependent shear model is not considered. With regard to the performance gain of parallelism, the authors wrote "on 480 processors for 75K particles, the speedup was 1.99, while on 960 processors for same number of particles speedup achieved was 2.52" in comparison to 120 processors, which is a relatively low performance gain.

We emphasize that this work is focused on complex-shaped DEM, not Molecular Dynamics (MD). MD and DEM are essentially very different computational methods in that: (a) MD simulations have become prominent because of the availability of accurate interatomic potentials for a range of materials, whereas granular particles in DEM generally interact with each other through direct contacts and friction. In DEM simulations the significant majority of floating-point operations is consumed on contact geometry resolution, while that is not the case for MD where equivalent spheres are used; (b) rotation of granular particles plays a critical role in determining the deformation and strength of particle assemblies, and non-linear and history-dependent interparticle contact mechanical models need to be employed to reflect this behavior, whereas that is not the case with atoms or molecules in MD; (c) granular materials are usually frictional materials, for which particle size, shape, size distribution and change of displacement or force boundary conditions have strong bearing on the assembly mechanical properties.

Chorley and Walker (2010) tested the hybrid DL_POLY (a large scale Molecular Dynamics code) on multicore clusters comprised of 256 compute nodes linked by a 20GB/s Infiniband interconnect, and found that: at lower processor numbers, the extra overheads from shared memory threading in the hybrid code outweigh performance benefits gained over the pure MPI code; on larger core counts the hybrid model performs better than pure MPI, with reduced communication time decreasing the overall runtime.

Pal et al (2014) conducted hybrid MPI-OpenMP based molecular dynamics simulations on a cluster of 6 dual-quad-core blade servers (SMP nodes) connected using Infiniband. They showed that a coupled scheme can work nearly twice as fast as a pure message-passing based implementation for certain system sizes, owing to the additional overheads in the latter being circumvented by the former scheme. For larger systems, however, with increasing work load per SMP node, the performance differential may become negligible.

Henty (2000) chose to investigate the performance of a much smaller test code that implements precisely the same algorithm but has limited functionality rather than tackle a complete physics DEM application with all of its functionality and complexity. He implemented a hybrid parallelization of the message-passing and shared-memory models simultaneously for spherical particles. The superlinear speedup is observed in his tests. The author pointed out: "The results are somewhat disappointing, showing that the pure MPI code is always more efficient for a given granularity" than a hybrid scheme. He attributes the poor performance of hybrid parallelism to OpenMP overhead and atomic locks in force calculation (including particle contact detection) at fine computational granularity.

All these efforts on parallelism of 3D DEM have only modeled particles as simple spheres. However, modeling complex-shaped particles could make a big difference in that the contact resolution between complex-shaped particles is much more computationally demanding. For instance, contact resolution between true ellipsoids is approximately 50 times as expensive as that of spheres, and contact resolution between poly-ellipsoids is nearly 300 times as expensive as that of spheres. The floating-point operation gap by orders of magnitude

between complex-shaped particles and simple spheres changes the internal computational structure of 3D DEM significantly, and the parallel computing performance and efficiency need to be analyzed accordingly.

On contemporary multiprocessing supercomputers there are two main parallel programming models: message-passing model and directives-based data-parallel model. They are differentiated by how the address space of the parallel computer is organized. The message-passing model is the most commonly used model for parallel programming on distributed-memory architectures, and Message Passing Interface (MPI (Gropp et al, 1999; Pacheco, 1997; Michael, 2003)) is the standard Application Programming Interface (API) for message passing. In the directives-based data parallel model, programming languages make serial code parallel by adding directives, which tell the compiler how to distribute data and work across the processors. A standard API that uses this model for parallel programming on shared-memory architectures is OpenMP (Dagum and Enon, 1998; Chandra, 2001; Michael, 2003). The directives in OpenMP are primarily used for parallelizing iteration loops.

A newer approach in parallel programming is the hybrid or combination of MPI and OpenMP within a single parallel-programming model (Michael, 2003; Jost et al, 2003; Drosinos and Koziris, 2004; Rabenseifner et al, 2009; Luecke et al, 2010). To fully exploit the potential of multiprocessing clusters, this combination of models relies on data distribution and explicit message passing between the nodes of a cluster ("internode" mode) as well as shared-memory and multithreading within the nodes ("intranode" mode). Michael (2003) pointed out that in many cases hybrid MPI-OpenMP programs execute faster than pure MPI programs as a result of lower communication overhead and improved load balance. He studied two problems on a commodity cluster containing four dual-processor nodes: (1) conjugate gradient method in solving a system of linear equations, where the hybrid program exhibits lower performance using less than 4 processors, but shows higher performance using more than 6 processors, than the pure MPI program; (2) Jacobi method in solving steady-state heat distribution, where the hybrid program is uniformly faster than the pure MPI program.

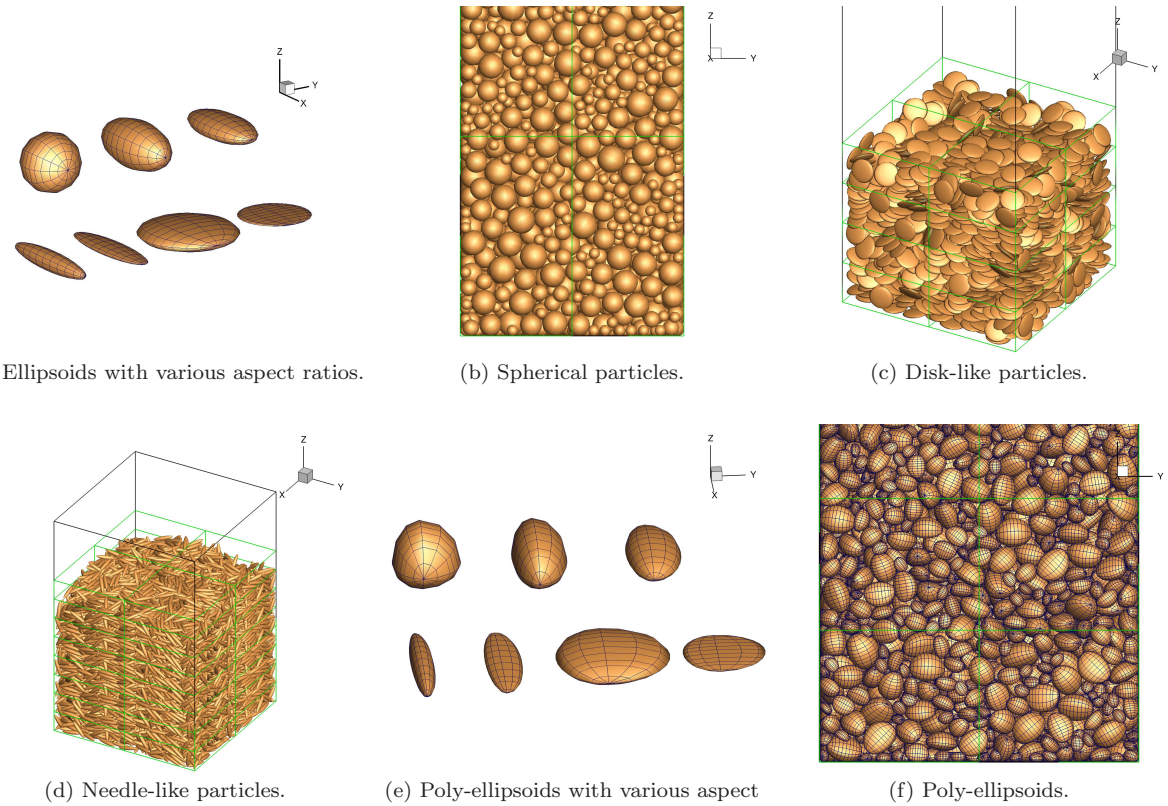
This paper presents performance comparison and analysis between pure MPI and hybrid MPI-OpenMP parallelism for 3D DEM of ellipsoidal and poly-ellipsoidal particles across four orders of magnitude of simulation scale. It contains ten sections: Section 1 has stated the motivation of 3D DEM parallelism and two different programming models; Section 2 is a brief introduction to DEM, focusing on its computational feature and challenge; Section 3 describes the MPI design methodology based on the link-block algorithm and gives a flowchart of the parallel DEM code; Section 4 first gives serial code profiling, and then presents various OpenMP implementations and evaluations; Section 5 presents performance analysis of OpenMP using a static simulation of 2.5k ellipsoidal particles, and reveals that the unparallelizable code puts a ceiling on speedup; Section 6 compares the single-node performance between MPI and OpenMP using a dynamic simulation of 2.5k ellipsoidal particles; Section 7 compares multinode performance between pure MPI and hybrid MPI-OpenMP across 1.2k, 150k and 1 million particles, profiles execution time of various modules of 3D DEM, and analyzes the causes of the performance difference; Section 8 investigates the effect of different mapping schemes/granularities, along with the influence of NUMA architectures. Section 9 compares the hybrid mode performance between ellipsoids and poly-ellipsoids; a summary is given in the last section.

2 DEM FUNDAMENTALS

A typical procedure of DEM analysis consists of three major computational steps in sequence, which are integrated in time using central difference method until a simulation is completed:

- contact detection between particles, including two phases:
  1. *neighbor estimate*
  2. *contact resolution*
- contact force computation for each pair of particles in contact.
- particle motion update (translations and rotations) using Newton's second law.

The contact detection process is usually the major computational bottleneck, especially for a large number of complex-shaped particles. It is divided into two phases: the neighbor estimate phase, and the contact resolution phase. Neighbor estimate identifies objects near the target object. It often uses an approximate geometry for the objects, such as bounding box or bounding sphere. The geometric contact resolution phase then uses a specific geometric representation of each body to resolve the contact geometry.



(a) Ellipsoids with various aspect ratios. (b) Spherical particles. (c) Disk-like particles.

(d) Needle-like particles. (e) Poly-ellipsoids with various aspect ratios. (f) Poly-ellipsoids.

Fig. 1 Ellipsoids and poly-ellipsoids represent a wide variety of shapes in DEM.

The neighbor estimate can have different time complexities: $O(n^2)$, resulting from n-by-n simple search; $O(n\log n)$, resulting from *tree-based algorithms* (Jagadish et al, 2005; Muja and Lowe, 2009); and $O(n)$, resulting from *binning algorithms* (Munjiza and Andrews, 1998; Williams et al, 2004) or link-cell (LC) method (Grest et al, 1989). The LC method divides a computational spatial domain into equal cubical cells of size not smaller than the cutoff distance (in MD) or diameter of the largest particle (in DEM). Each particle is referenced to the cell according to the position of the particle centroid. The neighbor estimate comprises referencing of individual particles to the cells and constructing of the neighbors list of particles using surrounding cells. It is worth noting that the overall performance improvement from these methods of different time complexity might be highly limited for complex-shaped particles, because neighbor estimate only takes up a small fraction in floating-point operations.

Yan et al (2010) developed a robust contact resolution algorithm for three-axis ellipsoidal particles by constructing an *extreme value problem* of finding the deepest penetration of one particle into the other, as shown in Figure 2. Such an extreme value problem results in a sixth order polynomial equation. Conventional polynomial root finders cannot satisfy the high-precision numerical requirement in the 3D DEM computation. For example, the elastic overlap between two particles of typical quartz sand may vary between 10^{-8} to 10^{-5} meters depending on particle size, shape and external force, and a low-precision solver can lead to numerical instability or spurious explosion of particles. Therefore, an iterative eigenvalue method is selected to find roots of the polynomial and determine the contact geometry. The algorithm and its implementation has been shown to be robust such that it is applicable to not only regularly bulky ellipsoidal shapes but also extreme-shaped ellipsoidal particles such as disks and needles, as shown in Figure 1(a∼d).

Peters et al (2009) proposed a non-symmetric poly-ellipsoid shape which joins eight component ellipsoids in eight different octants respectively to produce continuous surface coordinates, normal directions and inter-

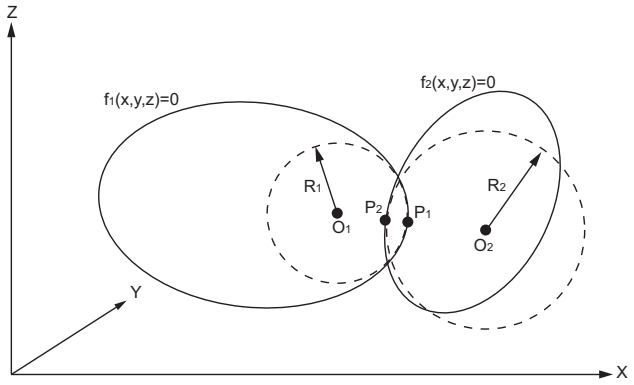


Fig. 2 Contact between two ellipsoidal particles.

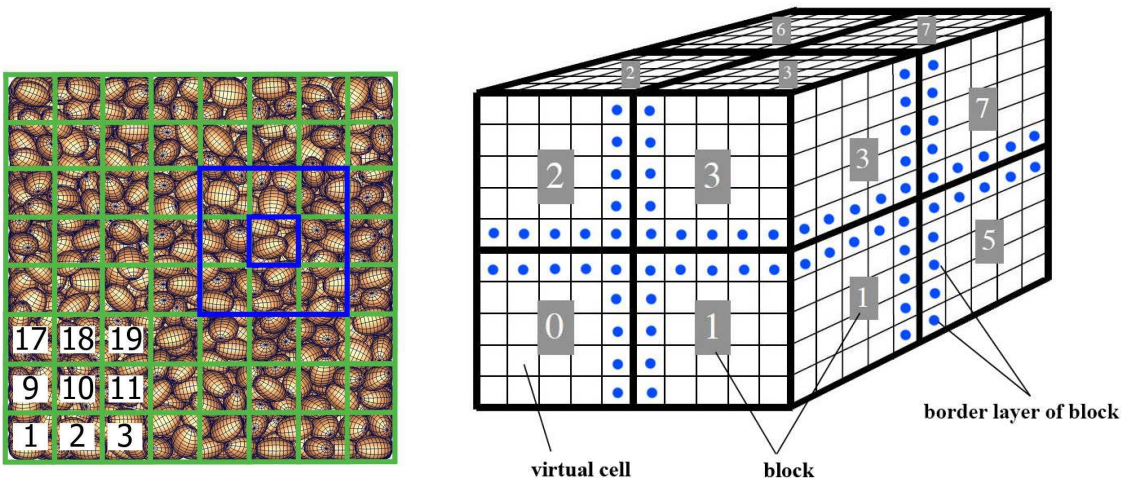
sections. It is more computationally expensive than a symmetric ellipsoid but it acts as a useful extension, as shown in Figure 1(e∼f).

As pointed out in Section 1, contact resolution between complex-shaped particles (Yan et al, 2010; Peters et al, 2009) is orders of magnitude more expensive than that of simple spheres. However, complex-shaped particles can represent a much wider range of realistic shapes in 3D DEM simulations, as shown by ellipsoids and poly-ellipsoids in Figure 1.

3 MPI DESIGN

3.1 Link-block and four-step MPI design

We upscale the link-cell (LC) to link-block (LB) technique in parallel DEM, and apply Foster's four-step paradigm in designing a parallel algorithm in DEM.



(a) Link-cell.

(b) Link-blocks, virtual cells and border layers.

Fig. 3 Upgrade from link-cell to link-block.

