Powder Technology 278 (2015) 234–247

Contents lists available at ScienceDirect

Powder Technology

journal homepage: www.elsevier.com/locate/powtec

Hybrid parallelization of the LIGGGHTS open-source DEM code

R. Berger [a,*], C. Kloss [c], A. Kohlmeyer [b], S. Pirker [a]

[a] Johannes Kepler University Linz, Department on Particulate Flow Modelling, Altenbergerstrasse 69, 4040 Linz, Austria
[b] College of Science & Technology, Temple University, Philadelphia, PA, USA
[c] DCS Computing GmbH, Altenbergerstr. 66a - Science Park, 4040 Linz, Austria

ARTICLE INFO

Article history:
Received 7 November 2014
Received in revised form 10 February 2015
Accepted 14 March 2015
Available online 24 March 2015

Keywords:
LIGGGHTS
Discrete Element Method
Hybrid parallelization
MPI
OpenMP

ABSTRACT

This work presents our efforts to implement an MPI/OpenMP hybrid parallelization of the LIGGGHTS open-source software package for Discrete Element Methods (DEM). We outline the problems encountered and the solutions implemented to achieve scalable performance using both parallelization models. Three case studies, including two real-world applications with up to 1.5 million particles, were evaluated and demonstrate the practicality of this approach. In these examples, better load balancing and reduced MPI communication led to speed increases of up to 44% compared to MPI-only simulations.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

LIGGGHTS is an open-source software package used for numerical simulation of granular materials and of heat transfer [1]. It is a DEM [2] code implemented on top of LAMMPS [3], a molecular dynamics code developed by Sandia National Laboratories. Technically, LIGGGHTS is a fork of LAMMPS, the main adaptations being the addition of mesh geometry support, granular models for particle–particle and particle–wall interactions, and particle–particle and particle–wall heat transfer. Numerous other features are added continuously to support special requirements for large industrial cases. Such cases include simulations of bulk solids needed in the chemical industry and steel industry. Examples are the coating process of pills in the pharmaceutical industry and the optimization of dust filters in industrial plants.

LIGGGHTS operates on macroscopic particles and tracks the trajectory of each. It is designed around an integration loop which integrates Newton's second law and resolves particle–particle and particle–wall collisions using a soft-sphere approach. Spring-dashpot models are used to compute forces caused by particle–particle interactions (pair forces) and particle–wall interactions. Additionally, volume forces such as gravity are applied. The total amount of computational work during collisions is cut in half by utilizing Newton's third law, thus avoiding recomputation of the same force with a different orientation.

Fig. 1 illustrates the flow of control for Velocity-Verlet time integration as implemented in LAMMPS and LIGGGHTS. The two time integration steps, which update particle positions and velocities, are bracketed by system modification hooks that allow the simulated system to be manipulated in various ways. Binning of particles and Verlet lists [4] are used to improve the efficiency during collision detection. Periodic spatial sorting of particles ensures that those in close proximity are in nearby memory locations, thereby increasing cache utilization.

Because of the common code base, many positive performance characteristics are inherited from LAMMPS. Both codes can be used in a parallel environment through message passing (MPI) [5]. The original MPI code of LAMMPS used a static domain decomposition [3,6,7] which partitions space such that the area of communication between MPI ranks is minimized. A similar approach was taken by Kacianauskas et al. [6], but they observed an increase in computational effort for simulations of polydisperse material compared to monodisperse material. Gopalakrishnan and Tafti [7] reported an almost ideal speed increase in the DEM portion of a CFD-DEM fluidized bed simulation on up to 64 processors, and a parallel efficiency of 81% on 256 processors.

Static domain decomposition works well for homogeneous condensed matter simulations, but in DEM the typically inhomogeneous and changing distribution of particles across subdomains result in performance-limiting load imbalances. This motivated the development of a load-balancing scheme in LIGGGHTS through which domain boundaries are dynamically adjusted at runtime. This has since been backported into LAMMPS [8] and is described in Section 2.1.

The idea of moving domain boundaries is not new. It has previously been implemented by Srinivasan et al. [9] in the context of molecular dynamics. They generate equally loaded rectangular regions by moving

* Corresponding author.
E-mail addresses: richard.berger@jku.at (R. Berger), christoph.kloss@dcs-computing.com (C. Kloss), akohlmey@gmail.com (A. Kohlmeyer), stefan.pirker@jku.at (S. Pirker).

http://dx.doi.org/10.1016/j.powtec.2015.03.019
0032-5910/© 2015 Elsevier B.V. All rights reserved.

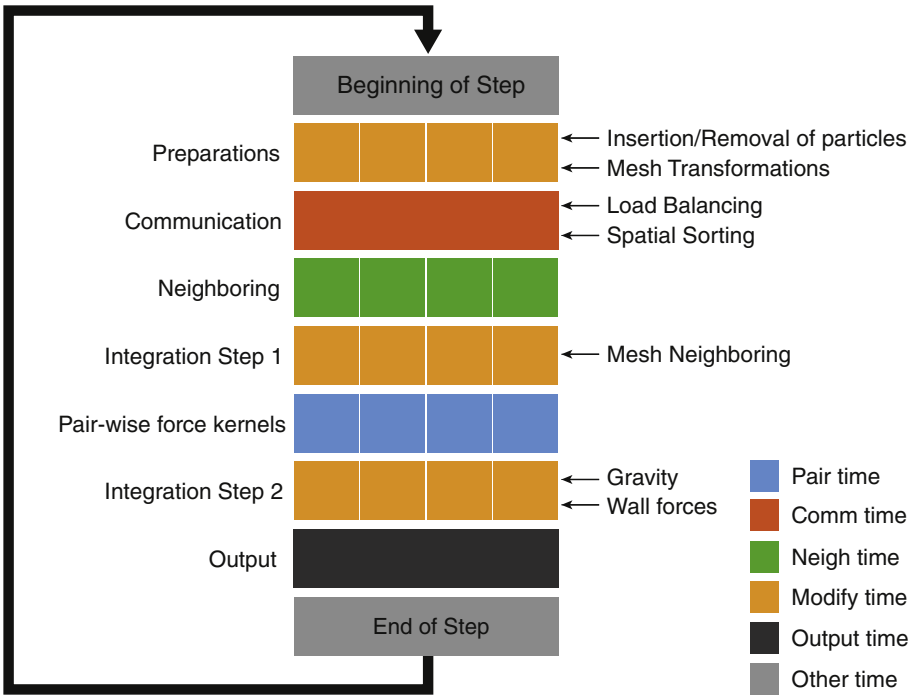


Fig. 1. LIGGGHTS integration loop (simplified). At the beginning of each step, particles may be inserted and meshes transformed. Particles are exchanged between MPI processes, and load balancing may occur. After communication, neighbor lists are built. Forces are computed between integration steps 1 and 2. The parts of the loop that can be parallelized are shown as 4 separate squares. The color codes indicate how these sections contribute to the total timing breakdown.

domain boundaries in increments of a load-discretizing (LD) grid that is coarser than the computation domain (CD) grid. Load balancing occurs on multiple levels by adjusting cuts in the x, y and z directions. The load balancing is based on particle density or number of pairs in each subdomain. In their simulations, a reduction in computation time by as much as 50% was achieved.

More sophisticated load-balancing techniques were employed by Plimpton et al. [10], who applied three different decompositions in a joint finite-element (FE) and smoothed particle hydrodynamics (SPH) code. They employed static FE decomposition of mesh elements, while SPH-decomposition and contact-decomposition of contact-nodes and SPH-particles were performed dynamically using Recursive Coordinate Bisection (RCB) [11]. This geometric algorithm was chosen because it produced well-shaped subdomains and exhibited linear scaling to the problem size.

RCB and many other dynamic load-balancing algorithms were evaluated by Hendrickson and Devine [12]. They listed geometric methods like RCB as being well suited to geometric problems such as particle simulations. RCB in particular is considered to be one of the fastest and easiest to implement in this group of algorithms. It also has the valuable property of being incremental, meaning that small changes in a domain only lead to small changes in the decomposition. This property minimizes expensive communication between processors.

Recently, LAMMPS has also gained the ability to use RCB directly for its MPI decomposition [13]. However, this implementation only balances the number of particles in each subdomain, not the actual workload. It is also incompatible with the current MPI parallelization of meshes in LIGGGHTS.

A different parallelization strategy, known as particle subset method, was employed by Kafui et al. [14] in their CFD-DEM code. They applied a "mincut" graph-partitioning algorithm to a graph of particles and their contacts that generates partitions of particles with a minimal number of contacts with particles in another partition. Based on these partitions, for each MPI process working on a single partition, particles are assigned and halo regions defined. Partitions are recomputed at regular intervals for dynamic balancing.

The desire for better load balancing and better utilization of available compute resources motivated many groups to experiment with the particle subset method using shared-memory parallelizations in their codes. Since fall 2011, LAMMPS has included an add-on package called USER-OMP [15], which provides multi-threaded and thread-safe variants of selected modules and subroutines, in particular force kernels, neighbor-list builds and a few selected other modules. In 2013, our exploration of using these OpenMP modifications for LIGGGHTS started [16].

Concurrently, Amritkar et al. [17] developed an OpenMP parallelization for MFIX DEM code which uses the particle subset method. They argued that for the N-body particulate phase of their CFD-DEM simulation, a parallelization over the number of particles is more suitable. To support their claim, they presented measurements of a fluidized bed simulation (uniform particle distribution) and a rotary kiln heat transfer simulation (non-uniform distribution). Despite higher overheads for fetching non-local data, the better load balancing makes the OpenMP parallelization 50–90% faster than MPI-only. To achieve optimal speed increases, they ensured that data was stored locally by following the first-touch policy, and that thread/process affinity was set using placement tools.

Finally, Liu et al. [18] further expanded on existing MPI and OpenMP work and described a hybrid MPI/OpenMP parallelization of their MFIX-DEM solver. They emphasized that data locality and thread placement policies play a critical role in scaling OpenMP to large core counts. They also mentioned the necessity of distinguishing between private and global data in multi-threaded code, which avoids race conditions by each thread using its own copy of data and reductions. Due to reduced MPI communication, their hybrid becomes faster. Scaling was presented using a coupled CFD-DEM run of a 3D fluidized bed simulation. They reported speed increases of 185× on 256 cores (72% efficiency) of their hybrid parallelization compared to 138× using a standalone MPI computation with 5.12 million particles.

Previous works, such as Henty et al. [19], also developed MPI, OpenMP and hybrid versions of a DEM benchmark code and evaluated their performance. They explained the difficulties introduced by multi-threading, such as updating a global force array from multiple threads. Due to overheads introduced to their OpenMP implementation, however, their hybrid models were not more efficient on SMP nodes.

Rabenseifner et al. [20] provided an overview of parallel programming models on hybrid platforms and described the difficulties of mapping different approaches to hardware. They listed potential gains of using a hybrid approach, including the benefit of improving load balance and reducing memory consumption.

The potential benefits and pitfalls of hybrid parallelization using MPI and OpenMP were also mentioned by Smith et al. [21]. In their work, they showcased results of a Game of Life implementation and a mixed-mode Quantum Monte Carlo code. They concluded that a mixed strategy might not always be the most effective, but in situations where MPI exhibits poor scaling due to load imbalance an improvement might be achieved.

Any multi-threaded code must differentiate between local data manipulated by single threads and global data accessible by multiple threads. Other groups [19,17,18] have reported on either working with local copies of data or using synchronization mechanisms, such as reductions, locks, critical sections and atomic updates.

Most algorithms were parallelized simply by instrumenting them with OpenMP compiler directives. Work-sharing constructs such as parallelized for-loops were used. By default, these constructs produce a static decomposition of loop iterations. This means that, if a computation is applied to a list of elements, this list is split into chunks of equal size, each of which is then processed by a separate thread.

It is trivial to parallelize portions of code using OpenMP when each computation is independent of all other computations. However, calculations such as particle–particle collisions are inherently dependent on each other if they use Newton's third law and try to avoid duplication of force computations. Each computed force of a contact pair is applied to each contact partner, but with opposing directions. Without proper safeguards, updating the resulting force on a particle by multiple threads can lead to lost or even false data, since such operations are not atomic. Atomic updates of this kind would have to be requested explicitly, and incur a significant performance penalty.

One naive approach to solving this problem is to wrap all write operations on shared data in critical sections or to manually place locks. However, this effectively serializes these portions of code and reduces the total amount of exploitable parallelism in the application, as argued by Amdahl's law [22].

Another solution is giving each thread its own force array to write in. After completion, all of these arrays are reduced to one global array. This is called array reduction and is the strategy currently used in the USER-OMP package [15]. The main downside of this approach is its extensive use of memory and memory bandwidth. Each thread needs to allocate enough storage to hold computation results for each particle. Contention for memory bandwidth and the growing overhead of the reduction operation with increasing amounts of threads limit the parallel efficiency of this approach.

Henty et al. [19] generate a look-up table which includes only particles that are updated by more than one thread. This table is then consulted during each update of the force array to protect critical accumulations with atomic locks. Since the look-up table is valid for many force computations, and most force computations do not require protection, this method minimizes the locking overhead.

In this work, we describe our efforts of bringing multi-threading optimizations to LIGGGHTS in order to improve utilization of available computational resources. We created a second layer of parallelization using OpenMP on top of the existing dynamic MPI domain decomposition. Both particle–particle interactions and particle–wall interactions with meshes are load-balanced dynamically by a combination of methods described in Section 2. While many of the individual building blocks of this hybrid parallelization are well known in the literature, to the best of our knowledge, our work is the first to illustrate the benefit of combining multiple load-balancing techniques using two levels of parallelization. It showcases how partitioning can be used to avoid data races and to implement a lock-free version of important DEM routines. By doing some additional work in advance, we avoid look-up tables in our force loops and reduce safeguard complexity inside the loop to constant time.

Distributing *particles* evenly among threads does not imply an even distribution of the *workload*. Therefore, we employ dynamic load balancing at the OpenMP level. A simpler load-balancing mechanism is used to load-balance particle–wall interactions with meshes. Three case studies, including two real-world examples, are introduced in Section 3, the results of which are presented in Section 4 to demonstrate how our hybrid parallelization handles a variety of loads.

2. Methods

The overall concept underlying our MPI/OpenMP hybrid parallelization is illustrated in Fig. 2. We continue to use existing MPI domain decompositions to generate subdomains. The existing MPI load-balancing mechanism can still be used to adjust boundaries dynamically along one or more axes. This allows us to create a first (coarse) distribution of the workload, which enables us to utilize multiple compute nodes and to avoid – through process pinning – costly communication between nodes and sockets.

After domain decomposition, each MPI process further distributes the workload of particle–particle and particle–wall interactions by splitting them into partitions. These are recomputed periodically to adjust to changing workloads. Partitions are processed by a team of threads, and each partition is assigned to a single thread. By making our algorithms aware of partitions, we are able to implement force computations without the need for critical sections or custom-placed locks. The following sections describe the individual building blocks of our implementation.

2.1. Dynamic MPI domain decomposition

The implementation of the MPI domain decomposition allows users to specify the number of domain cuts in the x, y and z directions. Using this information, it produces a Cartesian grid of subdomains which is mapped to MPI processes. Each process is responsible for computations inside its subdomain. It therefore “owns” all particles and mesh triangles in its region. Since these elements might interact with other particles or triangles in neighboring processes, each process maintains copies of remote data from halo regions, which extend into neighboring subdomains. Their size is defined by a cutoff that is proportional to the largest particle diameter. Particles inside a halo region are called *ghost particles*, and their information – such as position, velocity and contact history – is synchronized from the process that owns the particle data. The same approach is taken for mesh triangles by introducing *ghost triangles*.

Starting from this static Cartesian grid of subdomains, the dynamic domain decomposition adjusts the location of subdomain cuts along one or more directions periodically. This is done for each direction individually. First, the number of particles in each slice of a direction is aggregated. Ideally, the number of particles is evenly distributed among all slices in that direction. The imbalance in the distribution is measured by the maximum number of particles in a single slice divided by the total number of particles. If this value exceeds a threshold, load balancing along that direction commences.

Slices are resized by adjusting their boundaries using a recursive multi-sectioning algorithm. Based on the previous split locations and the aggregated sum of particles up to these positions, the algorithm interpolates new split locations so each slice could contain the desired number of particles. Using these new splits, the actual number of particles in the new slices is then computed over all processors and the

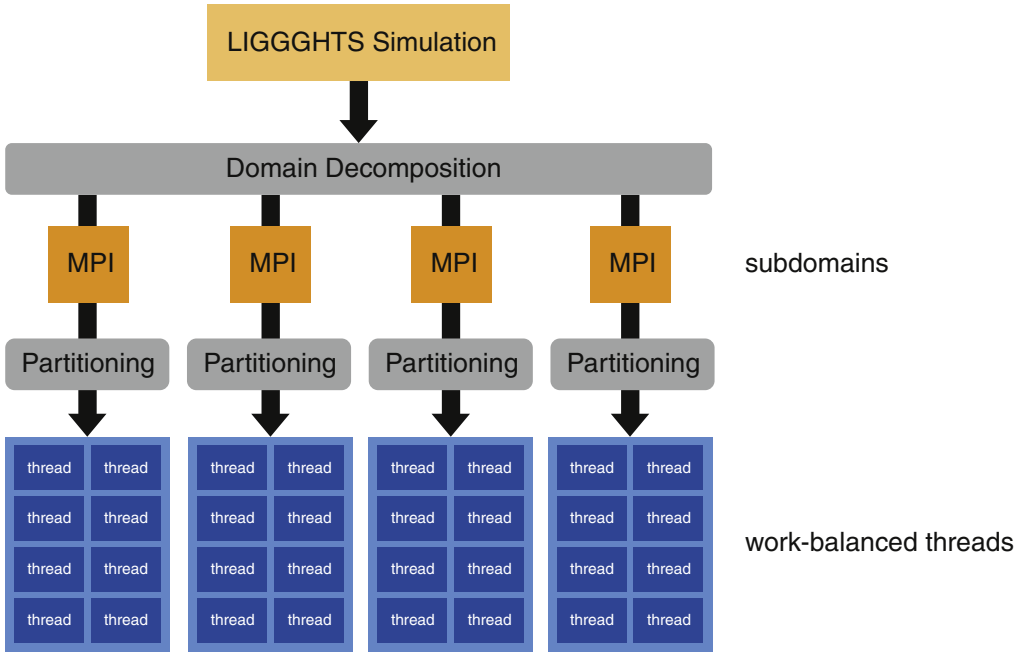


Fig. 2. Overview of the MPI/OpenMP hybrid parallelization. The workload of a simulation is first distributed along one or more axes using a simple MPI decomposition. MPI load balancing, which adjusts boundaries over time, can still be used. Each MPI subdomain then further divides its subdomain into partitions of equal workload. All work-intensive algorithms launch multiple threads that work only on particles in a partition assigned to them.

quality of the new decomposition evaluated. This is repeated until either the desired distribution of particles is reached by a fair margin or the maximum number of attempts has been made.

Finally, all processors are made aware of the new decomposition, and particle data is migrated. During the next communication phase inside the integration loop, this triggers an update of particle neighbor lists and other helper data structures. The following summarizes the sequence of important operations during and after MPI communication:

1 (Optional) Periodically adjust domain boundaries along one or more axes. If necessary, migrate data.
2 Exchange particles and contact history data between MPI processes
3 Apply spatial sorting of local particle data to each MPI process to increase data locality and improve cache utilization
4 Update particle neighbor lists
5 Exchange mesh triangles and contact history data between MPI processes
6 Update mesh triangle–particle neighbor lists

Note that, for historical reasons, mesh triangle migration occurs at a later stage, but before any new force computations are performed. This is why some MPI communication occurs after the first integration step, which is denoted "Modify" time in Fig. 1 and Section 4.