Partitioning: The computational domain is divided into blocks using space decomposition. Each block may consist of many virtual cells. In Figure 3(b), there are 8 blocks numbered from 0 to 7, each containing 5 x 5 x 5 small virtual cells. The size of the virtual cells is chosen to be the maximum diameter of the discrete particles.

Communication: Each cell, as a primitive task unit, can communicate with 26 possible surrounding ones to determine contact detection. However, the communication manner is changed after the process of agglomeration.

Agglomeration: Agglomeration is the process of grouping primitive tasks into larger tasks in order to improve performance or simplify programming. By combining 5 x 5 x 5 virtual cells into a block, communication overhead is lowered in that each block only needs to communicate with neighboring blocks through border/ghost layers, which are virtual cells marked by blue dots shown in Figure 3(b).

Mapping: Mapping is the assignment of agglomerated tasks to processors. Referring to the architecture in Figure 4, there are choices of mapping a block of particles to a core, a CPU, multiCPUs within a node, or even a whole node. If a process is assigned to more than a core (multicores, one CPU, multiCPUs, or all CPUs on a node), then multiple threads can run in that process, thus allowing a hybrid combination of processes and threads. Apparently different hierarchies of the hybrid combination exist depending on how a process is assigned. Very often each block is mapped to a whole compute node.

The communications between processes, i.e., interblock communications, are carried out by MPI. Within a process multiple threads are run using OpenMP. When each process (or each block) is mapped to a compute node, OpenMP makes the parallelism within the node, so-called "intranode" parallelism, and MPI makes the parallelism between nodes, so-called "internode" parallelism.

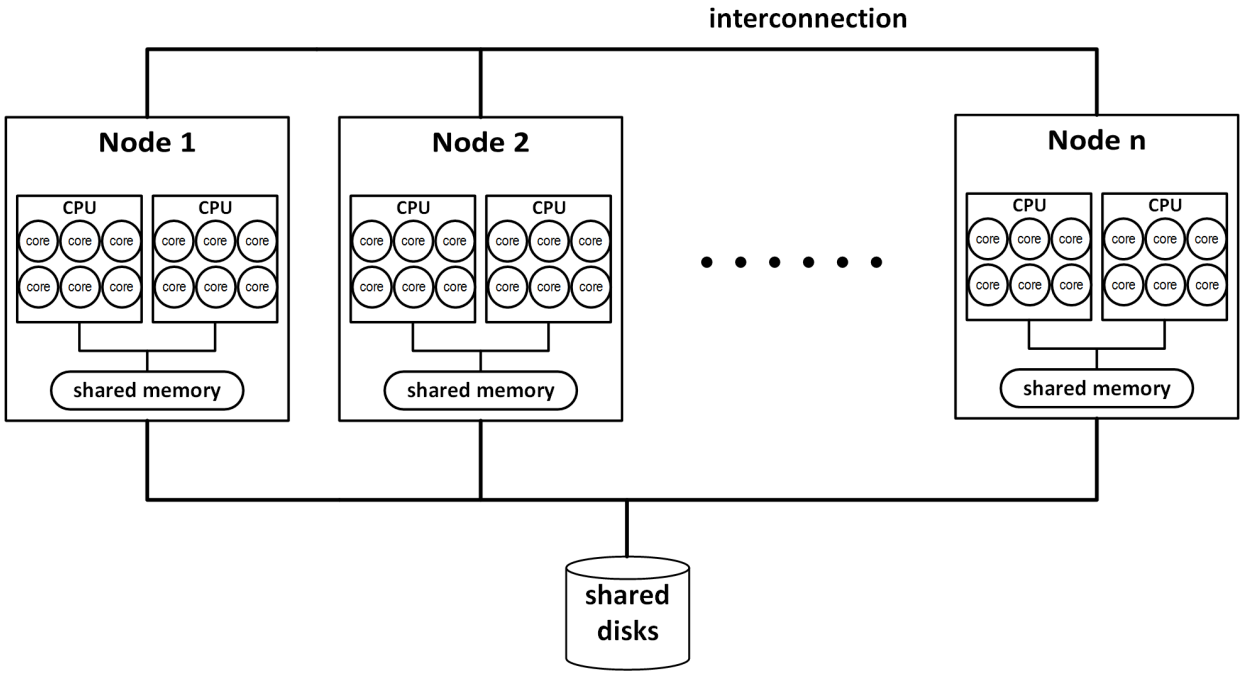
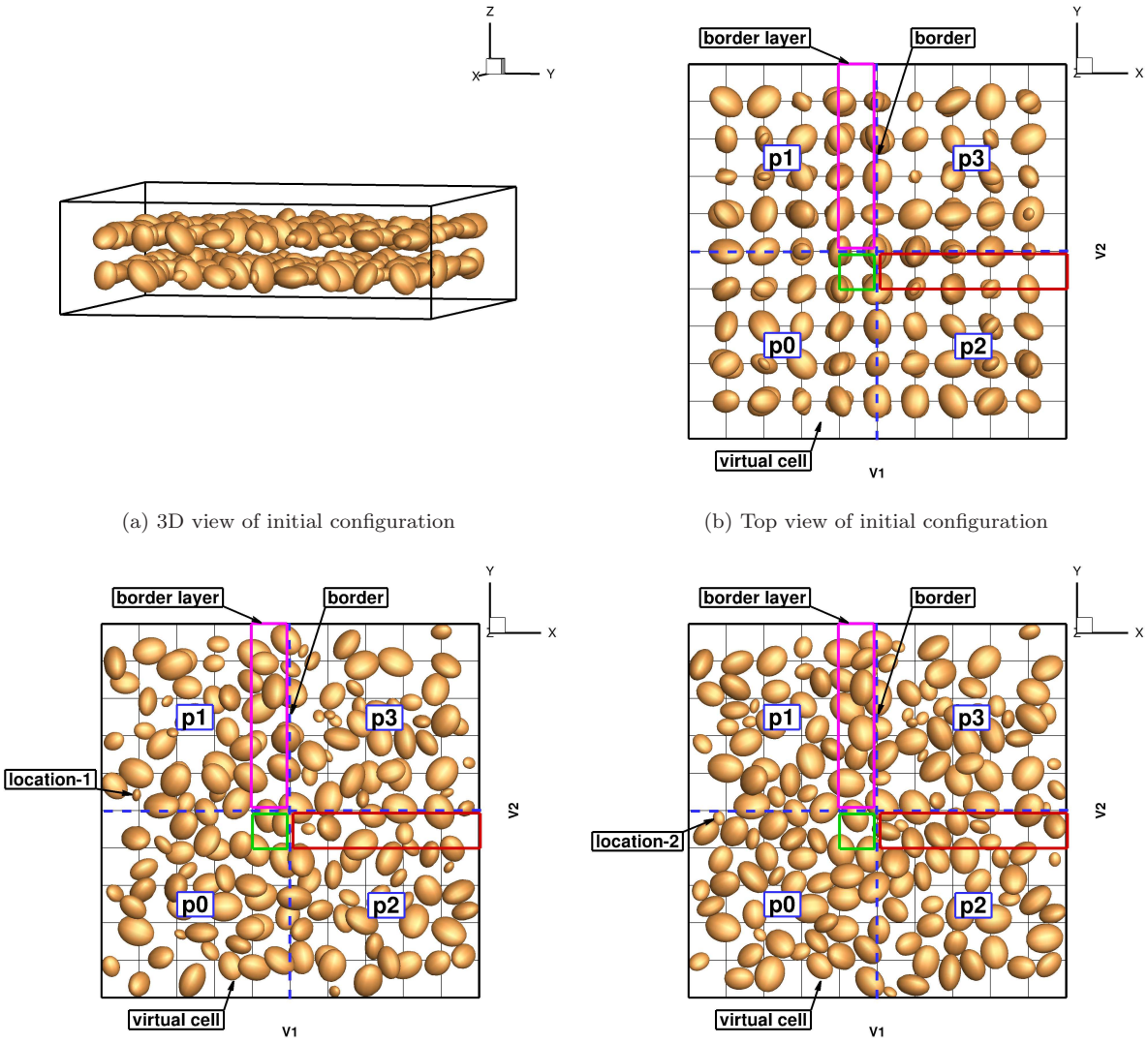


Fig. 4 Schematic of typical architecture of a modern supercomputer.

3.2 Interblock communication

In Figure 3, beware that a border/ghost layer is not limited to constructing a surface layer between two adjacent blocks, as there are other forms. For example, block 3 communicates with block 1 through a surface border layer, block 3 communicates with block 0 through an edge border layer, while block 3 communicates with block

4 through a vertex border layer, as shown in Figure 3. Usually a block needs to exchange information with its neighbors through six surface border layers, twelve edge border layers and eight vertex border layers. The width of border layers is selected to be the radius of the largest particle and works well.



(a) 3D view of initial configuration

(b) Top view of initial configuration

(c) Top view at time $t1$ during simulation

(d) Top view at time $t2$ during simulation

Fig. 5 Illustration of interblock communication.

A "patch" test is designed using 162 ellipsoidal particles. The particle assemblage is composed of two layers of 81 particles, gravitationally deposited into a rigid container, illustrated in Figure 5(a). The container is partitioned into four blocks separated by blue dashed lines shown in Figure 5(b), which also represents the initial configuration of the randomly-sized particles as shown from top view.

Each block is mapped to and computed by an individual process, so there are four processes, p0 to p3. Each process needs to communicate with other processes to determine its own boundary conditions. For example, process p3 needs to know those particles from process p1 that are enclosed by the pink rectangular box, those from process p2 enclosed by the red rectangular box, and those from process p0 enclosed by the green square box. A detailed movie records how those particles move across the borders and collide with particles from other

blocks, and it reveals that each process is able to determine its boundary conditions accurately. The overall motion of the 162 particles through parallel computing is observed to be the same as that observed in serial computing.

3.3 Flowchart of the parallel algorithm

The flowchart of the parallel code is depicted in Figure 6. It exhibits twelve flow processes or steps, among which one is sequential, two are partially parallel, and nine are fully parallel. Ten of the twelve processes are integrated in time using time increment loops until a simulation is completed, as shown by the bottom-up loop arrow.

In comparison to the serial code (Ellip3d), the parallel code (ParaEllip3d) ends up with six more steps as follows:

1. step 2: 2-Root process divides and broadcasts info. This step only runs once so it does not cost much CPU time.
2. step 3: 3-All processes communicate with neighbors. This interprocess communication is the most important step in the parallel algorithm.
3. step 9: 9-All processes update compute grids. This step arises from consideration of computational load balance.
4. step 10: 10-All processes merge and output info. This step serves the goal of snapshotting simulation states. Beware that it does not execute at each time increment, otherwise it could cause unacceptable cost.
5. step 11: 11-All processes release memory of receiving particle info. This step arises from MPI transmission mechanism and must be carefully taken care of.
6. step 12: 12-All processes migrate particles. This step handles the situation when particles move across block borders.

4 OPENMP IMPLEMENTATIONS

4.1 Serial code profiling

Profiling is performed on the serial code (Ellip3d) using a combination of two different methods: *Gprof tool* and *time function*. The time function is particularly adopted to distinguish the CPU time of neighbor estimate from that of contact resolution, because the parts of the codes for contact detection are designed to be inseparable for best performance. Two different algorithms of neighbor estimate are implemented: one is the n-by-n search with time complexity $O(n^2)$, and the other is link-cell (LC) method with time complexity $O(n)$, as described in Section 2. In LC method, the reference cell includes 26 surrounding neighbors in 3D space, and 8 surrounding neighbors in 2D space, as illustrated in Figure 3(a).

Figure 7 plots pie charts on time percentage of DEM components using 2,000 particles for different shapes and neighbor search algorithms. For spheres, the neighbor estimate accounts for 74.7% CPU time by $O(n^2)$ algorithm, and 20.8% by $O(n)$ algorithm; for ellipsoids, the neighbor estimate accounts for 13.2% CPU time by $O(n^2)$ algorithm, and it takes a smaller fraction, 2.9%, by $O(n)$ algorithm; for poly-ellipsoids, the neighbor estimate only accounts for 4.0% CPU time by $O(n^2)$ algorithm, and exhibits as low as 0.7% by $O(n)$ algorithm. Beware that the complex-shaped particles are much more computationally demanding than simple spheres in computing the contact geometry resolution. For instance, the contact resolution between a pair of true ellipsoids is approximately 50 times as expensive as that of a pair of spheres, and contact resolution between a pair of poly-ellipsoids is nearly 300 times as expensive as that of a pair of spheres in terms of floating-point operation, although they cost nearly the same in neighbor estimate.

The LC method is superior to the n-by-n search in estimating spatial neighbors. However, we must be cautious in evaluating the LC method:

1. It can only help improve the neighbor estimate, which normally accounts for a small fraction of the overall CPU time in 3D DEM of complex-shaped particles. It does not help improve the contact resolution or other