While this dynamic decomposition alone improves load balancing of MPI-only simulations significantly, it is limited by its focus on balancing the number of particles rather than the actual workload among processors. Meshes are simply decomposed on the basis of the subdomains obtained by balancing particles. This can shift computational effort to individual processors, which leads to additional undesirable load imbalance.

2.2. OpenMP parallelization overview

On top of the MPI domain decomposition, our hybrid parallelization allows each MPI process to spawn multiple OpenMP threads. This is controlled either by the LIGGGHTS input script or through the OpenMP environment variable OMP_NUM_THREADS. Since for better performance threads should not migrate between cores, we implemented thread-pinning manually inside our implementation, and the development was based on the GCC 4.8–4.9 OpenMP 3.1 implementation. We relied on the fact that this OpenMP implementation uses internally the pthread library, which allowed us to implement thread-pinning using pthread methods inside of OpenMP parallel regions. In future versions, this could be replaced with the more generic "Places" feature introduced in OpenMP 4.

In order for LIGGGHTS to scale well using OpenMP, all major portions of computation needed to be converted to multi-threaded code. The USER-OMP package of LAMMPS, which contains multi-threaded versions of many essential algorithms, such as neighbor list building and Velocity-Verlet integration, had already done this in part. The focus of our work is on the OpenMP parallelization of granular particle–particle and particle–wall interactions.

The lock-free strategy developed for our hybrid parallelization uses a combination of well-known algorithms to obviate the need for such synchronization mechanisms. We achieve this by additional work for data partitioning and rearrangement. This is done during the communication and load-balancing phase of LIGGGHTS and extends the version presented in Section 2.1:

1 (Optional) Periodically adjust domain boundaries along one or more axes. If necessary, migrate data.
2 Exchange particles and contact history data between MPI processes
3 Use RCB to partition local particle data into $N_{threads}$ partitions
4 Sort local particle data on each MPI process based on partitions and apply spatial sorting to increase data locality and improve cache utilization
5 Update particle neighbor lists
6 Exchange mesh triangles and contact history data between MPI processes
7 Update mesh triangle–particle neighbor lists
8 Distribute mesh triangles based on particle neighbors between threads
9 Distribute particles based on triangle neighbors between threads

Steps 3 and 4 constitute dynamic load balancing of particle–particle interactions across threads. Section 2.3.1 describes how particle data is partitioned by the RCB algorithm. In Section 2.3.2, we outline necessary changes to the existing spatial sorting algorithm and the enforced

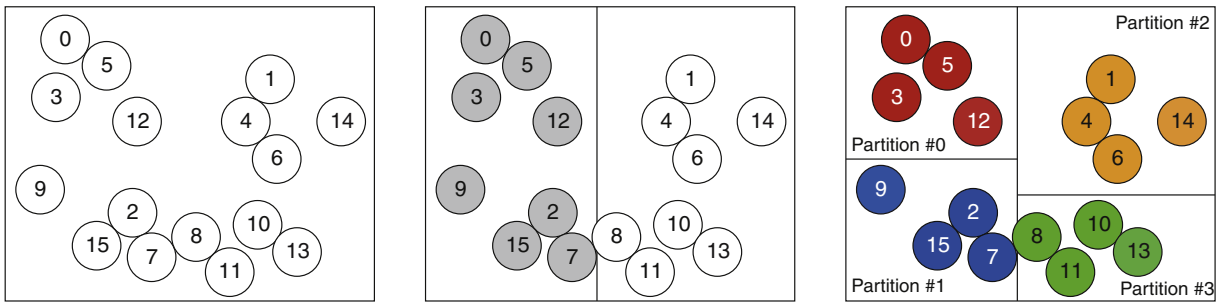


Fig. 3. Illustration of the Recursive Coordinate Bisection (RCB) algorithm. It gradually partitions space by inserting cut planes, balancing the number of nodes based on their weights.

partition-aligned particle memory layout. The resulting multi-threaded granular force kernels are described in Section 2.3.3. Section 2.3.4 mentions some performance considerations and optimizations needed to keep this approach scalable. Load balancing of particle–wall interactions occurs in Steps 8 to 9, which are explained in Section 2.4.

2.3. OpenMP parallelization of particle–particle interactions

2.3.1. Dynamic load-balancing using the Zoltan library

A general approach to distributing the particle–particle collision workload across threads was implemented using existing partitioning algorithms available in the Zoltan library [23,24]. This library – developed by Sandia National Laboratories – contains many traditional graph-based and geometric decomposition algorithms. Use of the library requires implementing callback functions, giving access to necessary data such as particle locations. All algorithms return a set of lists which can then be used to generate a mapping of particles to threads.

By default, we use RCB for partitioning [11]. This algorithm gradually introduces cuts in space, thus dividing partitions recursively into equal halves, as illustrated in Fig. 3. Since not all particle contacts generate equal amounts of work, each particle is given a weight proportional to the number of contact partners. This allows Zoltan RCB to balance particle partitions based on their weights. Zoltan returns a list of particle IDs and their corresponding partition IDs, which each particle then stores. Each partition ID corresponds to the thread that is responsible for that partition.

By periodically updating partitions and using the number of contact partners as feedback, this method keeps the total number of particle–particle contacts evenly distributed across threads.

2.3.2. Partitioned spatial sorting (PSS)

After assigning each particle to a partition, particle data should be rearranged. Keeping particle data sorted by partitions allows threads to work on contiguous chunks of memory, which improves cache performance. However, partition-based sorting alone does not have a lasting effect. The spatial sorting algorithm available in LIGGGHTS would destroy any artificial particle ordering. It rearranges all local particle data such that it optimizes the access pattern based on their position.

For this reason, the global spatial sorting of particle data had to be replaced. Sorting particles by partitions improves data locality to some extent, but this alone does not achieve the same quality as explicit spatial sorting. Depending on the number of particles assigned to each thread, particle data might still be scattered inside a partition. Neighboring particles need not necessarily be placed next to each other in memory. This can lead to sub-optimal performance and should be avoided.

To resolve this issue, partition-based sorting and spatial sorting were therefore combined into what we call *PSS*. The major difference in this algorithm is that it takes partition boundaries into consideration. The procedure is visualized in Fig. 4 and rearranges data which is taken from the scenario shown in Fig. 3. All sorting operations are performed on integer arrays which contain only particle IDs. After partition-based sorting of these particle indices, spatial sorting is applied to each partition separately. This generates a permutation vector for each partition, and these are then merged to form a global permutation vector. Since the original spatial sorting algorithm [4] also produces such a global permutation vector, data migration then occurs using the same data migration routines.

2.3.3. Multi-threaded granular particle–particle force kernels

Introducing particle partitions to an MPI subdomain allows us to create working sets which can be processed almost independently by multiple threads. Each thread works only on particles that are assigned to its partition. Since collisions of particles in the same partition do not introduce any write conflicts, no synchronization mechanisms such as locks or critical sections are needed. This enables use of Newton's third law within a partition and avoids duplicated force calculations.

Contact partners which are not in the same partition can easily be determined by comparing their partition assignments. For such cases, two different strategies were implemented to resolve the write conflict:

Work duplication: One solution is not to let threads apply forces to particles which are not their responsibility and to avoid writing to another thread's memory. Rather, if the two particles live in different partitions, the threads of both contact partners compute the force of the contact. Neighbor lists which only consider the upper triangle of the force matrix must be extended accordingly to include these additional force computations when necessary. This method therefore effectively duplicates work in conflict cases, but it avoids costly thread synchronization.

```
#pragma omp parallel
for(each particle i in partition of current thread) {
  for(each neighbor particle j) {
    // neighbors j are particles in the same partition
    // whose contact F_ji force has not been
    // computed yet as -F_ij AND
    // particles which live in other partition

    // compute F_ij

    // apply force F_ij to particle i

    // check if particle j is in same partition as particle i
    if(thread[j] == thread[i]) {
     // apply force -F_ij to particle j
    } else {
     // do nothing.
     // newton's third law inactive,
     // will be recomputed as F_ji by thread owning partition of j
    }
  }
}
```

Patch-up list: Instead of duplicating work, each thread can use a list to store force updates to particles which are not in the same partition. After all threads have finished working on their partitions, all force updates in these lists are applied serially. Because of the relatively low number of conflicts compared to the number of regular collision pairs, this solution is a good compromise. It is similar to array reduction, but avoids allocation of large force arrays for all particles by each thread.

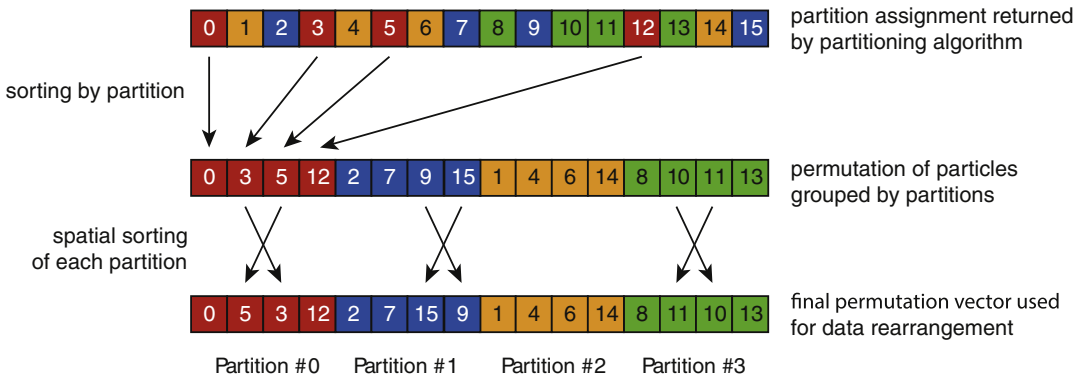


Fig. 4. Distributing all particles across partitions of equal size is done by Zoltan, but particles must then be sorted by partition to optimize performance per thread. However, the generated particle order is still not optimal. Partitioned Spatial Sorting; each partition is spatially sorted separately, creating a permutation vector which aligns neighbor data in a cache-efficient way.