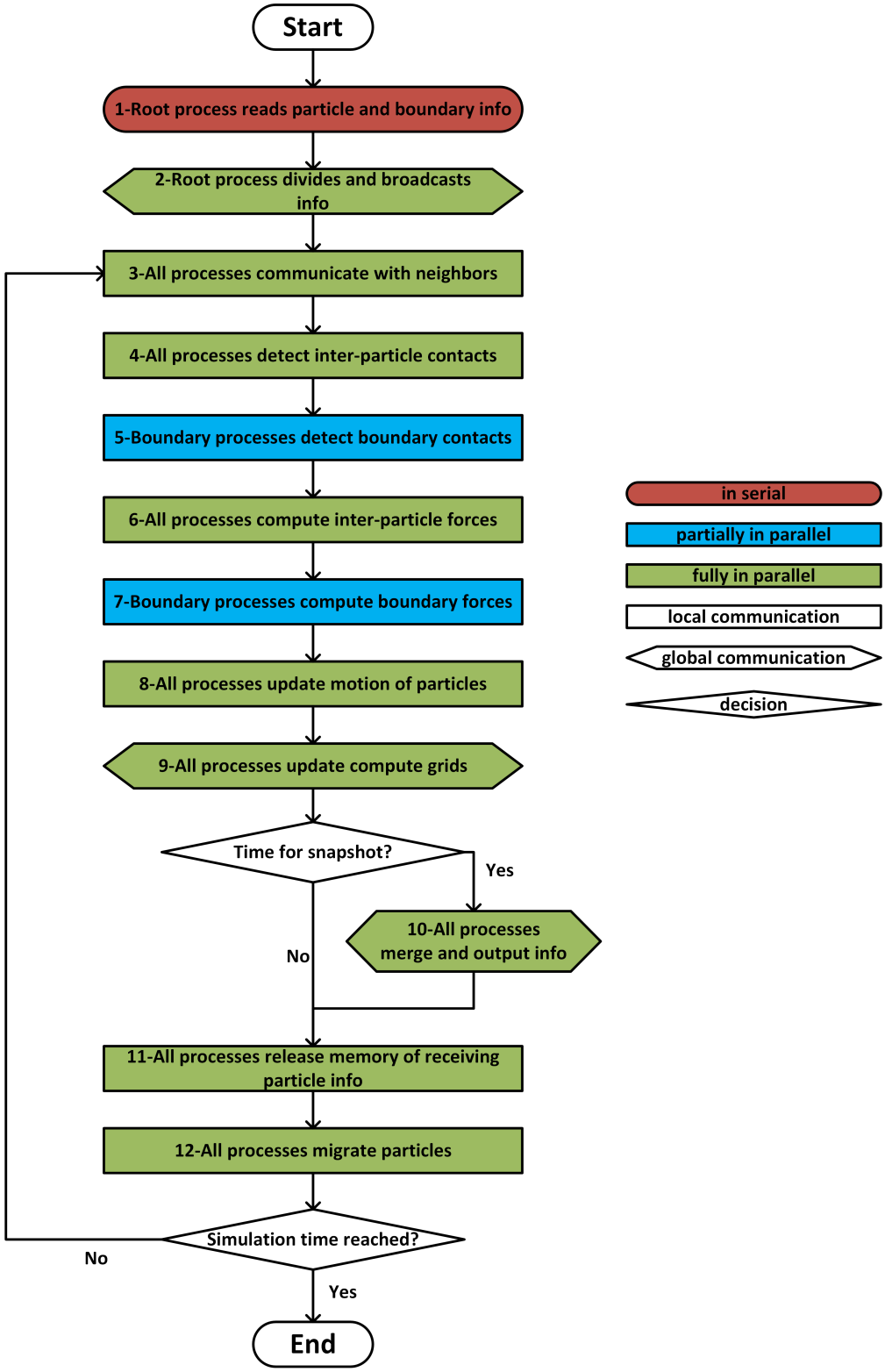
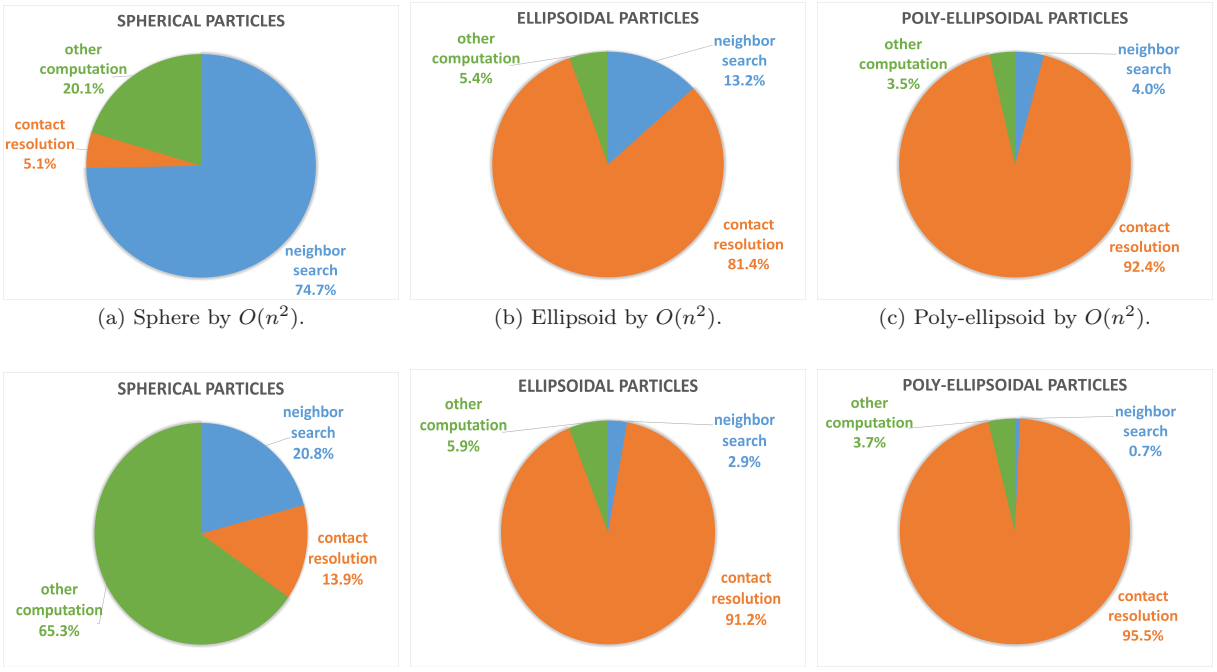


Fig. 6 Flowchart of the parallel algorithm of 3D DEM.



(a) Sphere by $O(n^2)$. (b) Ellipsoid by $O(n^2)$. (c) Poly-ellipsoid by $O(n^2)$.

(d) Sphere by $O(n)$. (e) Ellipsoid by $O(n)$. (f) Poly-ellipsoid by $O(n)$.

Fig. 7 Wall time percentage of DEM components by $O(n^2)$ and $O(n)$ algorithms using different particle shapes.

computation in DEM. As an example, the speedup of LC method over the n-by-n search is found to be as low as 1.15 on the simulation of 2,000 ellipsoidal particles, which implies no significant performance gain.

2. It cannot be parallelized using thread-based OpenMP. Referring to Figure 3(a), a serial LC method marks each cell that has been searched in sequence, and guarantees that the cell will not be searched again to generate duplicate contact pairs when cycling through all of the cells. For example, cell 1 checks with cells 9, 10 and 2, cell 2 only checks with cells 9, 10, 11, 3, and cell 10 only checks with cells 17, 18, 19 and 11, when the search occurs in sequence. However, this mark-and-avoid trick cannot be implemented due to iteration dependence and induced race condition between threads when each cell is assigned to an OpenMP thread in parallel, and consequently it cannot avoid generating a big number of duplicate and improper interparticle contact pairs (and costing twice the CPU usage); if a special data structure such as "unique hashed associative container" is employed aiming to eliminate the duplicate contact pairs, not only does the data structure involve race condition between threads (a workaround 'critical' section exhibits poor speedup), but also such a complicated data structure slows down the computation; furthermore, the complicated implementation of LC method is not well suited for high-performance OpenMP parallelization. Therefore, if OpenMP is adopted for parallelism, a n-by-n search has to be used accordingly.

4.2 Performance of various OpenMP schemes on contact detection

Note that the underlying data structures are carefully redesigned for OpenMP parallelization, and details such as data/loop reordering are implemented. To compare different OpenMP schemes, the GNU Compiler Collection, GCC-4.6.4, is used on a Dell T7500 Precision workstation with dual Intel Xeon X5690 CPUs which provides 12 total cores for multithreading. GCC-4.6.4 implements OpenMP version 3.0 and supports POSIX threads (pthreads). Note different OpenMP specifications can have consequences on performance.

4.2.1 Pragma "parallel"

Partitioning: A parallel code is first implemented by partitioning all of the n particles into ts partitions, where n denotes the number of particles and ts the number of threads, and assigning each partition to an OpenMP thread, as shown in Figure 8(a). Within each partition there are n/ts particles, and each particle in the partition needs to check against all its subsequent particles in the underlying data structure (array or vector), even though they are not within the same partition, which leads to load imbalance between threads. For example, if an array particle[10] denotes 10 particles and 3 threads are used for parallelization, then each thread/partition deals with 3 or 4 particles and particle[i] needs to check with particle[i+1], particle[i+2], ..., particle[9] to see if they are possibly in contact, and if they are, then find the contact points and penetration information by conducting contact resolution.

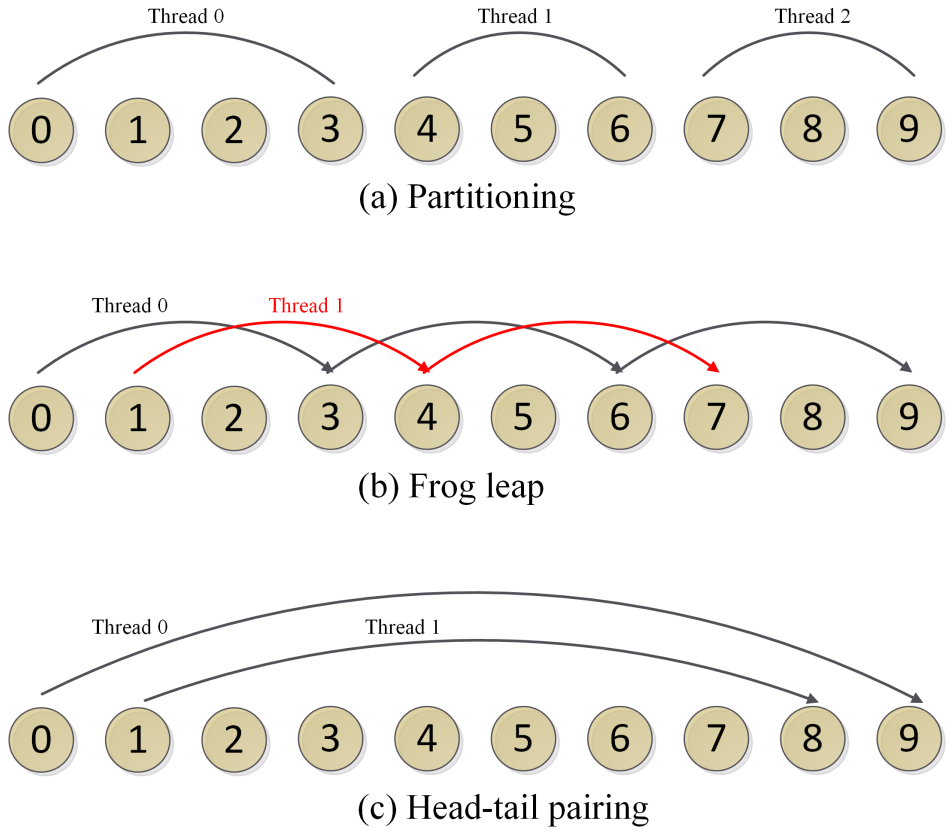


Fig. 8 Various OpenMP implementations using pragma "parallel": (a) *Partitioning*: assign a contiguous group of particles in the underlying data structure to a thread; (b) *Frog leap*: each thread does frog leap to locate and process particles; (c) *Head-tail pairing*: each thread processes a pair of particles located from the head and tail of underlying data structure.

Frog leap: This approach does not partition particles in the underlying data structure; instead, it lets each thread locate and process particles in a frog leap manner. For example, assume there are 10 particles and 3 threads, as illustrated in Figure 8(b), thread 0 works on particle[0], particle[3], particle[6] and particle[9]. The interval of frog leap is just the total number of threads ts. Similarly, thread 1 works on particle[1], particle[4] and particle[7]. Each thread does a frog leap until it approaches the end of the data structure. Apparently this approach makes a better load balance between threads because each thread processes particles as adjacent as possible.

Head-tail pairing: The third approach aims to achieve an even better load balance between threads by pairing two particles from the head and tail of the underlying data structure. As shown in Figure 8(c), thread 0 works on particle[0] and particle[9], thread 1 works on particle[1] and particle[8], etc. Each thread will do the same type of frog leap until it approaches particle[$n/2$], where n denotes the total number of particles. Now each thread performs the same number of steps in neighbor estimate, which results in a nearly perfect nominal load balance. However, actual load balance is not guaranteed because it depends on whether two particles are actually in contact. If they are, then contact resolution is performed in succession, which leads to more computation.

4.2.2 Pragma "parallel for"

As OpenMP provides iterative loops with various scheduling policies, it is interesting to see how they work and compare with the above implementations. The *schedule (static)* and *schedule (dynamic)* are tested.

All the implementations are tested on a Dell T7500 Precision workstation of dual Intel Xeon X5690 CPUs which provides 12 total cores for multithreading. A total of 12 threads are assigned in execution for maximum speedup on the workstation. The tests are performed on an assemblage of 2.5k particles at rest, shown by Figure 3(a). To better understand the parallelism of contact detection, which includes neighbor estimate and contact resolution, the code for contact detection is particularly isolated for performance evaluation, listed in Table 1.

Table 1 12-thread performance of contact detection using various OpenMP implementations on a dual hexa-core workstation.

method		speedup
serial	N-by-N	1.00
	link-cell	1.17
pragma "parallel"	partitioning	8.53
	frog leap	10.30
	head-tail pairing	10.38
pragma "parallel for"	schedule(static)	10.06
	schedule(dynamic)	11.15

Among those "parallel" pragma methods, the partitioning method achieves a speedup of 8.53 using 12 threads. It is the load imbalance between threads that gives rise to this lower speedup. It is anticipated that the frog leap method has a higher speedup, 10.3, due to a better load balancing scheme. The head-tail pairing method achieves a speedup of 10.38, a little higher than the frog leap method.

The "parallel for" pragma allows us to choose different scheduling policies. It is discovered that the dynamic schedules achieve better performance than the static schedules, which have a lower speedup than the frog leap and head-tail pairing methods. Static schedules have low overhead but may exhibit high load imbalance, and dynamic schedules have higher overhead but can reduce load imbalance (Michael, 2003). In our tests it seems the cost of load imbalance dominates the cost of overhead, and this is probably due to the fact that the computations consume much more CPU time than the hidden thread fork-join process.

The schedule (dynamic) dynamically allocates only one iteration at a time to the tasks, which has the finest granularity and achieves the highest speedup, 11.15, by running 12 threads.

Furthermore, it is observed that OpenMP achieves a nearly linear relationship between the speedup and number of threads in contact detection, as shown by Figure 9. This also confirms that the overhead of thread fork-joining process is sufficiently small compared with actual numerical computations.

Beware on the speedup number of 10∼11, which looks high in terms of 12 threads but actually only reflects the performance increase of contact detection. It does not account for the 6% fraction of the other computations (shown in Figure 7), which are not suitable for parallelism. Speedup of the overall program cannot be as high, referring to Section 5 where Amdahl's law is considered.

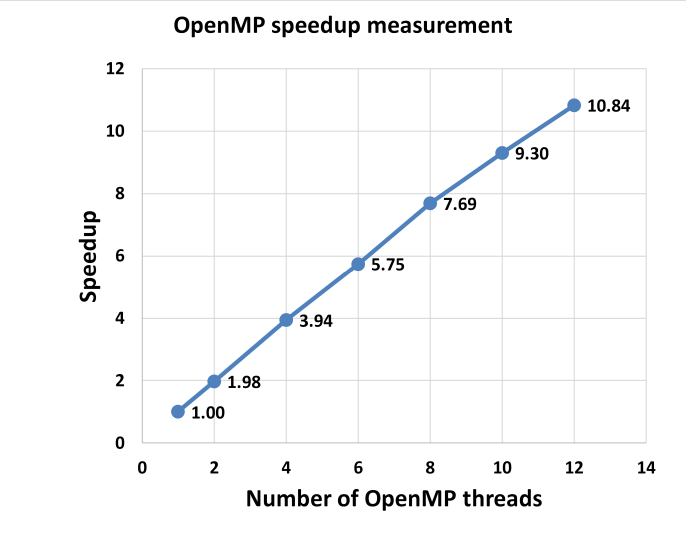


Fig. 9 Relationship between speedup and number of threads in OpenMP parallelism (using dynamic scheduling policy) of contact detection.

5 OPENMP PERFORMANCE

Starting from this section, the simulations are performed on multiple DoD HPC supercomputers: Spirit, Excalibur, Thunder and Topaz, and their parameters are listed in Table 2. The compute nodes of these supercomputers are non-uniform memory access (NUMA) architectures; for example, each compute node of Spirit has two Sandy Bridge-based Intel Xeon CPU E5-2670 2.60 GHz processors and 32 GB memory. Note various combinations of compilers and libraries are provided across these platforms, for example, Intel compilers 16.0.3 (OpenMP 4.0), SGI MPT 2.14 and Boost C++ 1.57 are selected on Spirit; PrgEnv-Intel/5.2.40 (OpenMPI 4.5), Cray-MPICH/7.1.0 and Boost C++ 1.65 are selected on Excalibur.