Both approaches lead to deterministic results. This means that the resulting forces stored as double floating-point numbers are not dependent on the thread scheduling. They may still differ from those in serial runs, but this is due to differences in the ordering of operations between the serial and threaded executions. No significant performance difference has been observed between the two implementations. All our results presented below were collected using the first strategy, which duplicates force computations when necessary. Listing 1 shows a pseudo-code implementation of the resulting parallelized particle–particle force loop. Although it closely resembles granular force kernels, which are well known in the published literature, this multi-threaded version of the code is free of any explicit thread synchronization due to partitioning and uses adapted neighbor lists which duplicate work in conflict cases.

2.3.4. Avoiding continuous repartitioning

A weakness of our partition approach is the requirement that these partitions be valid at all times. Partitions are defined as ranges inside of global particle arrays. Any change in the domain boundaries, or insertion or deletion of particles will trigger re-neighboring of particles and must therefore prompt a partitioning update.

Particles which enter a new subdomain must be assigned to a partition. At the same time, particles leaving a subdomain will create gaps in global particle arrays, which are then filled by migrating elements from the end of the global array. This causes particles of the last partition to suddenly become part of a different partition. Since partition-aware algorithms assume a certain data order, these changes need to be cleaned up by sorting data and reestablishing partition-based particle ordering.

Triggering a full repartitioning during all of these events would be too time-consuming. To avoid the computational cost of continuously repartitioning all particles, full repartitioning only happens at a certain frequency. In the meantime, new particles are assigned to existing partitions. Afterwards, the particle arrays are kept in order by using our adapted version of spatial sorting. Note that, while this strategy is significantly faster than full repartitioning, the continuous sorting of particle data adds an overhead to this partitioning approach.

2.4. OpenMP parallelization of particle–wall interactions

Similar to the particle–particle computations described in the previous sections, our hybrid parallelization also adds a second level of parallelization to particle–wall interactions. The existing MPI parallelization of mesh walls first distributes mesh triangles based on the domain decomposition of the simulation. Each MPI process operates only on triangles that are inside of its subdomain. Mesh movement leads to triangle migration, which is implemented using halo regions. This is extended with a second-level parallelization using OpenMP to distribute the remaining collision workload among multiple threads.

Particles may collide with more than one triangle of a mesh at any given time, which requires force contributions stored on a per-particle basis to be applied in a thread-safe manner. To avoid use of locks and critical sections, particles are again organized into partitions. Each partition is then processed only by a single thread.

While the procedure of creating partitions using Zoltan RCB and PSS described above improves load balancing for particle–particle contacts, particle–wall interactions require a different approach. Partitions obtained by RCB distribute the workload based on particle positions and the number of neighboring particles. Using the same partitions for particle–wall computations is not desirable because it does not consider the number of wall contacts, which is proportional to the wall workload. If wall contacts are considered during RCB, the resulting partitions produce a less efficient spatial sorting of particles, and the workload during particle–particle computations becomes less balanced. Partitions used by particle–particle and particle–wall computations are therefore decoupled.

Wall computations do not need to operate on all particles inside a domain. Collisions can be limited to particles which are in the proximity of walls. This is achieved by utilizing existing particle binning, and building triangle–particle neighbor lists. Due to this work reduction, mesh wall computations are split into two steps that must be parallelized to obtain a scalable OpenMP implementation:

- Periodically build triangle–particle neighbor lists which remain valid for multiple force computations.
- For each wall triangle, check all particles in its neighbor list for collisions and compute forces based on granular force kernels when necessary.

The overall force loop used by mesh walls is illustrated in Listing 2.

```
#pragma omp parallel
for(each mesh wall)
  for(each triangle in mesh wall)
    for(each particle neighbor of triangle)
      if(particle in partition of current thread) {
        // compute force of triangle-particle collision
      }
```

The computational effort of building mesh neighbor lists depends on the particle distribution around its triangles. Triangles located in dense particle areas require more checking than triangles in dilute or empty areas of a simulation domain. Load balancing of these triangle neighbor checks is therefore critical in order for all threads to be used efficiently. OpenMP work-sharing constructs could be used for this in combination with a dynamic scheduler or the OpenMP tasking model. However, to avoid non-determinism and overheads

introduced by these OpenMP features, a simple and deterministic solution was implemented. A list of triangle indices is sorted periodically according to the previous number of neighbors of each triangle and then assigned to threads in a round-robin fashion. This static thread schedule is then used during neighbor list building, and the periodic sorting ensures that collision checks remain balanced across all threads.

A similar approach is taken to generate particle partitions used by mesh walls. During neighbor list building, an additional list of particle indices is kept that contains all particles with at least one triangle contact partner. This list is sorted by the number of contact partners, and each particle is then assigned to a thread in a round-robin fashion.

All of these helper structures increase the memory footprint of the application. For example, constant time partition look-ups are enabled by storing a partition/thread ID for each particle per mesh. However, the added benefit of better load balancing outweighs the cost of additional memory usage.

3. Case studies

To investigate the performance and behavior of our hybrid implementation, the following three test cases were studied:

Box-in-a-box: Our first test case illustrates the load-imbalance issue of MPI decompositions. It uses a cubic domain ($50 \times 50 \times 50$ cm) with planar walls. A static cube mesh ($30 \times 30 \times 30$ cm) is placed inside at the center of the domain. 50,000 particles with a diameter of 5 mm are then inserted at the top of the domain. These particles hit the cube and pour over the top edges. Fig. 5a shows a screenshot of this simulation at halftime.

The reasoning behind this test case is that many collisions occur at the top face of the cube, making it one of the hot zones during computation. Since geometric domain decomposition generates a grid of processors, finding a good decomposition of this problem is hard. Many decompositions lead to MPI processes being forced to work inside the cube mesh, which results in no work being assigned to these processors.

Work is also needed along the outer border of the domain and at the bottom, where particles hit the floor and move back towards the center of the domain.

Silo discharge: This test case is a benchmark of a real-world example. 1.5 million particles with a diameter of 1.4 mm are poured into a silo of 40 cm height. The top half of the silo has a diameter of 25 cm, while the lower conical half narrows to a 4 cm diameter. Fig. 5b shows a cross-section of this silo at the beginning of the simulation. After some settling time, the notch at the bottom is opened, letting all particles escape. This test case has previously been used to illustrate the benefit of MPI load balancing along the z direction. The workload moves from the top portion of the domain downwards. Since the lower part of the silo is conical, the workload is focused on the center region of the lower domain.

Mixing process: Our final test case simulates a common process found in industry. Granular materials are filled into a mixer of a given geometry. Rotating blades then stir the composition. LIGGGHTS allows simulation of these processes as it supports moving STL meshes. In this example, the mixer consists of multiple blades rotating at a constant speed. The cylindrical mixer used has a length of 4.7 m and a diameter of 1 m. This huge geometry is filled with 770,000 particles with diameters between 9 and 11 mm. Multiple blades rotate around the x-axis inside and stir the particle mixture. The entire test case consists of 12 meshes with a total of 35,354 triangles. Additionally, there are now static and rotating meshes. Fig. 5c illustrates this setup.

Simulation parameters of each test case are summarized in the Appendix in Tables A.1, A.2 and A.3.

4. Results and discussion

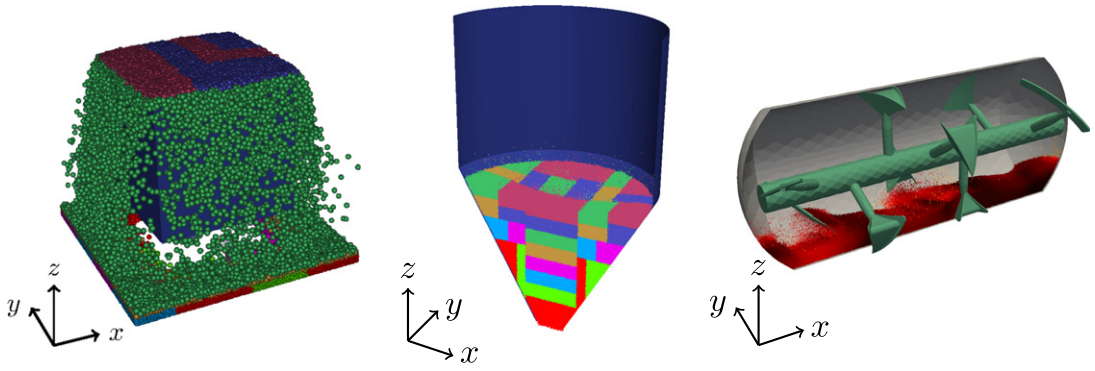
This section presents the test case results obtained through extensive automated testing of different decompositions and configurations. The parameter space for MPI decompositions and MPI load-balancing settings was explored. This was necessary because choosing an unfavorable MPI decomposition can easily lead to runtimes that are 2–4× slower than optimal. In each case, the result with the best runtime was chosen for a given number of cores. All of these tests were performed on AMD cluster blades, each having two sockets equipped with 16-core AMD Opteron 6272 CPUs. Runs up to 32 cores were executed on a single blade, while 64 core runs used two and 128 core runs four blades connected through Infiniband QDR (40Gb/s).