Table 2 Multiple DoD supercomputers

supercomputer	Spirit	Excalibur	Thunder	Topaz
system	SGI ICE X	Cray XC40	SGI ICE X	SGI ICE X
compute nodes	4,590	3,098	3,216	3,456
cores per node	16	32	36	36
total cores	73,440	101,184	125,888	125,440
memory per node	32 GB	128 GB	128 GB	128 GB
CPU	Xeon E5-2670	Xeon E5-2698v3	Xeon E5-2699v3	Xeon E5-2699v3
core speed	2.6 GHz	2.3 GHz	2.3 GHz	2.3 GHz
interconnect	4x FDR InfiniBand	Cray Aries	4x FDR InfiniBand	4x FDR InfiniBand
peak PFLOPS	1.50	3.77	5.62	4.66
MPI	SGI MPT	Cray MPICH2	SGI MPT	SGI MPT

Sand pluviation ("rainfall") is selected as our representative test in evaluating parallel performance: the sand particles are generated based on a specific soil gradation curve (so that they have different sizes) and "floated" in space initially without interaction; during the process of gravitational pluviation, the bottom particles start to pack up and interparticle contacts should be detected; at the end, all particles come to rest and stay in a relatively "dense" state statically under gravity. The overall pluviation process, and the static packed state, are both used in the performance evaluation of parallel computing.

Speedup is defined as the ratio between sequential execution time and parallel execution time, as shown in Eq. (1), and *efficiency*, a measure of processor utilization, is defined as speedup divided by the number of

processors used, according to Eq. (2).

$$\text{speedup } \psi(n,p) \equiv \frac{\text{sequential execution time}}{\text{parallel execution time}} \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \tag{1}$$

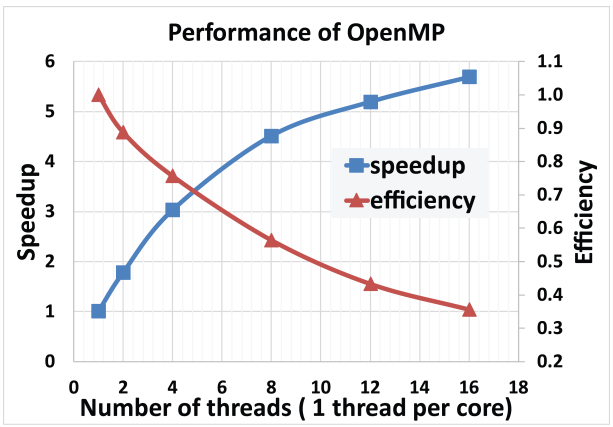
$$\text{efficiency } \varepsilon(n,p) \equiv \frac{\psi(n,p)}{p} \tag{2}$$

where n is the problem size (number of particles), p is the number of processors, $\sigma(n)$ is the inherently serial portion of computation, $\varphi(n)$ is the parallelizable portion of computation, and κ is the overhead of parallelization (communication operations plus redundant computation).

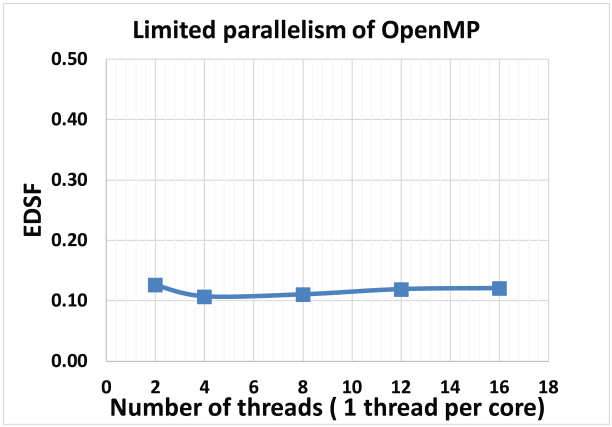
Amdahl's law (Michael, 2003) indicates that the maximum speedup ψ achievable by a parallel computer with p processors is,

$$\psi \leq \frac{1}{f+(1-f)/p}, \tag{3}$$

where $f = \sigma(n)/(\sigma(n)+\varphi(n))$ denotes the fraction of operations in a computation that must be performed sequentially.



(a) Speedup and efficiency.



(b) EDSF versus number of processors.

Fig. 10 OpenMP performance.

Figure 10(a) plots the overall speedup and efficiency of an OpenMP implementation versus number of processors (cores) on a node (two octa-core, i.e., 16 cores) of Spirit in a numerical test of static packed 2.5k ellipsoidal particles. It can be seen that as the number of processors increases from 1 to 16, the speedup increases from 1 to 5.38 and the efficiency decreases from 1 to 0.36. The relationship demonstrates nonlinearity clearly as speedup increase becomes slower as the number of processors increases. Compared to the linear relationship shown in Figure 9 in Section 4.2, which solely studies the performance of contact detection, it can be anticipated that the inherently sequential part, namely, the 6% fraction associated with other computations illustrated in Figure 7, plays an important role in placing an upper bound on the performance gain.

Let $f = 0.06$ and $p = 16$ in Eq. (3), the maximum speedup reads $\psi = 8.4$, which is greater than the measured value 5.38. This is normal because Amdahl's law cannot take into account the parallel overhead, namely, the thread fork-joining overhead in this case.

The experimentally determined serial fraction (EDSF) is plotted in Figure 10(b). Several points can be made from the plot:

1. The 11% EDSF is greater than the 6% profiled inherently sequential computation. Clearly this discrepancy results from parallel overhead of OpenMP. It might be a good estimate that the OpenMP overhead of 3D DEM is nearly 5% as a result of 11% minus 6%. Without EDSF measurement it is difficult to estimate OpenMP overhead because OpenMP is directive-based and its runtime behavior is hidden to programmers.

2. It is observed from Figure 10(b) that the EDSF is nearly constant when the number of processors increases. That indicates limited opportunities of parallelism for OpenMP in additional to the OpenMP overhead, that is, the fraction of inherently sequential computation is not negligible in the OpenMP implementation.
3. The inherently sequential computation in OpenMP parallelization could be overcome by applying global MPI parallelism, which decomposes the spatial domain and parallelizes nearly all of the computations.

6 SINGLE-NODE PERFORMANCE: MPI VS OPENMP

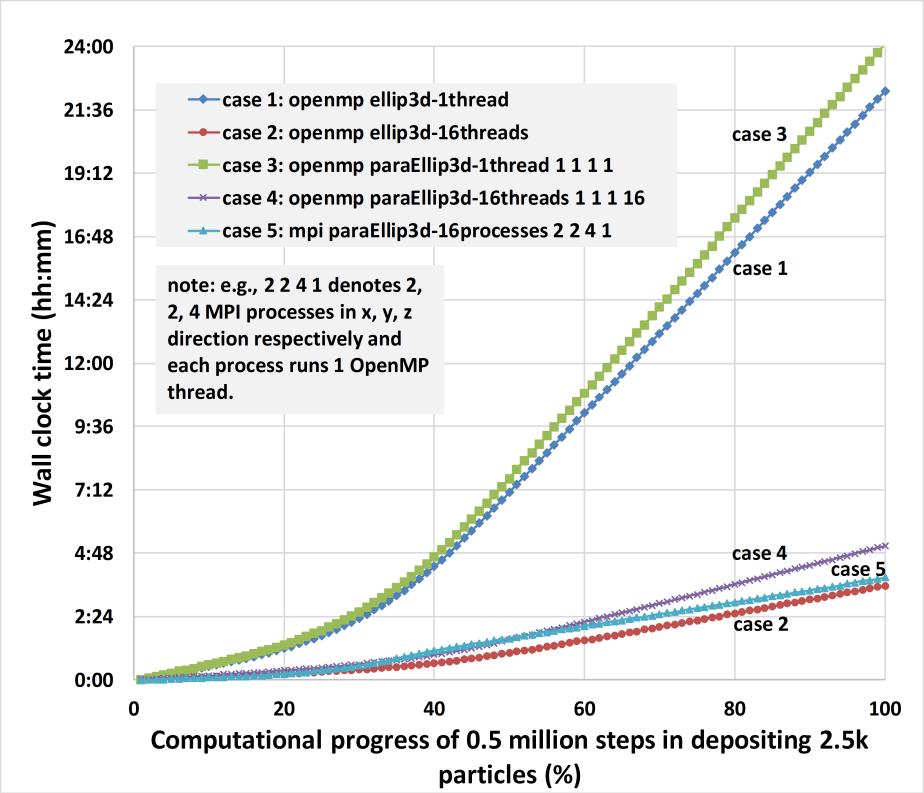


Fig. 11 MPI-OpenMP performance on a single node modeling 2.5k particles.

The sand pluviation process of 2.5k ellipsoidal particles is modeled on one node (two octa-core, i.e., 16 cores) of Spirit. Five combinations of MPI and OpenMP are tested in order to investigate the performance difference. For the best comparison a constant time interval (1.0e-6 seconds) is used in this dynamic simulation to run 0.5 million steps. Figure 11 plots the wall clock time versus percentage of the steps of the 5 combinations:

case 1: run OpenMP on the serial code (Ellip3d) using 1 thread.
case 2: run OpenMP on the serial code (Ellip3d) using 16 threads.
case 3: run OpenMP on the parallel code (ParaEllip3d) using 1 thread.
case 4: run OpenMP on the parallel code (ParaEllip3d) using 16 threads.
case 5: run MPI on the parallel code (ParaEllip3d) using 16 processes ($2 \times 2 \times 4$ in x, y, z direction respectively).

First of all, it is clear that the increment of one step in the computational progress is faster at an earlier stage but slower at a later stage in all cases. This is due to the fact that interparticle contacts start from zero and increase gradually until they are fully packed.

Secondly, comparing case 1 to case 3, and case 2 to case 4 reveals that the parallel version (ParaEllip3d) is slower than the serial version (Ellip3d) in OpenMP mode. This arises from the redundant code and computation "offset" introduced by MPI parallelization, since there is no MPI parallel communication overhead involved in cases $1 \sim 4$. Computed based on the final state data of the 2,500 particle pluviation in case 2 and case 4 that use 16 OpenMP threads, the parallel version is approximately 30% slower than the serial version; however, the 16-process MPI version is only 8% slower than the serial version. The redundant code and computation introduced by MPI parallelization could consume an important fraction of the parallel overhead.

Thirdly, using the same parallel code but running OpenMP and MPI independently gives an interesting comparison between case 4 and case 5: MPI with 16 processes is even faster than OpenMP with 16 threads on a single compute node, and the speedup is approximately 1.3. There are several reasons for this phenomenon:

1. MPI actually uses shared memory to exchange messages in intranode mode, so its implementation may be as efficient as OpenMP implementation. A simple OpenMP parallelization of the code may not outperform the MPI.
2. As pointed out in Section 5, OpenMP does not parallelize all the code as there is 6% inherently sequential code, whereas MPI is able to parallelize all the code in the sense it decomposes the computational domain such that each of the "6%" sequential code in its own partition is running simultaneously with the others, thus increasing the parallelism.
3. Most likely, it could be that the $2 \times 2 \times 4$ MPI processes lead to decrease of computational granularity and increase of locality, namely, each MPI process handles around 156 particles, while the sole process in OpenMP handles 2.5k particles using a dynamic scheduling policy. This comparison is quantitatively investigated in Section 7.

7 MULTINODE PERFORMANCE: PURE MPI VS HYBRID MPI-OPENMP

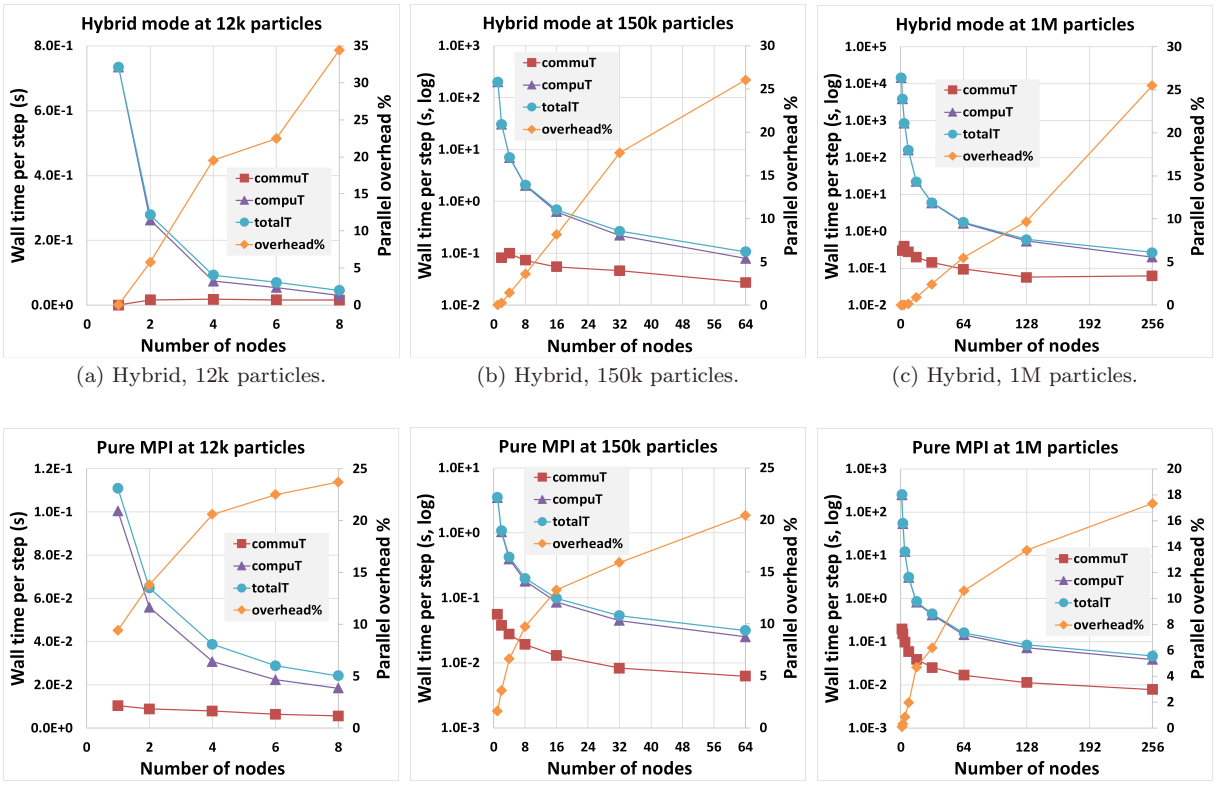
As stated in Section 3, the MPI design of 3D DEM allows a flexible combination of MPI and OpenMP adapting to supercomputer architectures. In this section we analyze a hybrid MPI-OpenMP parallel mode by mapping every link-block to a compute node and letting each node execute multiple OpenMP threads, and compare its performance with that of pure MPI mode.

As stated in Section 3.1, various choices exist on how to map domain data into processors, for example, mapping a block of particles to one core, multiple cores within one CPU, the whole CPU, multiCPUs within a node, or even a whole node. Herein, mapping a block of particles to one core is the so-called "pure MPI" mode, while mapping a block of particles to one whole node is the so-called "hybrid MPI-OpenMP". There are in-between mapping granularities which should exhibit intermediate performance.