Throughout the following text, we refer to the scalability and timing of code sections of the LIGGGHTS code. These statistics are produced by LIGGGHTS itself and include:

Pair time (T_{pair}): Time spent inside of granular particle–particle force computation kernels.

Comm time (T_{comm}): Time spent in the explicit MPI communication portion of the integration loop.

Neigh time (T_{neigh}): Time spent building neighbor lists for particles.



(a) Box-in-a-box (b) Silo discharge (c) Mixing process

Fig. 5. Case studies (a) Box-in-a-box example; particles are dropped onto a cube, causing them to flow around it. The domain center does not require any computations and causes load imbalance. (b) Silo discharge; 1.5 million particles filled into a silo are released. (c) Mixing process; 770,000 particles are stirred by a rotating mesh geometry.

Modify time (T_{modify}): Time spent in parts of the code which manipulate the state of particles, meshes, and other computations such as gravity, integration, and particle–wall interactions.
Output time (T_{output}): Time spent dumping data to disk. We neglect these results in the following sections.
Other time (T_{other}): All remaining time between the end of a time step and the beginning of a new one.

Note that the times measured by LAMMPS/LIGGGHTS are averaged over all MPI processes. This is done because code regions other than MPI communication measured in T_{comm} are not synchronized by barriers, but truly run in parallel. Each MPI process measures the time in each code section independently, and the resulting final time of each code section is then averaged over all processors at the end of the simulation. This *hides* the load imbalance from timings of code sections such as T_{pair} and T_{modify}, but the imbalance becomes visible in T_{other}. Since all processors have to synchronize after each time step, faster processors must wait until all processors have completed. This waiting or synchronization time between time steps is accumulated in T_{other}. It is the result of the total loop time minus the total average time spent in all code sections:

$$T_{other} = T_{total} - \left(T_{pair} + T_{comm} + T_{neigh} + T_{modify} + T_{output}\right)$$

If all processors spend an equal amount of time in computation, the waiting time in T_{other} will be minimal and only constitute a minimal percentage of the total runtime. As T_{other} increases, more processors will be idle and load imbalance will grow. We therefore define load imbalance as follows:

$$\text{load imbalance in } \% = \frac{T_{other}}{T_{total}} \cdot 100$$

Note that for executions of the hybrid parallelization, T_{other} also contains the overhead introduced by partitioning. Using this definition, we can compare the load imbalances of MPI and the hybrid parallelization, and determine whether the added overhead of partitioning pays off.

4.1. Box-in-a-box

The first test case contained only 50,000 particles and was simulated on up to four 32-core blades with a time step of 10 μs. In serial mode, a simulation of 50,000 time steps took about 29 min to complete. Fig. 6 shows the runtimes of MPI-only, OpenMP-only and the hybrid MPI/OpenMP implementations utilizing all 32, 64 and 128 cores. Table A.4 in the appendix lists which decompositions were used. On 32 cores, an OpenMP-only simulation needed about 4 min 18 s for completion. The fastest MPI-only simulation on 64 cores needed 3 min and 11 s. Beyond 64 cores, the MPI-only performance deteriorated. Our hybrid parallelization, on the other hand, continued to scale and finished within 1 min and 13 s on 128 cores.

All runtimes in Fig. 6 are divided into categories, showing the average amount of time spent in each phase of the integration loop. Note again that, while the total runtimes shown are absolute times, the timings of the individual categories are results averaged over all MPI processes (except "Other time").

Most remarkable in Fig. 6 is the poor performance of MPI-only at 128 cores. To emphasize that this is not caused by poor choice of decomposition, runtimes of other MPI decompositions are shown in Fig. 7. A similarly large variation of MPI-only runtimes was observed for 64 cores, but in this case there was a decomposition that outperformed the 32-core simulation.

Figs. 6 and 7 illustrate that significant time is spent synchronizing and waiting. While T_{pair} decreases linearly between 32, 64 and 128 cores, the long timespan spent outside computation in T_{other} suggests significant load imbalance. Detailed timings further showed that T_{pair}

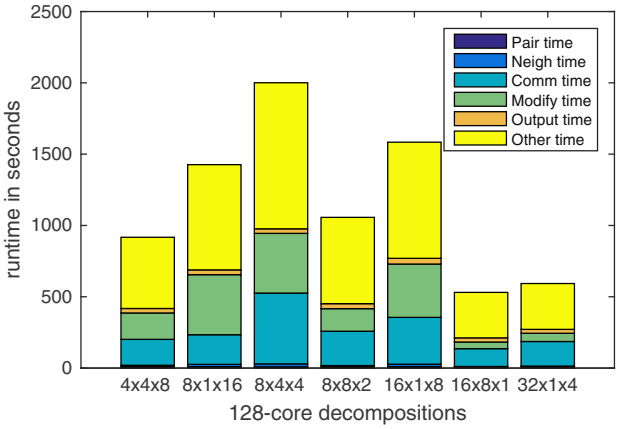


Fig. 7. Runtimes of MPI-only simulations using different 128-core decompositions. All decompositions adjusted domain boundaries dynamically at runtime.

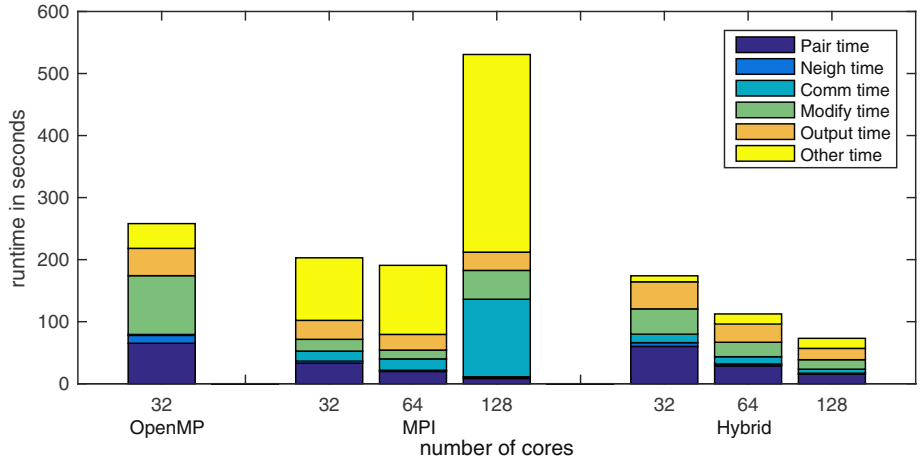


Fig. 6. OpenMP, MPI, MPI/OpenMP hybrid runs of Box-in-a-box test case on 32, 64 and 128 cores. The OpenMP-only run suffers from limited memory bandwidth in memory-bound algorithms inside the Modify section of the code. MPI-only has short average runtimes for each section, but long periods outside computation, which suggests a high load imbalance. Hybrid timings are on average slightly longer, but due to better balancing, processes have shorter waiting times. At 128 cores, MPI domain decomposition no longer scales in this test case.

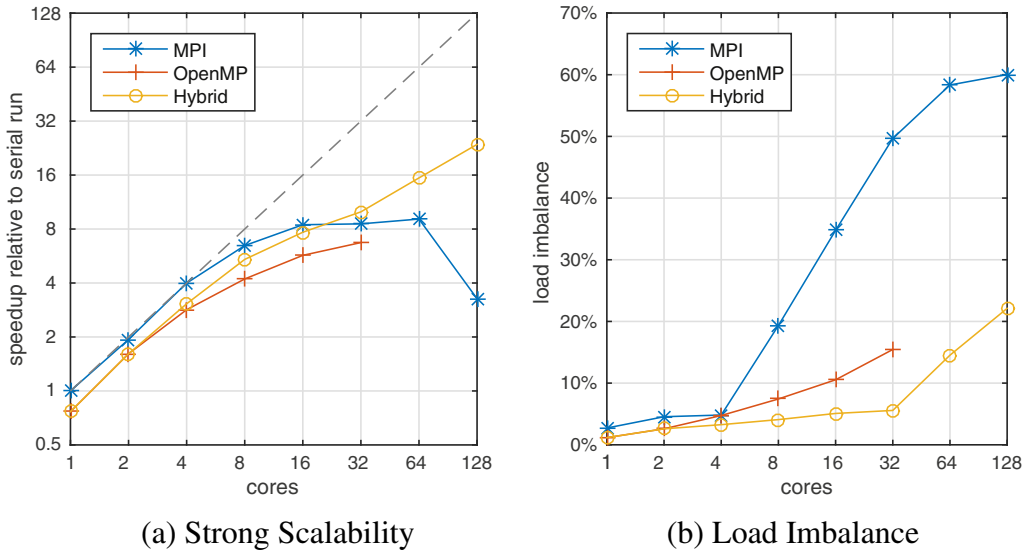


Fig. 8. Strong scalability and load-imbalance of the Box-in-a-box test case with 50,000 particles. The hybrid parallelization benefits from additional memory bandwidth compared to OpenMP-only. It outperforms MPI when the compute node is fully utilized. The main reason is the increasing amount of load-imbalance in MPI for growing core counts.

is not uniform across MPI processes. Some processes spend twice the amount of time in pairwise force computations than others. On average, the time spent in pairwise force computations was shorter in MPI runs than in both OpenMP-only and the hybrid. However, because of the imbalance, many MPI processes were idle at the end of each time step.

Detailed analysis revealed what happened during 128-core runs: The MPI dynamic domain decomposition constantly tried to shift domain boundaries because of the (intentionally) unfavorable geometric setup of this case. Each process was supposed to be assigned about 400 particles to compute. However, since this was not possible due to geometric constraints, the MPI performance of LIGGGHTS no longer scaled. Off-node MPI traffic due to data migration and increased synchronization times added up and became more dominant with increasing number of blades.

Our hybrid implementation performed better at distributing the workload in this situation. It allowed simpler MPI domain decompositions to be used. Our selection was guided by looking at the architecture of the compute nodes used. Each blade had two sockets, each of which contained two teams of 8 cores operating on their own L3 cache. By assigning 4 subdomains to each 32-core blade and binding each MPI process to one of these 8-core teams, we reduced the amount of communication between sockets and through memory on each blade. The hybrid parallelization therefore operated on 4, 8, or 16 teams of 8 cores. Each MPI process then utilized the full 8 cores of each team by spawning 8 threads and applying thread-pinning.

Fig. 8a shows the scalability of all three implementations for this test case. All speed increases were measured relative to a serial run, which is equivalent to an MPI-only simulation with a $1 \times 1 \times 1$ decomposition. While MPI-only scaled well for lower core counts, finding good decompositions with 8 or more processes becomes increasingly difficult. Because of the large empty region in the center of the domain, the dynamic MPI decomposition algorithm struggled to keep all subdomains busy with work. Fig. 8b illustrates how load-imbalance becomes more of a problem as the core count increases. While this is less of an issue for the hybrid parallelization, we observed growing imbalance in Fig. 8b. This is due to the increasing amount of work needed for partitioning with higher core counts.

OpenMP-only runs did not reach the same speedups as their MPI counterparts. Reasons are the added overhead introduced by the OpenMP implementation and the memory bandwidth limitations caused by multiple threads fighting over global memory.

By default, Linux – the operating system in all of these tests – uses a memory policy called first touch. This means that memory is mapped to physical memory not during allocation, but during first access of the first word. Since any global array of data is allocated and initialized by the main thread, it is mapped to the physical memory closest to that processor. This leads to the following bottleneck: memory allocated by threads on one socket cannot be accessed directly by threads on another socket. Any access must pass through the interconnection between CPUs. This means that all memory accesses of threads on a second socket will pass through the processor owning the memory [25].

MPI avoids this problem because each process works on its own smaller memory regions. These automatically map to the closest physical memory and thereby maximize memory bandwidth usage. The hybrid implementation allows us to combine the two approaches and improves performance by using multiple MPI processes working on different memory regions.

In summary, both MPI and OpenMP implementations have very similar speedup characteristics between 1 and 8 cores, with a clear advantage for MPI. However, as the core count increases further the MPI performance becomes limited by load imbalance. The hybrid implementation outperforms the MPI at 32 cores. Due to automatic load balancing and simpler MPI decompositions, the hybrid continues to scale by a factor of 1.5 beyond 32 cores.

4.2. Silo discharge

Our second test case was a larger real-world simulation with about 1.5 million particles. Because of their high number, particle–particle interactions accounted for the majority of simulation time. A time step of 0.5 μs was used.

Serial runs of 100,000 time steps took 2 days and 16 h on one of our cluster blades. By fully utilizing all 32 cores of a single blade, MPI reduced the total runtime to 3 h 41 min. Our hybrid completed in 2 h 56 min. Similarly, when two blades or 64 cores were fully utilized, MPI finished within 2 h 6 min, while our hybrid only took 1 h and 41 min. In both cases, the speed increase achieved by our hybrid was about 20%. Increasing the number of cores further broadened the gap between the two implementations. On 128 cores, MPI completed in 1 h and 49 min. Our hybrid used a simpler decomposition and finished within 59 min, which constitutes a 44% improvement. Table A.5 in the appendix summarizes

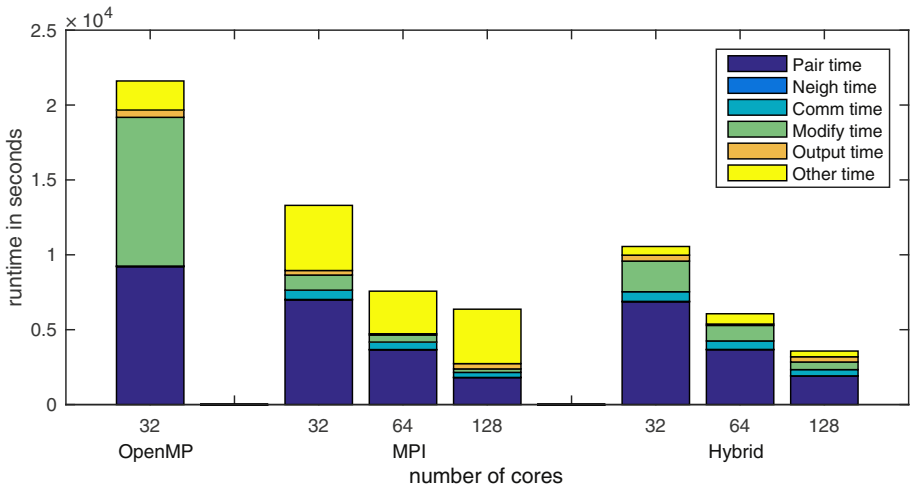


Fig. 9. Silo discharge results of running MPI, OpenMP and hybrid implementations on 32, 64, and 128 cores. OpenMP- suffers only from limited bandwidth in memory-bound algorithms (Modify timing). Load imbalance of the MPI-only runs, visible as long waiting times, becomes a dominant factor. The hybrid parallelization shows continuous scaling and significantly less load imbalance.

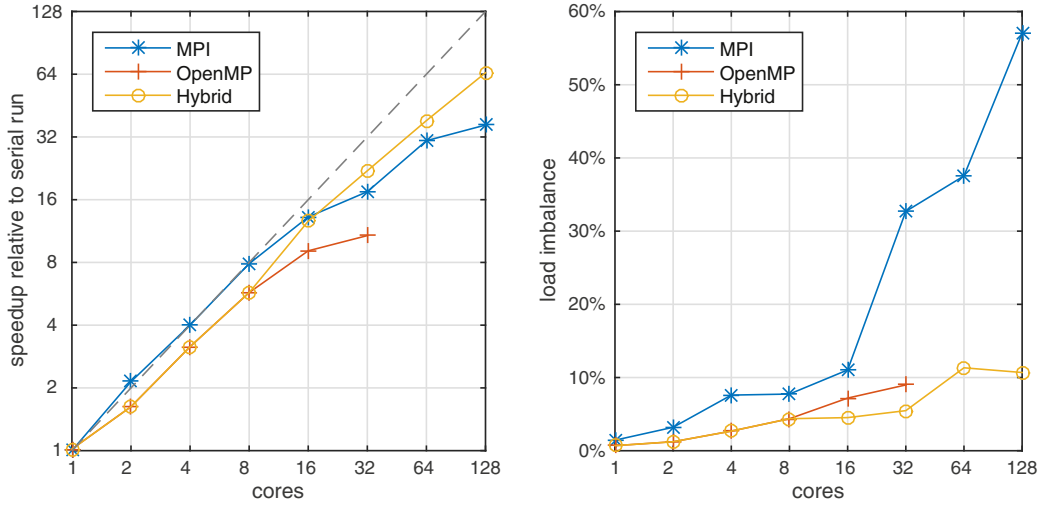
these results and includes further information, such as the decompositions used.

Fig. 9 shows a detailed comparison of the OpenMP, MPI and Hybrid runtimes. Unlike in the previous case, MPI-only runs continued to scale beyond 32 cores, but longer timespans were, again, spent outside computation. OpenMP and especially the Hybrid runs exhibited much less load imbalance. The OpenMP run was added here to show how severe the memory bandwidth limitation is, particularly in larger test cases. Memory-bound algorithms such as integration, which are part of the Modify code section, slow down the simulation significantly.

The scalability of this test case is shown in Fig. 10a. Again, OpenMP and the hybrid did not reach the speed increases of MPI with core counts up to 8. OpenMP saturated at that point, while the hybrid performed similarly at 16 cores and outperformed MPI at 32 cores. Both MPI and the Hybrid then gained a factor of 2× by adding a second 32-core blade. Similar to Fig. 8b of the first example, Fig. 10b shows how MPI struggles with load imbalance at higher core counts. In this case, load imbalance was caused by the conical geometry of the silo, which does not map well to the Cartesian grid of subdomains introduced by the MPI decomposition.

4.3. Mixing process

Our final test case simulated a mixing process using a large mesh geometry stirring about 770,000 particles. With a time step of 10 μs, this benchmark took about 15 h 38 min to complete the simulation of 50,000 time steps in a serial run. Our MPI implementation completed this benchmark after 47 min on 32 cores. With another blade added and running on 64 cores, it completed within 40 min. With 128 cores, the MPI implementation reduced simulation time to 35 min. Our hybrid, however, needed 52 min to complete on 32 cores, and 34 min on 64 cores. 20 min was needed with 128 cores, which is 42% faster than MPI-only using the same number of cores. The configurations used for these simulations can be found in Table A.6 in the appendix.



(a) Strong Scalability (b) Load Imbalance

Fig. 10. Strong scalability and load imbalance of the Silo discharge example with 1.5 million particles; OpenMP and the hybrid yield the same results in the 1-8 core cases. Starting with 16 cores, the hybrid also uses MPI processes, resulting in speed increases that are equal to, or better than, those of MPI-only. Increasing load imbalance limits MPI-only scalability.

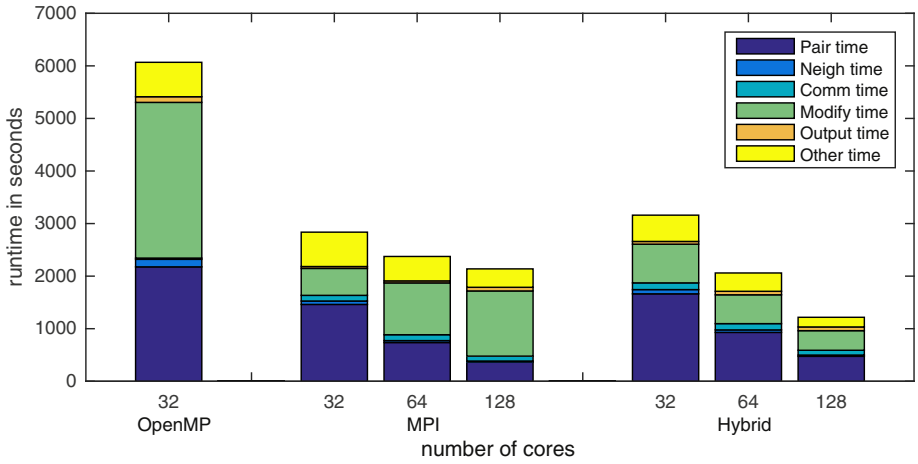


Fig. 11. Mixing process: results of running MPI, OpenMP and hybrid implementations on 32, 64, and 128 cores. Complex MPI decompositions lead to increased MPI traffic inside the mesh implementation. Mesh communication is a major part of Modify timing. The hybrid might be slightly slower at first, but exhibits continuous scaling, because it can use simpler MPI decompositions.