In this section DEM simulations are performed on the rested particle assemblies that are obtained from gravitational deposition. Figure 12 plots module execution time in hybrid MPI-OpenMP mode and pure MPI mode for static simulations across three scales: 12k, 150k and 1 million particles. The execution time of different modules in the parallel code is recorded using MPI time functions. The modules are described as follows:

- commuT: communication time in step 3 (3-All processes communicate with neighbors) of the flowchart in Figure 6.
- migraT: migration time in step 12 (12-All processes migrate particles) of the flowchart.
- compuT: numerical computation time, it equals totalT-commuT-migraT.
- totalT: total time at each step.
- overhead%: (commuT+migraT)/totalT, i.e., the overall parallel overhead percentage.

Note that the IO cost in 3D DEM is very limited. As a typical example, only 100 snapshots are taken for 5 million time increments of 3D DEM simulation. The synchronization overhead is less than 0.1% of the commuT across all of the simulation scales such that it is a negligible fraction of overall parallel overhead, of which the communication cost dominates.



(a) Hybrid, 12k particles. (b) Hybrid, 150k particles. (c) Hybrid, 1M particles.

(d) Pure MPI, 12k particles. (e) Pure MPI, 150k particles. (f) Pure MPI, 1M particles.

Fig. 12 Speedup and efficiency comparison between pure MPI and hybrid MPI-OpenMP.

7.1 Performance comparison

Firstly, as the number of compute nodes increases, the computation time and communication time (thus the total time) decrease, and the decrease rates are high at the very beginning and slow down later for a fixed problem size. The trend holds true for both pure MPI and hybrid MPI-OpenMP modes. This is the goal and achievement that we should anticipate from parallel computing.

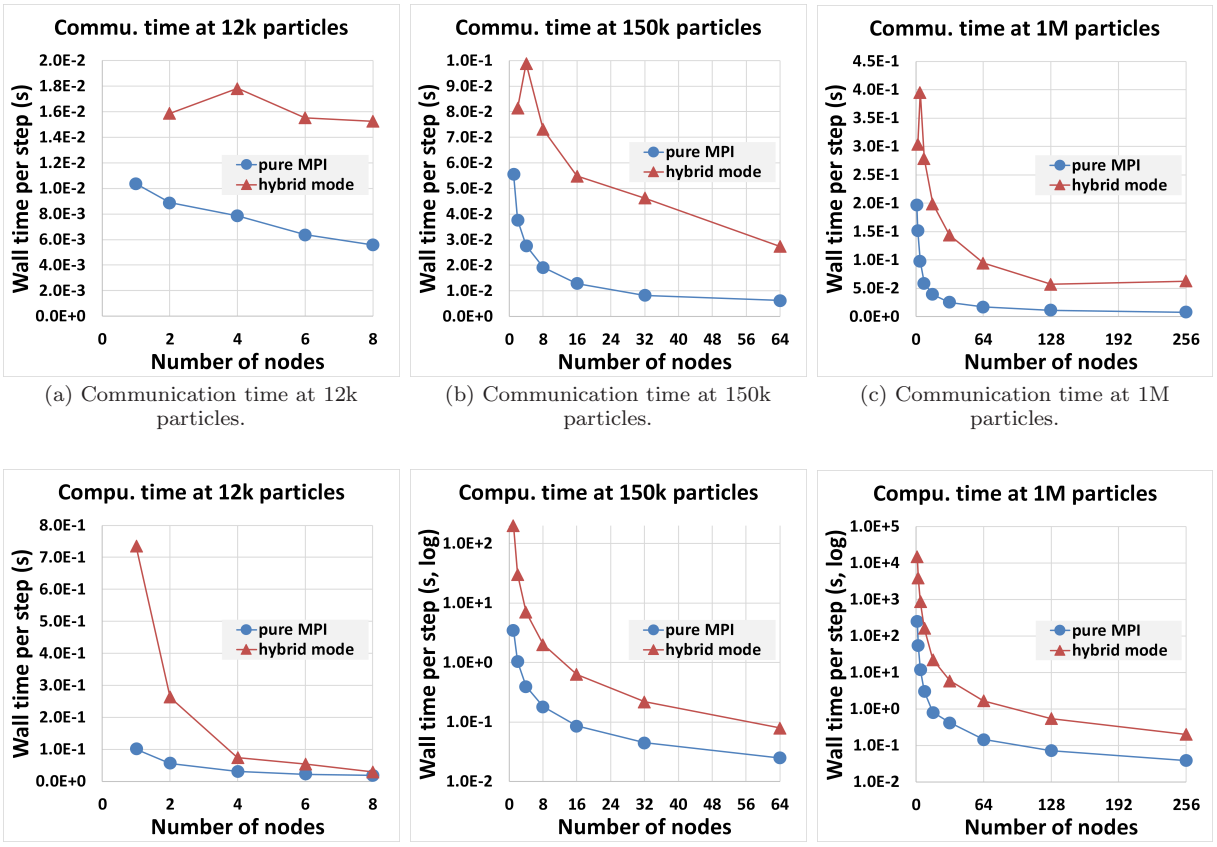
Secondly, the parallel overhead consumes a relatively low fraction of the wall time, approximately 15% by averaging across various number of compute nodes. The hybrid MPI-OpenMP reveals a slightly higher parallel overhead percentage than pure MPI.

However, the execution time of hybrid MPI-OpenMP is much larger than that of pure MPI.

7.2 Module execution time

Figure 13 compares the communication time and computation time between pure MPI and hybrid MPI-OpenMP modes across three scales: 12k, 150k and 1M particles. It is clear that both the communication time and computation time of hybrid MPI-OpenMP are much higher than that of pure MPI, and it is particularly pronounced on larger scales of 150k and 1M particles. For example, for 1M particles computed by 64 nodes, the communication time is 9.4E-2 seconds per step in hybrid mode and 1.7E-2 seconds per step in pure MPI (Figure 13(c)), and the computation time is 1.7E+0 seconds per step in hybrid mode and 1.4E-1 seconds per step in pure MPI (Figure 13(f), logarithmic scale in time).

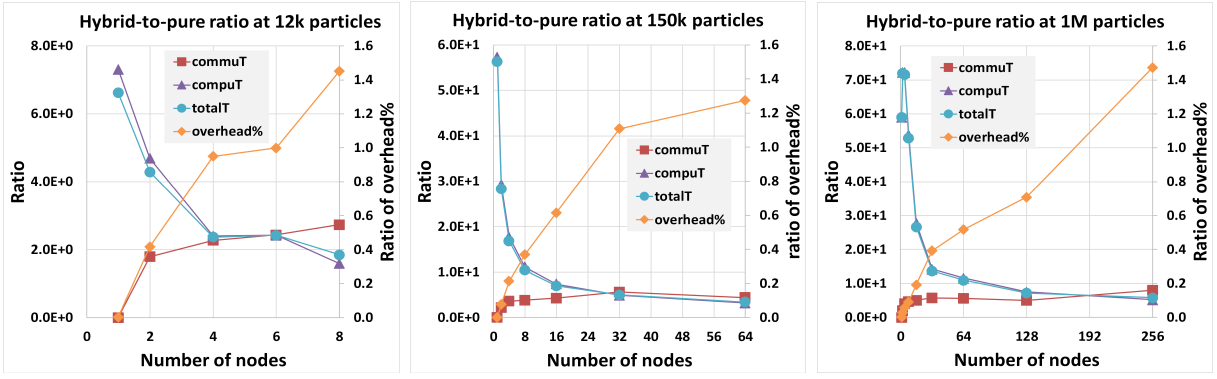
The ratios between hybrid MPI-OpenMP and pure MPI, including communication time, computation time, total time and overhead percentage, are plotted in Figure 14. Overall, the computation time ratio decreases and the communication time ratio increases with an increasing number of compute nodes (thus a decreasing



(a) Communication time at 12k particles. (b) Communication time at 150k particles. (c) Communication time at 1M particles.

(d) Computation time at 12k particles. (e) Computation time at 150k particles. (f) Computation time at 1M particles.

Fig. 13 Communication and computation time.



(a) 12k particles. (b) 150k particles. (c) 1M particles.

Fig. 14 Ratio between hybrid mode and pure MPI.

computational granularity). As an example, the computation time ratio decreases from 72.2 to 5.2, whereas the communication time ratio increases from 4.1 to 8.0, when the number of compute nodes increases from 4 to 256 in the case of 1 million particles. It is also observed that the computation time ratio between hybrid MPI-OpenMP and pure MPI decreases quickly with an increasing number of compute nodes, and the communication time ratio increases slowly. The ratio of parallel overhead percentage between hybrid MPI-OpenMP and pure MPI turns out to be an increasing function of number of compute nodes.

7.3 Profiling at optimal computational granularity

Yan and Regueiro (2018b) pointed out: for a fixed problem size (or simulation scale), the parallel overhead (mostly the interprocess communication) percentage increases with a decreasing computational granularity (CPG), due to the fact that computation time decreases faster than the communication time whenthe CPG decreases in 3D DEM of complex-shaped particles, and the best performance is achieved at a certain CPG in parallel computing.

Table 3 gives the module execution time at optimal computational granularity across 12k, 150k and 1M particles. The optimal computational granularity is obtained by testing various number (from 1 to 2,048 if applicable) of compute nodes and selecting the one which achieves shortest wall clock time: 8 nodes for 12k particles, 64 nodes for 150k particles, and 256 nodes for 1 million particles.

It should be emphasized that the the parallel optimal CPG of 150 $\sim$ 300 particles per core (PPC) is measured for complex-shaped particles such as ellipsoids or poly-ellipsoids; for spheres it is much larger due to significantly lower computational cost, e.g., $15,000 \sim 30,000$ PPC.

Table 3 Profiling on pure MPI and hybrid MPI-OpenMP.

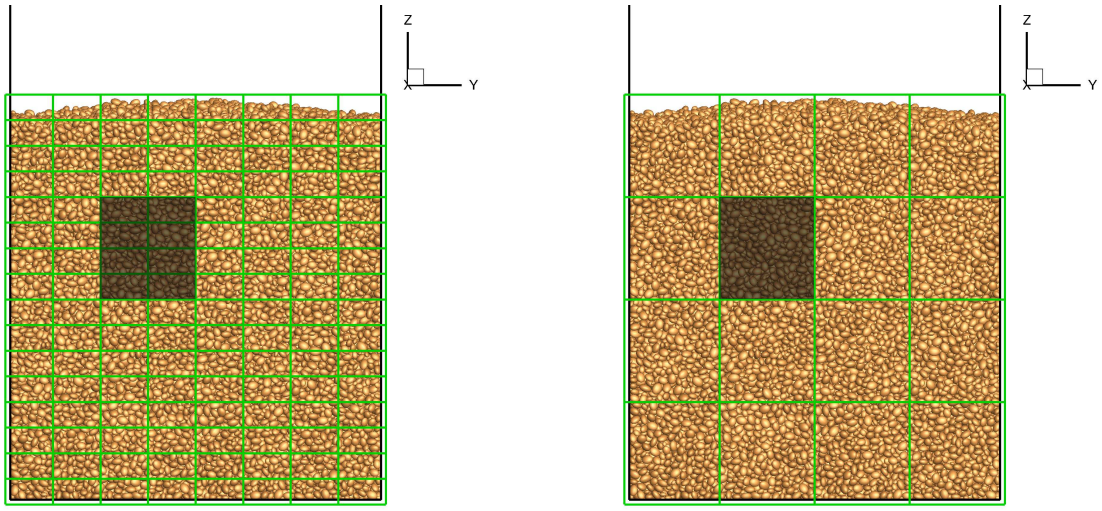
	time (s)	commuT	migraT	compuT	totalT	overhead%
12k, 8 nodes	pure MPI	5.58E-03	1.47E-04	1.85E-02	2.42E-02	23.7
	hybrid mode	1.53E-02	1.19E-04	2.93E-02	4.47E-02	34.4
	ratio	2.7	0.8	1.6	1.8	1.5
150k, 64 nodes	pure MPI	6.21E-03	2.11E-04	2.50E-02	3.15E-02	20.4
	hybrid mode	2.73E-02	4.56E-04	7.90E-02	1.07E-01	26.1
	ratio	4.4	2.2	3.2	3.4	1.3
1M, 256 nodes	pure MPI	7.82E-03	2.23E-04	3.84E-02	4.64E-02	17.3
	hybrid mode	6.25E-02	9.54E-04	2.00E-01	2.69E-01	25.5
	ratio	8.0	4.3	5.2	5.8	1.5

The tables clearly exhibit performance difference between hybrid MPI-OpenMP and pure MPI. For example, in the case of 1 million particles computed by 256 nodes, the hybrid MPI-OpenMP spends 8.0$\times$ communication time, 5.2$\times$ computational time and 5.8$\times$ total time as pure MPI. The hybrid MPI-OpenMP tends to have a pronounced performance gap between pure MPI.

7.4 Influence of computational granularity

Figure 15 illustrates the CPU mapping scheme and computational granularity for pure MPI and hybrid MPI-OpenMP, respectively, in the case of 150k particles. Computed by 64 compute nodes (each node containing 16 cores), pure MPI allows 1,024 (64x16) processes mapped to 8 x 8 x 16 blocks of particles in 3D space, while hybrid MPI-OpenMP uses only 64 processes (each process running 16 OpenMP threads) that are mapped to 4 x 4 x 4 blocks. As a result, pure MPI contains approximately 150 particles per block or per process, and the hybrid MPI-OpenMP contains approximately 2,300 particles per block or per process. It is noted again that the parallel optimal CPG of 150 $\sim$ 300 particles per core (PPC) is measured for complex-shaped particles that impose high CPU usage for contact geometry resolution; for spheres the number can be quite different, e.g., $15,000 \sim 30,000$ PPC.

If the shaded block in hybrid mode (marked in Figure 15(b)) is compared to the same volume in pure MPI mode (marked in Figure 15(a)), it is found that 16 threads (within 1 process) in hybrid mode, or 16 (2 x 2 x 4) processes in pure MPI mode, are computing over the same spatial volume of particles. However, the 16 processes in pure MPI not only have smaller computational granularity, which results in higher locality and computational efficiency, but also have smaller interprocess communication granularity (CMG), which leads to lower MPI transmission volume, than the 16 threads (1 process) in hybrid MPI-OpenMP. This is justified by the data in Table 3 that MPI consumes much less computational time and communication time in the code



(a) pure MPI. (b) hybrid MPI-OpenMP.

Fig. 15 Mapping scheme and computational granularity for 150k particles using 64 compute nodes.

profiling: hybrid MPI-OpenMP spends 4.4× communication time, 3.2× computational time and 3.4× total time as pure MPI. Beware that these data are only applicable to complex-shaped particles such as ellipsoid or poly-ellipsoid due to their much more expensive CPU demand for contact geometric resolution; they are not valid for spheres (not our focus).

The computational granularity difference between pure MPI and hybrid MPI-OpenMP is pronounced: 150 particles per process for the former, and 2,300 particles per process for the latter. Even though it utilizes 16 threads to compute the 2,300 particles, a hybrid MPI-OpenMP process exhibits much lower performance. This is mainly due to the fact that the LC method with time complexity $O(n)$ is not parallelizable with OpenMP (see Section 4.1), and OpenMP has to rely on parallelizable n-by-n search. It gives rise to the computation time ratio, 3.2, between hybrid MPI-OpenMP and pure MPI.