Fig. 11 illustrates these results in detail. It also includes an OpenMP run with 32 cores, which needs twice as much time, again showcasing that OpenMP alone cannot compete.

Unlike in the previous two examples, load imbalance was not the dominant factor in this simulation. Fig. 12a shows that, overall, the scalability is similar to that in the previous examples, but the hybrid was only able to outperform the MPI after adding more cluster blades. Fig. 12b shows that the load imbalance was between 15% and 25%. By our definition, both codes have similar load imbalance at 128 cores, but due to increased MPI traffic caused by the mesh implementation, the time spent in the Modify code section in an MPI-only run was about 3 times longer.

The major difference between this test case and the previous ones is that mesh computations and wall collisions now make up a significant amount of total compute time. These meshes consist of several thousand triangles. Mesh rotation increases MPI traffic between compute nodes due to triangle data migration.

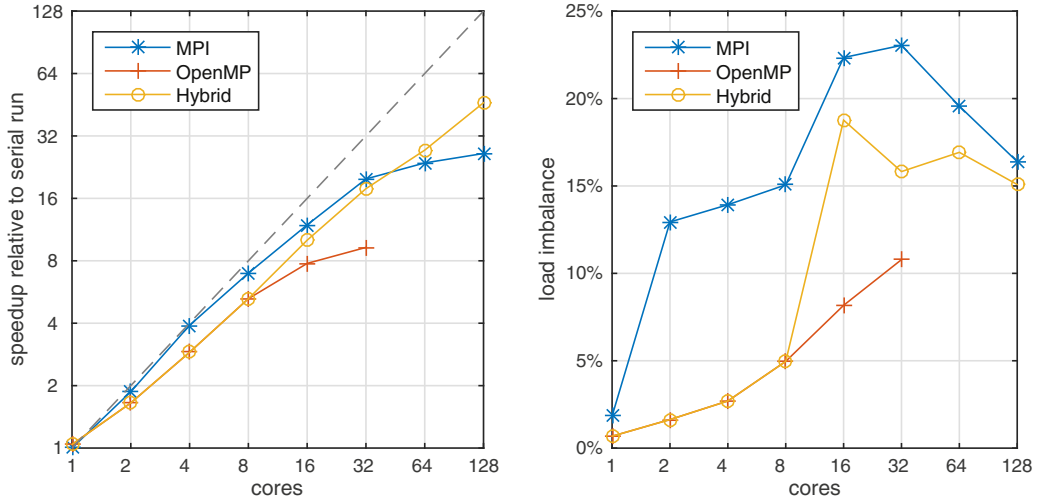
MPI decomposition of meshes also complicates load balancing because it adds an additional dimension to the imbalance: mesh imbalance. Geometric decomposition of meshes is very sensitive to where in space the cuts are placed. If an unsuitable load imbalance metric is used, one processor can be tasked with processing a significantly greater number of triangles than others. Additional computational costs, such as rotating meshes during each time step, make balancing even more important.

Because of the mesh rotation around the x-axis, choosing a domain decomposition in this example requires a tradeoff between minimizing load imbalance across MPI processes and minimizing MPI communication due to migrating mesh triangles. These two extremes are illustrated by the 64-core examples in Fig. 13, which shows the amount of time spent in mesh communication compared to the time spent being idle (Other time).

The hybrid parallelization was used to combine the best of both worlds: full utilization of all compute resources, minimization of load imbalance due to dynamic load balancing, and minimization of mesh communication by choosing domain cuts which reduce triangle migration.

The overall result was that load imbalance at the particle level became less relevant in this test case. For the hybrid, the time spent outside computation was dominated by continuous partition updates.

Well-balanced MPI runs are hard to beat as long as the MPI communication can be kept to a minimum. However, in this type of simulation there is a point beyond which more MPI subdomains lead to an increased amount of MPI traffic due to the mesh



(a) Strong Scalability (b) Load Imbalance

Fig. 12. Strong scalability and load imbalance of the mixing process test case with 770,000 particles and 35,000 mesh triangles; OpenMP and the hybrid yield the same results in the 1-8 core runs. While the hybrid parallelization shows continuous scaling, the MPI-only runs suffer from increased communication overhead. Load imbalance is not as relevant in this case.

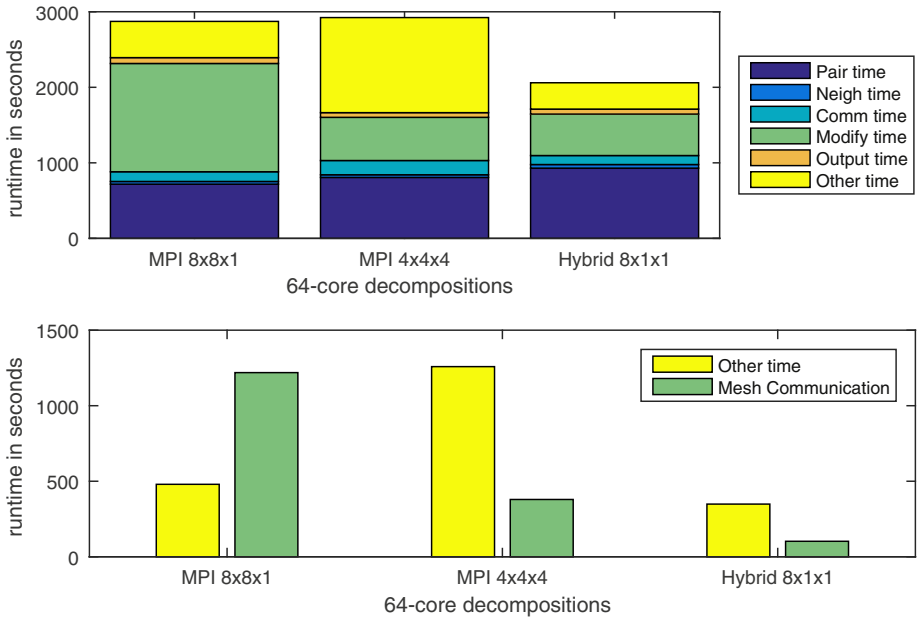


Fig. 13. Influence of different domain decompositions on mesh communication time (part of Modify time) and time spent outside computation (Other time). While the 8x8x1 de-composition reduced load imbalance, almost 50% of the total runtime was spent in mesh communication due to triangles migrating between subdomains in the y direction. A 4x4x4 decomposition improved mesh communication, but led to more load imbalance. The hybrid parallelization used a simple decomposition along the x-axis, reduced the amount of mesh migration, fully utilized all cores (8 threads per MPI process) and minimized load imbalance.

implementation. By keeping the number of MPI subdomains low, the hybrid eventually remains scalable, unlike MPI-only simulations, which stall or deteriorate because of communication overheads. The increase in MPI communication can be illustrated by the number of ghost particles and triangles in each simulation. Fig. 14 shows that the hybrid parallelization uses fewer ghosts while still allowing full utilization of all computational resources.

5. Conclusions

LIGGGHTS is a very scalable MPI code thanks to its rich heritage from LAMMPS and the continued collaboration between the two open-source projects. Implementing the hybrid parallelization showed that, although implementing threaded code is simpler to understand, one must go to great lengths to achieve scalable performance, and it is difficult to match a well-optimized MPI implementation.

While the merits of better load balancing are unquestionable, the main challenge lies in reaching a point where reaping these benefits becomes possible.

For low core counts, our current implementation of our MPI/OpenMP hybrid parallelization is less performant than existing code. The MPI implementation is very well suited to these cases, as long as dynamic MPI load balancing is used correctly. However, as the number of cores and the number of compute nodes increase, load imbalance and rising MPI traffic become a greater problem. In our test cases, the hybrid approach achieved speed increases of

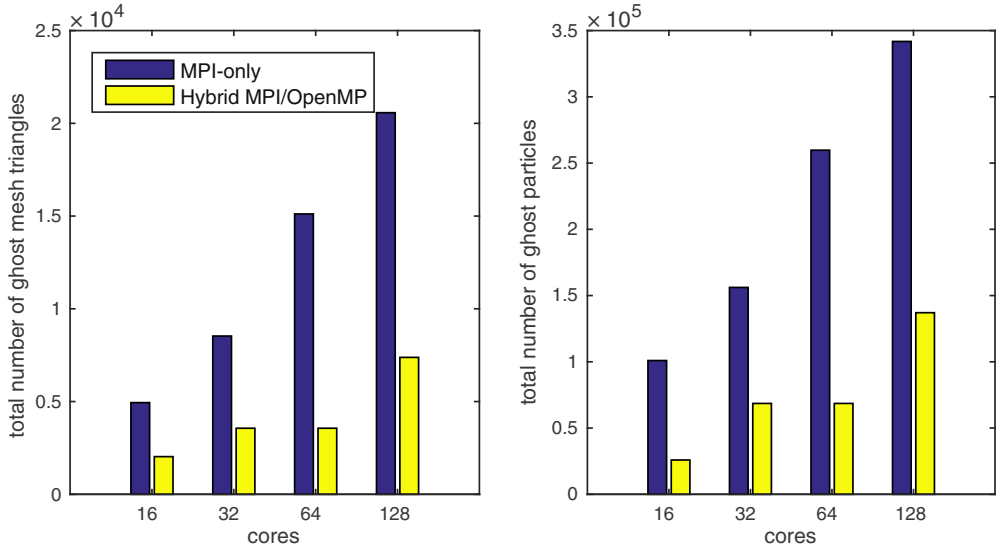


Fig. 14. Increase in MPI communication observed by visualizing the total number of ghosts in the simulation. Due to simpler decompositions, the hybrid parallelization runs introduced fewer cuts and therefore fewer halo-regions. Note that between 32- and 64-core runs the do-main decomposition used by the hybrid did not change, only the number of threads increased.

up to 44% compared to all-MPI parallelization with the same number of processor cores. This is due to improved load balancing with fewer domains in the domain decomposition of the MPI parallelization. The hybrid parallelization allows users to pick a coarse MPI decomposition which maps to the hardware used and to employ OpenMP to utilize all cores available.

The good MPI performance graphs presented in this paper were the result of extensive testing and variation. Picking an adequate MPI decomposition and load-balancing axes requires much insight and practice. Mistakes are quickly punished by extreme runtimes as illustrated in the first test case. The dynamic load-balancing of our hybrid parallelization provides a remedy by allowing simpler MPI decompositions to be used and offers consistent scaling without major pitfalls.

Acknowledgments

We thank the LAMMPS community, especially Steve Plimpton (Sandia National Laboratories), for their continued work. Thanks also to Josef Kerbl (DCS Computing) for providing the initial version of the Mixing process test case. We also thank the developers of Zoltan for their very useful library. This work was supported by the Christian Doppler Forschungsgesellschaft, Austria.

Appendix A. Simulation configurations & decompositions

Table A.1
Simulation parameters of box-in-a-box example.

Model	Hooke
Young's modulus	5×10^6 N/m^2
Poisson's ratio	0.45
Coefficient of restitution	0.3
Coefficient of friction	0.05
Characteristic impact velocity	2 m/s
Time step	10^{-5} s
Particle density	2500 kg/m^3
Particle diameter	5 mm
Number of particles	50,000
Duration	50,000 steps

Table A.2
Simulation parameters of silo discharge example.

Model	Hertz with tangential history
Young's modulus	2.5×10^7 N/m^2
Poisson's ratio	0.25
coefficient of restitution	0.5
Coefficient of friction (particle–particle)	0.2
Coefficient of friction (particle–wall)	0.175
Time step	5×10^{-7} s
Particle density	1000 kg/m^3
Particle diameter	1.4 mm
Number of particles	1,500,000
Duration	100,000 steps

Table A.3
Simulation parameters of mixing process example.

Model	Hertz with tangential history
Young's Modulus	5×10^6 N/m^2
Poisson's ratio	0.45
Coefficient of restitution	0.9
Coefficient of friction	0.05
Time step	10^{-5} s
Particle density	2500 kg/m^3
Particle diameters	9, 10, and 11 mm
Particle distribution	Uniform
Number of particles	772,326
Duration	50,000 steps
Rotation speed	6 rpm

Table A.4
Configurations used to simulate box-in-a-box example.

Cores	Name	Decomposition	Threads	MPI load-balancing	Runtime
1	Serial	$1 \times 1 \times 1$	1	–	28 m 55 s
32	MPI	$4 \times 4 \times 2$	1	z	3 m 23 s
32	MPI + OpenMP	$2 \times 2 \times 1$	8	–	2 m 54 s
64	MPI	$4 \times 2 \times 8$	1	xyz	3 m 11 s
64	MPI + OpenMP	$4 \times 2 \times 1$	8	xy	1 m 52 s
128	MPI	$16 \times 8 \times 1$	1	xy	8 m 51 s
128	MPI + OpenMP	$4 \times 4 \times 1$	8	xy	1 m 13 s

Table A.5
Configurations used to simulate silo discharge example.

Cores	Name	Decomposition	Threads	MPI load-balancing	Runtime
1	Serial	$1 \times 1 \times 1$	1	–	2 days 16 h
32	MPI	$4 \times 4 \times 2$	1	xyz	3 h 41 m
32	MPI + OpenMP	$2 \times 2 \times 1$	8	xy	2 h 56 m
64	MPI	$4 \times 2 \times 8$	1	xyz	2 h 06 m
64	MPI + OpenMP	$2 \times 2 \times 2$	8	z	1 h 41 m
128	MPI	$4 \times 4 \times 8$	1	xyz	1 h 49 m
128	MPI + OpenMP	$4 \times 4 \times 1$	8	xy	59 m

Table A.6
Configurations used to simulate mixing process example.

Cores	Name	Decomposition	Threads	MPI load-balancing	Runtime
1	Serial	$1 \times 1 \times 1$	1	–	15 h 38 m
32	MPI	$8 \times 4 \times 1$	1	xy	47 m
32	MPI + OpenMP	$8 \times 1 \times 1$	4	x	52 m
64	MPI	$8 \times 8 \times 1$	1	xy	40 m
64	MPI + OpenMP	$8 \times 1 \times 1$	8	x	34 m
128	MPI	$16 \times 8 \times 1$	1	xy	35 m
128	MPI + OpenMP	$16 \times 1 \times 1$	8	x	20 m

Appendix B. Supplementary data

Supplementary data to this article can be found online at http://dx.doi.org/10.1016/j.powtec.2015.03.019.

References

[1] C. Kloss, C. Goniva, A. Hager, S. Amberger, S. Pirker, Models, algorithms and validation for opensource DEM and CFD-DEM Progress in Computational Fluid Dynamics, Int. J. 12 (2) (2012) 140–152.
[2] P.A. Cundall, O.D.L. Strack, A discrete numerical model for granular assemblies, Geotechnique 29 (1979) 47–65 (18).
[3] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1) (1995) 1–19.
[4] Z. Yao, J.-S. Wang, G.-R. Liu, M. Cheng, Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method, Comput. Phys. Commun. 161 (1–2) (2004) 27–35.
[5] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 3.0, Tech. Rep., High Performance Computing Center Stuttgart (HLRS), Knoxville, TN, USA, September 2012.
[6] R. Kacianauskas, A. Maknickas, A. Kaceniauskas, D. Markauskas, R. Balevicius, Parallel discrete element simulation of poly-dispersed granular material, civil-Comp Special IssueAdvances in Engineering Software, 41 (1)2010. 52–63.
[7] P. Gopalakrishnan, D. Tafti, Development of parallel DEM for the open source code MFIX, Powder Technol. 235 (2013) 33–41.
[8] S. Plimpton, LAMMPS Manual, Fix Balance Command, http://lammps.sandia.gov/doc/fix_balance.html, (Online; accessed 9-February-2015).
[9] S. Srinivasan, I. Ashok, H. Jônsson, G. Kalonji, J. Zahorjan, Dynamic-domain-decomposition parallel molecular dynamics, Comput. Phys. Commun. 102 (1–3) (1997) 44–58.

[10] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, D. Gardner, Parallel transient dynamics simulations: algorithms for contact detection and smoothed particle hydrodynamics, J. Parallel Distrib. Comput. 50 (1–2) (1998) 104–122.
[11] M.J. Berger, S.H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, IEEE Trans. Comput. 36 (5) (1987) 570–580.
[12] B. Hendrickson, K. Devine, Dynamic load balancing in computational mechanics, Comput. Methods Appl. Mech. Eng. 184 (2–4) (2000) 485–500.
[13] S. Plimpton, LAMMPS Manual, Balance command, http://lammps.sandia.gov/doc/balance.html, (Online; accessed 9-February-2015).
[14] D. Kafui, S. Johnson, C. Thornton, J. Seville, Parallelization of a Lagrangian–Eulerian DEM/CFD code for application to fluidized beds, Powder Technol. 207 (1–3) (2011) 270–278.
[15] A. Kohlmeyer, LAMMPS Manual, USER-OMP package, http://lammps.sandia.gov/doc/accelerate_omp.html, (Online, accessed 9-February-2015).
[16] R. Berger, A. Kohlmeyer, C. Kloss, Toward Parallelization of LIGGGHTS Granular Force Kernels with OpenMP, 6th International Conference on Discrete Element Methods, School of Mines, Colorado 2013, pp. 181–185.
[17] A. Amritkar, S. Deb, D. Tafti, Efficient parallel CFD-DEM simulations using OpenMP, J. Comput. Phys. 256 (2014) 501–519.
[18] H. Liu, D.K. Tafti, T. Li, Hybrid parallelism in MFIX CFD-DEM using OpenMP, Powder Technol. 259 (2014) 22–29.
[19] D.S. Henty, Performance of Hybrid Message-passing and Shared-memory Parallelism for Discrete Element Modeling, Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC'00, IEEE Computer Society, Washington, DC, USA, 2000.
[20] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes, Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP'09, IEEE Computer Society, Washington, DC, USA 2009, pp. 427–436.
[21] L. Smith, M. Bull, Development of mixed mode MPI/OpenMP applications, Sci. Program. 9 (2,3) (2001) 83–98.
[22] G.M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS'67 (Spring), ACM, New York, NY, USA 1967, pp. 483–485.
[23] E.G. Boman, U.V. Çatalyürek, C. Chevalier, K.D. Devine, The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: partitioning ordering and coloring, Sci. Program. 20 (2) (2012) 129–150.
[24] K. Devine, E. Boman, R. Heapby, B. Hendrickson, C. Vaughan, Zoltan data management service for parallel dynamic applications, Comput. Sci. Eng. 4 (2) (2002) 90–97.
[25] U. Drepper, What every programmer should know about memory, http://people.redhat.com/drepper/cpumemory.pdf2007 (Online; accessed 9-February-2015).