Yan and Regueiro (2018a) provides a comprehensive comparison of the $O(n^2)$ vs $O(n)$ effect on computational performance. It is worth noting that the neighbor search algorithms, $O(n^2)$ and $O(n)$, only affect the performance of neighbor search. They have no bearing on contact resolution. The overall performance improvement resulting from these algorithms is highly limited for complex-shaped particles, because neighbor search only takes up a small fraction of floating-point operations in the whole computation.

They define the neighbor search fraction (NSF) as

$$NSF = \frac{T_{\text{neighbor estimate}}}{T_{\text{contact detection time}}} = \frac{T_{\text{neighbor estimate}}}{T_{\text{neighbor estimate}} + T_{\text{contact resolution}}},$$

where T denotes the wall clock time. They point out that in serial computing, the more complex the particle shapes (from sphere to ellipsoid to poly-ellipsoid), the smaller the neighbor search fraction (NSF); and the lower the CPG, the smaller the NSF. The $O(n)$ search algorithm is superior to $O(n^2)$ on the whole, especially for situations with coarser CPG; however, the $O(n)$ algorithm is inferior to $O(n^2)$ for CPG finer than the parallel optimal CPG. In parallel computing of complex-shaped particles, $O(n^2)$ algorithm is slower than $O(n)$ at very coarse CPGs; however, it becomes faster than $O(n)$ at fine CPGs that are mostly employed in practical computations to achieve best performance. This means that $O(n^2)$ algorithm outperforms $O(n)$ algorithm in large-scale parallel 3D DEM simulations generally.

7.5 Communication granularity (CMG) of 3D DEM

As shown in Figure 13(a) (b) and (c), the communication time decreases with an increasing number of compute nodes for a fixed problem size, and it increases as the problem size increases. The communication time is in response to the term $\kappa(n, p)$ in Eq. (2). Beware $\kappa(n, p)$ denotes the overall time required for parallel overhead, in which the communication time dominates.

When an ellipsoidal particle in 3D DEM is transmitted through the border/ghost layer or migration layer of a link-block using MPI, all of the information on the particle needs to be transmitted: particle ID, particle type, density, Young's modulus, Poisson's ratio, current and previous position, current and previous orientation, current and previous translational and rotational velocities, current and previous force and moment (note a history-dependent Mindlin's shear model is adopted). This amounts to 624 bytes in the C++ class for an ellipsoidal particle. In comparison, a computational cell in a 3D CFD (Computational Fluid Dynamics) solver for the Euler equations only needs to transmit 5 variables (density, three velocities, and energy) which amounts to 40 bytes in a C++ class.

In particular, there exists a number of particles in the ghost layers that need to be transmitted via MPI. For example, approximately 170 particles need to be transmitted per process in pure MPI illustrated by Figure 15(a); and approximately 1,200 particles need to be transmitted in hybrid MPI-OpenMP illustrated by Figure 15(b). These amount to MPI transmission of 0.1 MBytes per process (104 MB totally) for pure MPI, and 0.7 MBytes per process (46 MB totally) for hybrid MPI-OpenMP, at each time step in the DEM simulation. Since a typical DEM simulation takes millions of time steps, the overall MPI transmission in 3D DEM places a heavy burden for interprocess communications.

Using the example of 150k particles computed by 64 nodes, pure MPI runs 1,024 processes each simultaneously transmitting 0.1 MB data per time step, whereas hybrid MPI-OpenMP runs 64 processes each simultaneously transmitting 0.7 MB data per time step. This results in the overall communication time ratio, 4.4, between hybrid MPI-OpenMP and pure MPI.

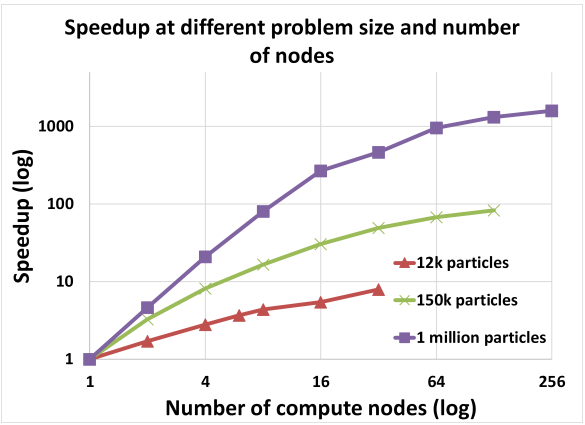
Due to the difference of computational granularity and accordingly communication granularity (CMG), pure MPI has many more partitions of particles and thus many more MPI point-to-point communications between these partitions than hybrid MPI-OpenMP; however, thanks to the fact that its communication size (i.e., MPI transmission volume) per process is much lower than that of hybrid MPI-OpenMP, and the local point-to-point non-blocking communications between neighboring partitions occur simultaneously, pure MPI turns out to have a much lower communication overhead than hybrid MPI-OpenMP.

In summary, pure MPI achieves both lower computational granularity (thus higher spatial locality) and lower communication granularity (thus faster MPI transmission) than hybrid MPI-OpenMP in 3D DEM, thus it works more efficiently than hybrid MPI-OpenMP.

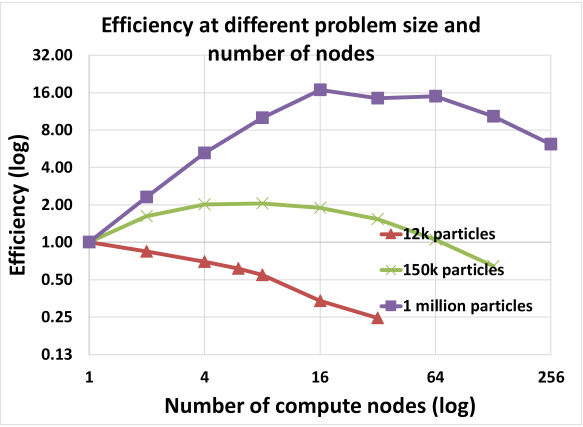
7.6 Influence of memory/cache hierarchy

Figure 16 compiles the speedup and efficiency data from static simulations of 150k, 1 million and 10 million particles for pure MPI and hybrid MPI-OpenMP modes, respectively. A loglog graph is plotted due to the wide range of problem size and number of processors. Firstly it is observed that speedup is an increasing function of the problem size for any fixed number of processors; secondly, strong superlinear speedup has been discovered: in pure MPI mode, the efficiencies of 150k and 1 million particle simulations go beyond 1 (100%); in hybrid MPI-OpenMP modes, all of the 12k, 150k and 1 million particle simulations go beyond 1 (100%). Overall, the larger the scale, the stronger the superlinear speedup; thirdly, the efficiencies are not monotonically increasing: increase first but decrease later.

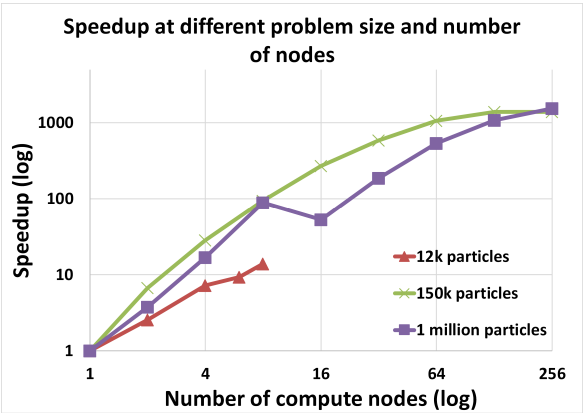
Yan and Regueiro (2018a,c) investigate the superlinear speedup in complex-shaped 3D DEM and conclude that: (1) Strong superlinear speedup has been discovered in large scale simulations of parallel 3D DEM for complex-shaped particles, which features "high-CPU-low-memory" demand and CPU-bound rather than memory-bound, and the larger the scale, the stronger is the superlinear speedup. The phenomenon is reproduced on multiple DoD supercomputers. The strong scaling and weak scaling measurements show that cache miss rate is sensitive to the memory consumption shrinkage per processor, and the last level cache (LLC) contributes most significantly to the strong superlinear speedup among all of the three cache levels of modern



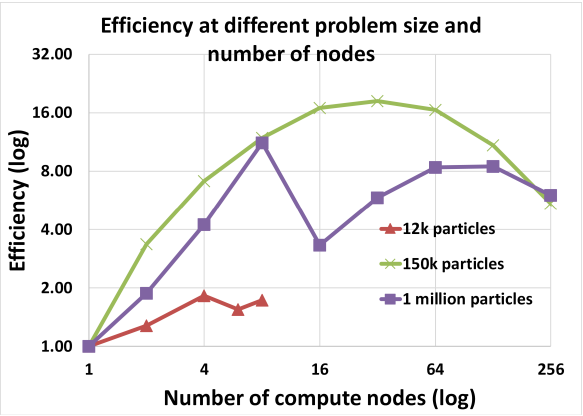
(a) Pure MPI speedup.



(b) Pure MPI efficiency.



(c) Hybrid MPI-OpenMP speedup.



(d) Hybrid MPI-OpenMP efficiency.

Fig. 16 Speedup and efficiency across orders of magnitude simulation scale.

microprocessors. The measurements are mostly attributed to the inherently perfect scalability of 3D DEM because its memory scalability function is a nonlinearly decreasing function of the number of processors.

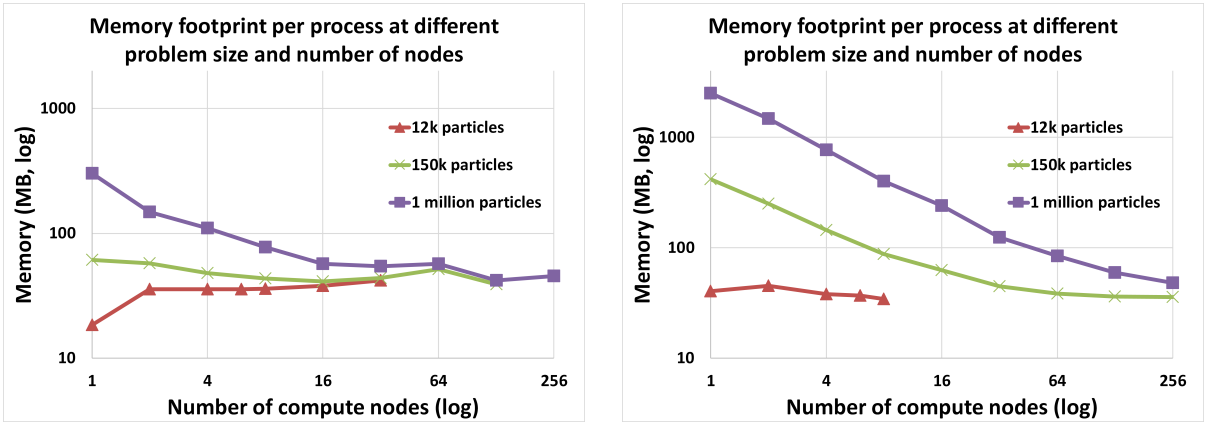
(2) It is worth pointing out that superlinear speedup exists in the low-memory-demand 3D DEM, aside from those high-memory-demand scientific computations that have been discovered. The superlinear speedup is commonplace for large-scale complex-shaped 3D DEM.

(3) Both $O(n^2)$ and $O(n)$ algorithms exhibit a strong superlinear speedup on large-scale simulations of complex-shaped 3D DEM. The $O(n)$ algorithm always exhibits a lower speedup than the $O(n^2)$ algorithm across all computational scales and granularities, mostly due to the base point measurement at a single compute node, whereby the $O(n)$ algorithm executes much faster than the $O(n^2)$ algorithm. On average, the speedup in $O(n)$ algorithm is reduced by approximately 1/3 relative to $O(n^2)$ algorithm on the simulation scale of 1 million ellipsoidal particles.

To measure the memory consumption, the system function `getrusage` is called in the code to acquire the resident memory per process. Each process prints out its own memory footprint and an average value is then calculated.

Figure 17 plots the memory consumption per process across various orders of magnitude of simulation scale for pure MPI and hybrid MPI-OpenMP modes, respectively. Overall, the memory footprints per process decrease with increasing number of compute nodes; and it is clear that hybrid MPI-OpenMP mode requires more memory usage than pure MPI mode.

At the optimal computational granularity (CPG), the memory consumption is below 150 MB per process/core. It is low in the sense of scientific computing. However, as the L3 cache per core is 2.5 MB on Sandy



(a) Pure MPI (log-log). (b) Hybrid MPI-OpenMP (log-log).

Fig. 17 Memory consumption per process/core on DoD supercomputers.

bridge-based CPU, the memory consumption per core is still one to two orders of magnitude larger than the L3 cache size.

It is particularly important to observe that: in pure MPI mode, for 12k particles the memory consumption per process does not decrease, and it corresponds to no superlinear speedup; whereas for 150k and 1M particles the memory footprint per process does decrease, and it corresponds to the superlinear speedup.

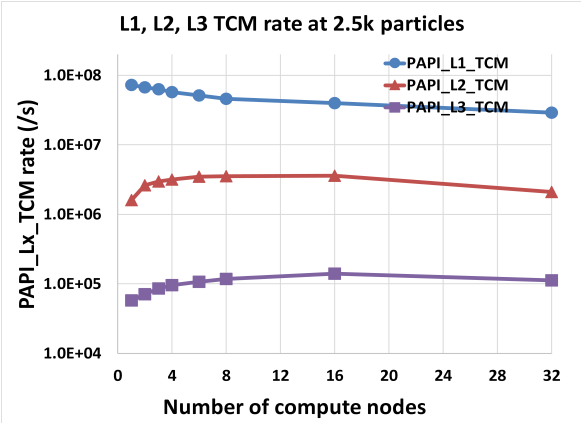
In hybrid MPI-OpenMP mode, all of the simulation scales demonstrate decreasing memory consumption with increasing number of compute nodes, which corresponds to the superlinear speedup shown in Figure 16 (d). The change of memory footprint per process plays a critical role in determining speedup characteristics.

Figure 18 plots the total cache miss (TCM) per second of L1, L2 and L3 caches across four orders: 2.5k, 12k, 150k and 1M particles on Thunder. Overall the TCM per second gives a clearly decreasing trend. In the cases of 2.5k and 12k particles, the TCM rates of L1, L2 and L3 caches decrease by nearly the same order of magnitude. However, in the simulations of 150k and 1M particles, the three cache level TCM rates decrease differently: L3_TCM rate $\gg$ L2_TCM rate $\gg$ L1_TCM rate. For example, L3_TCM rate decreases from 1.8E+7 to 6.5E+4 (nearly three orders of magnitude), L2_TCM rate decreases from 8.1E+7 to 1.4E+6 (one order of magnitude), and L1_TCM rate decreases from 7.1E+6 to 3.0E+6 (by 50%) for 1 million particles. It is clear that the L3 cache contributes most significantly to the strong superlinear speedup among all of the three cache levels.

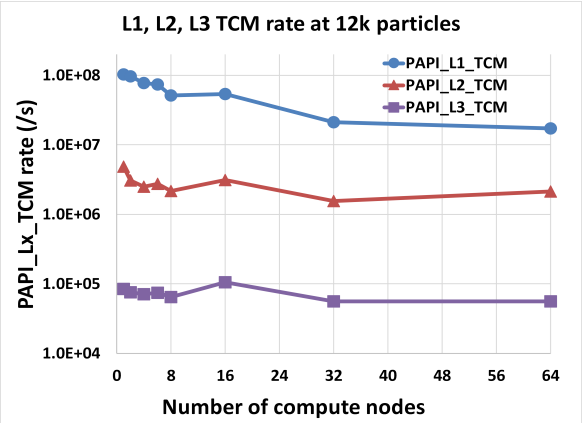
8 VARIATION OF HYBRID MPI-OPENMP MAPPING SCHEME

As pointed out in section 7, there are various mapping granularities, for example, mapping a block of particles to one core, multiple cores within one CPU, the whole CPU, multiCPUs within a node, or even a whole node. Furthermore, the size of a block (i.e., the number of particles within the block) can vary in terms of process/thread combination even if the computational resources are fixed. The mapping granularity and process/thread combination could result in different performance correspondingly. This section investigates the effect of different mapping schemes/granularities, along with the influence of NUMA architectures, in parallel computing.

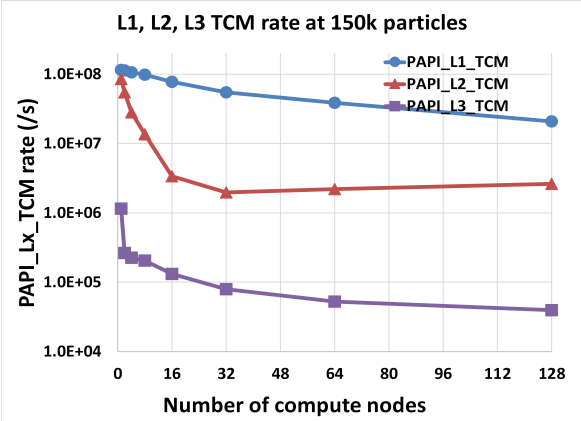
Note that all of the four DoD supercomputers listed in Table 2 have NUMA architecture, each node consisting of two Intel Xeon CPUs. Our tests are carried out on Thunder, an SGI ICE X system for which each compute node contains two Intel Xeon E5-2699v3 CPUs, namely, 36 cores and 128 GB memory per node. The 36-core architecture allows us to evaluate a wide range of hybrid MPI-OpenMP combinations such as: 1-1-1-1-36, 2-2-1-1-18, 4-2-2-1-9, 6-2-3-1-6, 9-3-3-1-4, 12-3-4-1-3, 18-3-3-2-2, 36-3-3-4-1. As an example, the combination 9-3-3-1-4 denotes 9 MPI processes (3, 3, 1 processes in x, y, z direction, respectively) and 4 threads per process, and 36-3-3-4-1 represents pure MPI mode. On Thunder, the following combination is used: Intel compilers 15.0.3.187 (OpenMP 4.0), SGI MPT 2.14 and Boost C++ 1.65.1.



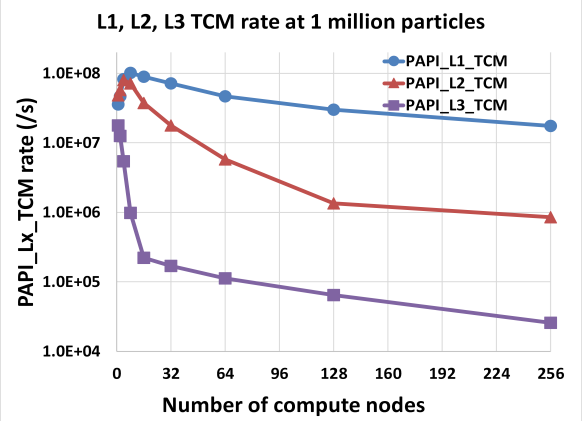
(a) 2.5k particles.



(b) 12k particles.



(c) 150k particles.



(d) 1M particles.

Fig. 18 L1, L2 and L3 cache miss (per second) comparison across simulation scales on Thunder.

8.1 Threads pinning effect of NUMA on a single Thunder node

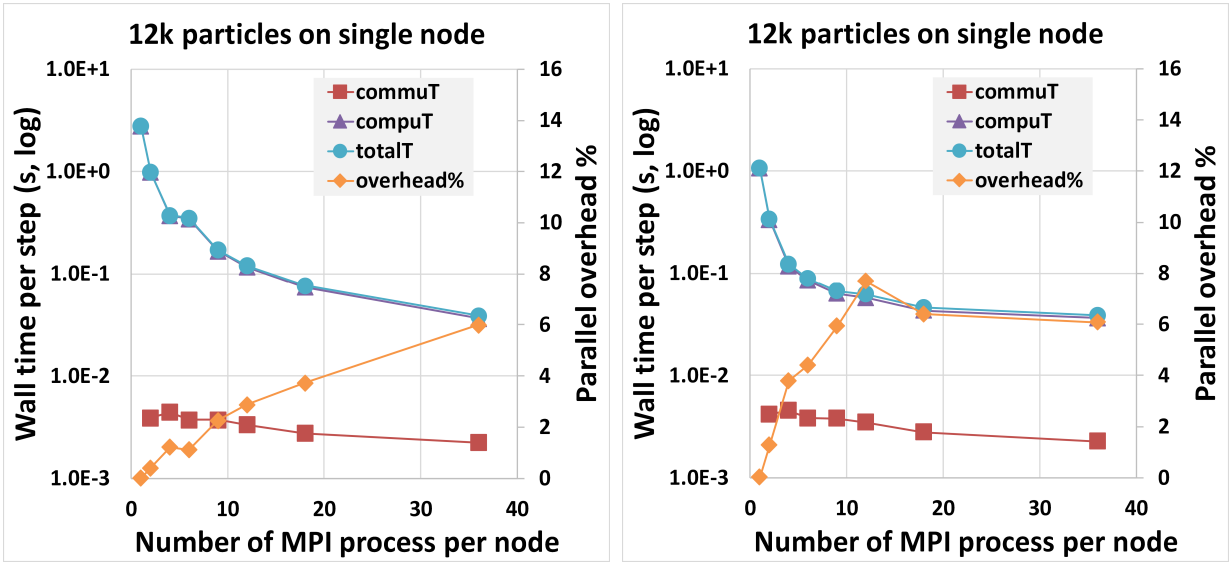
On Thunder, the software module combination of Intel compilers 15.0.3.187 (OpenMP 4.0), SGI MPT 2.14 and Boost C++ 1.65 is selected to compile and run the code. It is very important to realize that "omplace" option must be used for pinning processes and threads in hybrid MPI-OpenMP applications to achieve better performance on this SGI ICE X system with SGI MPT environment. To avoid interference between "dplace/omplace" and Intel's thread affinity interface, environment variable KMP_AFFINITY is set to disabled or OMPLACE_AFFINITY_COMPAT is set to ON.

Table 4 lists the execution time of 12k ellipsoidal particles for different processes/threads combinations with and without processes/threads affinity on a single Thunder node. Firstly, it is observed that as the number of MPI processes increases (and the number of threads per MPI process decreases accordingly, while the total number of threads remains at 36), both the computation time and communication time decrease, until the maximum performance is obtained in pure MPI mode. Pure MPI mode outperforms all of the hybrid MPI-OpenMP combinations on a single node, with or without processes/threads pinning, illustrated by Figure 19.

Secondly, the processes/threads pinning on NUMA architectures improves performance significantly when there are multiple threads per process. For example, the computation time drops from 2.78 s to 1.06 s for 36 threads per process, and it drops from 0.98 s to 0.34 s for 18 threads per process because of processes/threads affinity. As the number of threads decreases, the improvement becomes less pronounced; for example, the computation times are nearly the same, 0.036 s, for the case of 1 thread per process, i.e., pure MPI mode. The

Table 4 Execution time (s) of 12k particles without and with processes/threads affinity on a single Thunder node

	MPI processes	threads per process	commuT	migraT	computT	totalT	overhead%
no "omplace"	1	36	2.05E-05	9.61E-05	2.78E+00	2.78E+00	0.01
	2	18	3.91E-03	1.37E-04	9.84E-01	9.88E-01	0.41
	4	9	4.43E-03	1.34E-04	3.69E-01	3.74E-01	1.22
	6	6	3.74E-03	1.64E-04	3.47E-01	3.51E-01	1.11
	9	4	3.73E-03	1.41E-04	1.68E-01	1.71E-01	2.26
	12	3	3.34E-03	1.31E-04	1.17E-01	1.21E-01	2.88
	18	2	2.75E-03	9.49E-05	7.36E-02	7.64E-02	3.73
	36	1	2.24E-03	7.10E-05	3.64E-02	3.87E-02	5.99
"omplace"	1	36	3.57E-05	9.44E-05	1.06E+00	1.06E+00	0.02
	2	18	4.18E-03	9.35E-05	3.35E-01	3.39E-01	1.26
	4	9	4.54E-03	9.01E-05	1.18E-01	1.23E-01	3.78
	6	6	3.82E-03	6.20E-05	8.44E-02	8.83E-02	4.40
	9	4	3.79E-03	1.70E-04	6.30E-02	6.70E-02	5.91
	12	3	3.47E-03	1.12E-04	5.72E-02	6.19E-02	7.69
	18	2	2.79E-03	1.40E-04	4.30E-02	4.59E-02	6.40
	36	1	2.27E-03	7.19E-05	3.63E-02	3.86E-02	6.08



(a) W/O threads pinning.

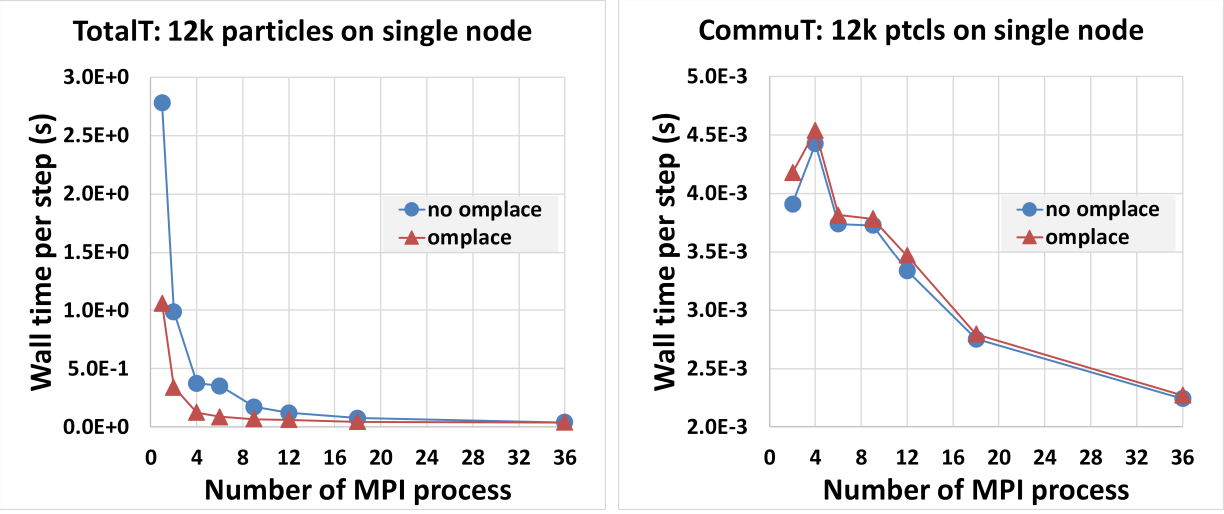
(b) With threads pinning.

Fig. 19 Execution time without and with threads pinning on a single node.

overall NUMA effect is clearly illustrated in Figure 20 (a). The communication time is compared in Figure 20 (b): it is increased very slightly by the processes/threads pinning.

Table 5 compares the best and worst performance of hybrid MPI-OpenMP mapping scheme to that of pure MPI mode. The worst performance of hybrid MPI-OpenMP consumes approximately 6 times wall clock time as pure MPI; and the best performance consumes approximately 1.2 times wall clock time as pure MPI, so it is still inferior to pure MPI mode.

It should be noted that the best performance of hybrid MPI-OpenMP occurs when there are two threads per MPI process, and pure MPI executes one thread per process. In a sense, this is not typical of how hybrid MPI-OpenMP is expected to work, for which more threads per process may be used.

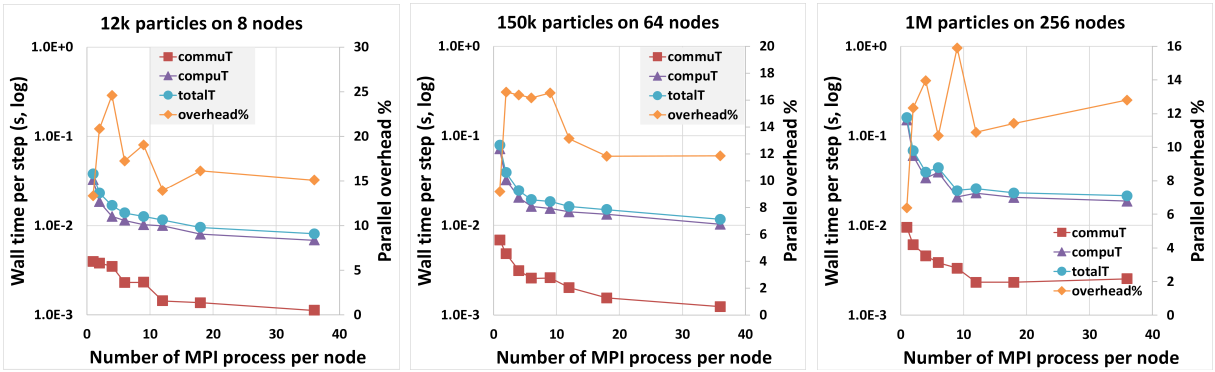


(a) Total time. (b) Communication time.

Fig. 20 Threads pinning effect on a single node.

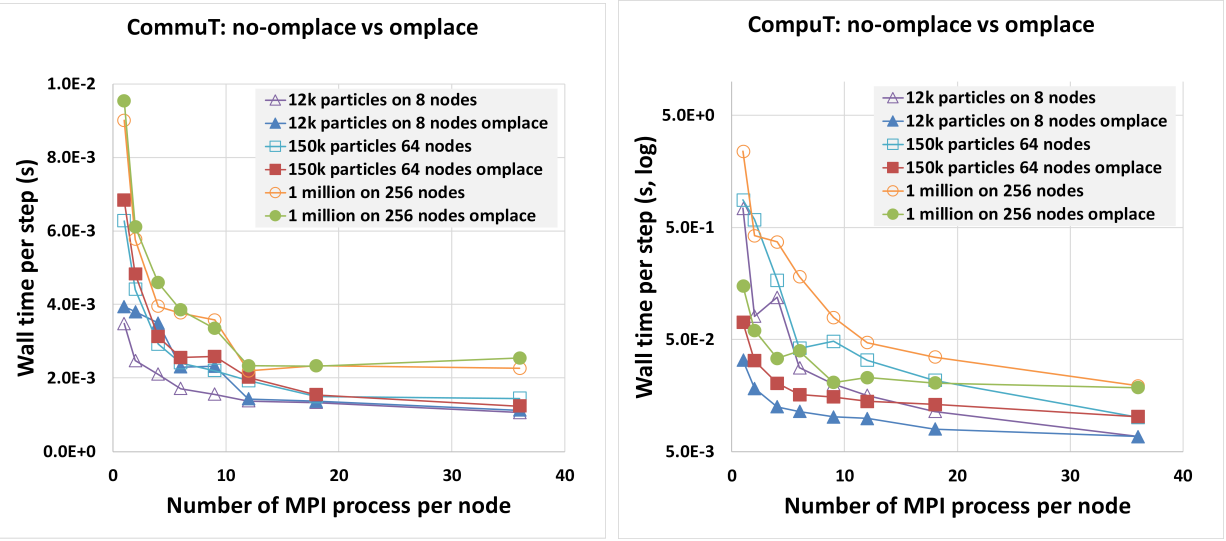
Table 5 Profiling on various hybrid MPI-OpenMP mapping scheme

	time (s)	commuT	migraT	computT	totalT	overhead%
12k, 8 nodes	pure MPI	1.12E-03	8.86E-05	6.84E-03	8.05E-03	15.1
	hybrid (best)	1.36E-03	8.46E-05	7.98E-03	9.51E-03	16.1
	hybrid (worst)	3.94E-03	9.58E-05	3.28E-02	3.79E-02	13.3
	ratio (best)	1.2	1.0	1.2	1.2	1.1
	ratio (worst)	3.5	1.1	4.8	4.7	0.9
150k, 64 nodes	pure MPI	1.23E-03	1.39E-04	1.03E-02	1.16E-02	11.9
	hybrid (best)	1.54E-03	9.25E-05	1.32E-02	1.50E-02	11.8
	hybrid (worst)	6.84E-03	3.03E-04	7.11E-02	7.83E-02	9.2
	ratio (best)	1.3	0.7	1.3	1.3	1.0
	ratio (worst)	5.5	2.2	6.9	6.7	0.8
1M, 256 nodes	pure MPI	2.54E-03	1.19E-04	1.86E-02	2.14E-02	12.8
	hybrid (best)	2.33E-03	1.23E-04	2.05E-02	2.31E-02	11.4
	hybrid (worst)	9.54E-03	6.88E-04	1.50E-01	1.61E-01	6.4
	ratio (best)	0.9	1.0	1.1	1.1	0.9
	ratio (worst)	3.8	5.8	8.1	7.5	0.5



(a) 12k particles. (b) 150k particles. (c) 1M particles.

Fig. 21 multinode performance with threads pinning across three simulation scales.



(a) Communication time.

(b) Computation time.

Fig. 22 Threads pinning effect on multinode.

8.2 Multinode behavior

Figure 21 plots the multinode performance with processes/threads pinning across three simulation scales, 12k, 150k and 1M particles. All of the scales exhibit the same overall trend as the single node behavior: as the number of MPI processes increases (and the number of threads per MPI processes decreases accordingly, while the total number of threads remains at 36), both the computation time and communication time decrease, until the maximum performance is obtained at pure MPI mode.

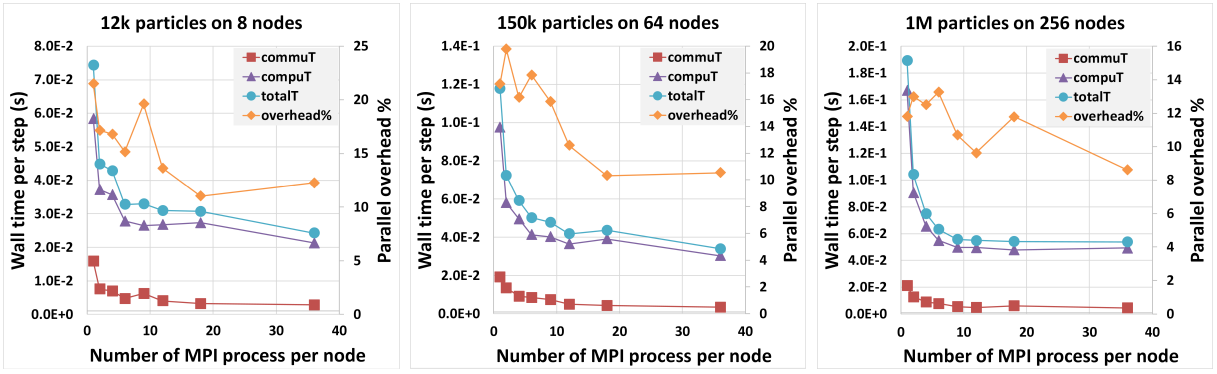
Figure 22 shows the effect of processes/threads pinning by comparing three sets of data (12k particles with vs w/o "omplace", 150k particles with vs w/o "omplace", 1M particles with vs w/o "omplace"): the communication time increases very slightly; and the computation time decreases significantly. We can conclude that processes/threads pinning on NUMA architectures improve performance for hybrid MPI-OpenMP substantially when the number of threads per process is not low.

9 FROM ELLIPSOID TO POLY-ELLIPSOID

As mentioned earlier, the contact resolution between poly-ellipsoids is approximately 6 times as expensive as that of ellipsoids. It is interesting to see how poly-ellipsoids differ in the hybrid MPI-OpenMP mode.

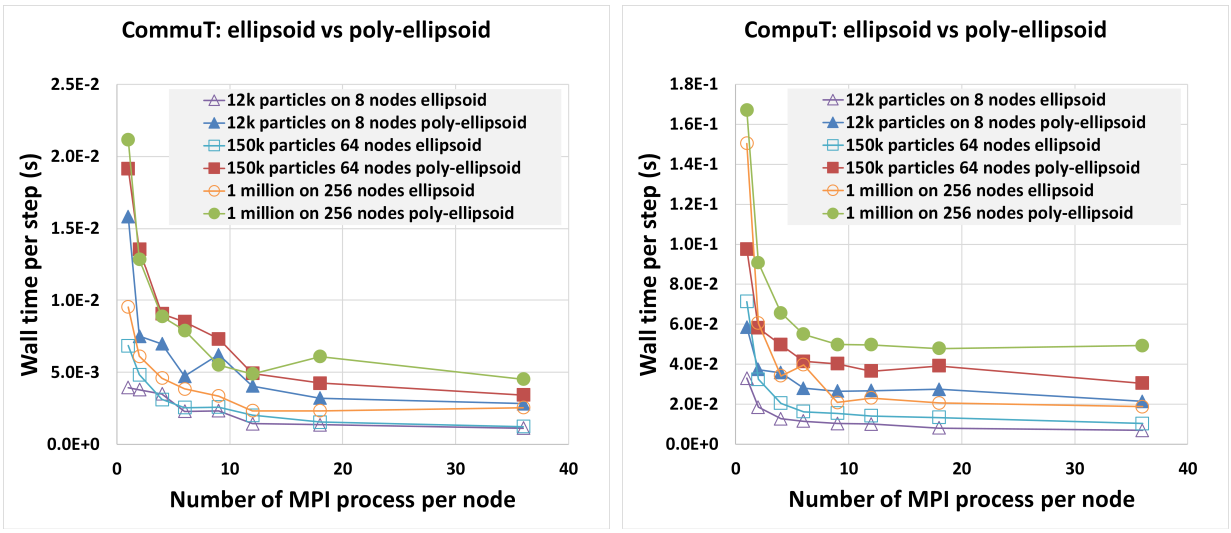
Figure 23 plots the multinode performance with processes/threads pinning across three simulation scales, 12k, 150k and 1M poly-ellipsoidal particles. All of the scales exhibit the same overall trend as the ellipsoidal particles: as the number of MPI processes increases (and the number of threads per MPI processes decreases accordingly, while the total number of threads remains 36), both the computation time and communication time decrease, until the maximum performance is obtained at pure MPI mode. The only difference is that the computation time becomes very close in the case of 18 processes (2 threads per process) and pure MPI (36 processes, 1 thread per process), although the pure MPI still outperforms hybrid MPI-OpenMP slightly.

Figure 24 compares the communication time and computation time for ellipsoids and poly-ellipsoids, respectively, across three simulation scales, 12k, 150k and 1M poly-ellipsoidal particles. Under each scale, the computational granularity (CPG) for ellipsoids and poly-ellipsoids are the same for the purpose of comparison. It is seen that both the communication time and computation time increase substantially from ellipsoids to poly-ellipsoids (12k ellipsoids vs poly-ellipsoids, 150k ellipsoids vs poly-ellipsoids, 1M ellipsoids vs poly-ellipsoids, respectively). This is not surprising: poly-ellipsoids are much more expensive in contact resolution



(a) 12k particles. (b) 150k particles. (c) 1M particles.

Fig. 23 multinode performance of poly-ellipsoids across three simulation scales.



(a) Communication time. (b) Computation time.

Fig. 24 Execution time of ellipsoid and poly-ellipsoid on multinode.

because of their shape, and they result in larger MPI transmission volume because of their more complicated data structure.

10 SUMMARY

The paper presents performance comparison between pure MPI and hybrid MPI-OpenMP modes for 3D Discrete Element Method (DEM) of ellipsoid-like complex-shaped particles. The following influences are also examined: memory/cache hierarchy, processes/threads pinning, variation of hybrid MPI-OpenMP mapping scheme, ellipsoid vs poly-ellipsoid. Based on the foregoing numerical simulations and relevant analyses, the following conclusions can be drawn:

Complex-shaped 3D DEM features a low-fraction neighbor estimate but high-fraction contact resolution in terms of computational cost for interparticle contact detection, which dominates the overall computation. The more complex the shapes, the lower fraction the neighbor estimate.

On a single node, OpenMP achieves high efficiency (close to 100%) in interparticle contact detection, but unparallelizable or inherently sequential part of the codes prevents OpenMP from achieving the same high efficiency in overall performance. The redundant MPI parallelization code and computation offsets OpenMP

performance gain to some extent. As a result, MPI executes slightly faster than OpenMP in the MPI-parallelized framework.

Both pure MPI and hybrid MPI-OpenMP give the same trend: the computation time and communication time (thus the total time) decrease, and the decrease rates are high at the very beginning and slow down later as the number of compute nodes increases.

In hybrid MPI-OpenMP mode, as the number of MPI processes increases (and the number of threads per MPI process decreases accordingly), the total execution time decreases, until the maximum performance is obtained in pure MPI mode; the processes/threads pinning on NUMA architectures improves performance significantly when there are multiple threads per process, whereas less pronounced as the number of threads per process decreases; and both the communication time and computation time increase substantially from ellipsoids to poly-ellipsoids. Pure MPI achieves both lower computational granularity (thus higher spatial locality) and lower communication granularity (thus faster MPI transmission) than hybrid MPI-OpenMP in 3D DEM, where computation dominates communication, and it works more efficiently than hybrid MPI-OpenMP in 3D DEM simulations of ellipsoidal and poly-ellipsoidal particles.

Compliance with Ethical Standards

Funding: This study was funded by ONR MURI grant N00014-11-1-0691.

Conflict of Interest: The authors declare that there is no conflict of interest.

Acknowledgements We would like to acknowledge the support provided by ONR MURI grant N00014-11-1-0691, and the DoD High Performance Computing Modernization Program (HPCMP) for granting us the computing resources required to conduct this work.

References

Baugh Jr JW, Konduri R (2001) Discrete element modelling on a cluster of workstations. Engineering with Computers 17(1):1–15

Chandra R (2001) Parallel programming in OpenMP. Morgan kaufmann

Chorley MJ, Walker DW (2010) Performance analysis of a hybrid mpi/openmp application on multi-core clusters. Journal of Computational Science 1(3):168–174

Dagum L, Enon R (1998) Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE 5(1):46–55

Delaney GW, Cleary PW, Sinnott MD, Morrison RD (2010) Novel application of dem to modelling comminution processes. In: IOP Conference Series: Materials Science and Engineering, IOP Publishing, vol 10, p 012099

Drosinos N, Koziris N (2004) Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, p 15

Grest GS, Dünweg B, Kremer K (1989) Vectorized link cell fortran code for molecular dynamics simulations for a large number of particles. Computer Physics Communications 55(3):269–285

Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message-passing interface, vol 1. MIT press

Henty DS (2000) Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, IEEE Computer Society, p 10

Jagadish HV, Ooi BC, Tan KL, Yu C, Zhang R (2005) idistance: An adaptive b+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems (TODS) 30(2):364–397

Jost G, Jin HQ, anMey D, Hatay FF (2003) Comparing the openmp, mpi, and hybrid programming paradigm on an smp cluster. ntrsnasagov

Lim KW, Andrade JE (2014) Granular element method for three-dimensional discrete element calculations. International Journal for Numerical and Analytical Methods in Geomechanics 38(2):167–188

Luecke G, Weiss O, Kraeva M, Coyle J, Hoekstra J (2010) Performance analysis of pure mpi versus mpi+ openmp for jacobi iteration and a 3d fft on the cray xt5. Cray User Group 2010 Proceedings

Maknickas A, Kačeniauskas A, Kačianauskas R, Balevičius R, Džiugys A (2006) Parallel dem software for simulation of granular media. Informatica 17(2):207–224

Michael JQ (2003) Parallel Programming in C with MPI and OpenMP. New York: McGraw-Hill Press

Muja M, Lowe DG (2009) Fast approximate nearest neighbors with automatic algorithm configuration. VISAPP (1) 2:331–340

Munjiza A, Andrews K (1998) Nbs contact detection algorithm for bodies of similar size. International Journal for Numerical Methods in Engineering 43(1):131 – 149

Ng TT (1994) Numerical simulations of granular soil using elliptical particles. Comput Geotech 16(2):153 – 169

Ng TT (2004) Triaxial test simulations with discrete element method and hydrostatic boundaries. Journal of Engineering Mechanics 130(10):1188 – 1194

Pacheco PS (1997) Parallel programming with MPI. Morgan Kaufmann

Pal A, Agarwala A, Raha S, Bhattacharya B (2014) Performance metrics in a hybrid mpi–openmp based molecular dynamics simulation with short-range interactions. Journal of Parallel and Distributed Computing 74(3):2203–2214

Peters JF, Hopkins MA, Kala R, Wahl RE (2009) A polyellipsoid particle for nonspherical discrete element method. Engineering Computations 26(6):645–657

Rabenseifner R, Hager G, Jost G (2009) Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on, IEEE, pp 427–436

Vedachalam V, Virdee D (2011) Discrete element modelling of granular snow particles using liggghts. M Sc, University of Edinburgh

Washington DW, Meegoda JN (2003) Micro-mechanical simulation of geotechnical problems using massively parallel computers. International journal for numerical and analytical methods in geomechanics 27(14):1227–1234

Wellmann C, Lillie C, Wriggers P (2008) A contact detection algorithm for superellipsoids based on the common-normal concept. Engineering Computations 25(5):432–442

Williams JR, Pentland AP (1992) Superquadrics and modal dynamics for discrete elements in interactive design. Engineering Computations 9(2):115–127

Williams JR, Perkins E, Cook B (2004) A contact algorithm for partitioning n arbitrary sized objects. Engineering Computations 21(2/3/4):235–248

Yan B, Regueiro R (2018a) Comparison between o(n2) and o(n) neighbor search algorithm and its influence on superlinear speedup in parallel discrete element method (dem) for complex-shaped particles. Engineering Computations 35(6):2327–2348

Yan B, Regueiro RA (2018b) A comprehensive study of mpi parallelism in three-dimensional discrete element method (dem) simulation of complex-shaped granular particles. Computational Particle Mechanics 5(4):553–577

Yan B, Regueiro RA (2018c) Superlinear speedup phenomenon in parallel 3d discrete element method (dem) simulations of complex-shaped particles. Parallel Computing 75:61–87

Yan B, Regueiro RA, Sture S (2010) Three-dimensional ellipsoidal discrete element modeling of granular materials and its coupling with finite element facets. Engineering Computations 27(4):519–550