A fast triangle-based discrete element code

Konstantinos Krestenitis Tobias Weinzierl
Tomasz Koziara[*]

August 19, 2016

We should again discuss whether we can get Jon on-board

Abstract

Nothing written yet. Nothing written yet.

1 Introduction

Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum.

What is DEM. Why does DEM matter (application areas)?

Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum.

There are two important shortcomings of many codes: only spheres and too few of them. Furthermore, many codes suffer if spheres are of different orders of magnitudes. Why does it make sense to tackle this?

Major objective/contribution of this paper.

Upscaling a DEM code with respect to particle count and machine size challenges the objective to work with triangulated particles from a vast range of particle sizes. DEM codes spend a majority of their compute time in collision detection. This phase becomes significantly more complicated if we switch from sphere-to-sphere checks to the comparison of billions of triangles. Geometric comparisons suffer from poor SIMDability if realised straightforwardly. Multiple contact points between any pair of particles may exist, and it is impossible to predict statically how this computational workload distributes between ranks if particles are distributed among ranks. The distribution itself is non-trivial if there are particles from a vast range of scales, which in turn again makes the contact point relation more complicated than for particle sets of (roughly) the same size. Finally, the amount of data to be exchanged per particle is higher

Tomek: citation.

[*]School of Engineering and Computing Sciences, Durham University, DH1 3LE Durham, UK, konstantinos.krestenitis@durham.ac.uk

than for spherical data where a centre and a radius describe the object of interest. The computational work thus increases significantly with any increase of particle counts. The scalability of this computational work in turn is not given by construction. The present paper introduces realisation idioms of a triangle-based DEM code that fuses efficiency and scalability on all levels of the machine architecture spanning from vectorisation and memory access characteristics over manycore to inter-node data exchange. It shows that the upcoming massively parallel era will allow us to simulate non-sperical DEM formulations with unpreceded speed.

What do we do and how does it differ/fit to others' work? Shortcomings and limitations of the present approach.

Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum.

Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum. Lorem ipsum.

- Non-spherical particles reduce efficiency significantly [78,104] in Samiei - Analytical shape functions besides spheres [41,56,61,118] in Semiei und er selbst - Generic shape composition with spheres [75] und Samiei selber

The remainder of the paper is organised as follows: We start from a brief sketch the overall DEM simulation with explicit time stepping in Section 2. A multiscale grid meta data structure (Section 3) helps us to reduce the algorithms complexity to linear. Starting from this, the main part of the contribution studies a proper data layout of particle and collision data (Section 4) that is well-suited for vectorisation, allows us to exploit multi- and manycore architectures in various ways (Section 5) as well as classic parallel architectures via MPI and domain decomposition (Section **??**). In Section 6, we use all algorithmic ingredients to run some benchmarks that reveal the potential and impact of a technology supporting non-spherical particles efficiently, before we present performance studies (Section 7). A brief summary and an outlook close the discussion.

2 Algorithm outline

Algorithm 1. Blueprint of explicit DEM code's overall structure.

```
for $t < T$ do
    for i = 0 to N triangles do
        for j = i+1 to N triangles do
            $distance = TTD(i, j)$
            if (distance < margin) AND ParticleID(i) != ParticleID(j) then
                $contact(PID(i)).add(point, normal)$
            end if
        end for
    end for
    for z = 0 to NB particles do
        for k = 0 to contacts(z).size() do
```

```
12:             $force = granular(velocity(z), position(z), contacts(z).getcontact(k))$
13:         end for
14:     end for
15:     $t \leftarrow t + \Delta t$
16: end for
end
```

We focus on DEM with explicit time stepping (Algorithm 1). A straightforward implementation consists of one outer time stepping loop hosting two inner loops. The first inner loop detects contact points between particles' triangles. It thus has to loop over all particles. The second loop runs over these contact points and translates them into forces. While contact points could in principle be translated into forces straightaway, it makes sense to outsource the force computation into a separate algorithm phase. The third loop applies the forces to the particles that are in contact and updates the particles position.

Contact detection. Equipped with a small halo layer of size $\epsilon > 0$ two particles are assumed to contact each other if their distance is smaller than 2ϵ. However, the distance always is positive. Such an approach equals a Minkowsi sum approach [4] where the actual particles are extended by a circle with radius ϵ and may penetrate each other up to a depth of ϵ.

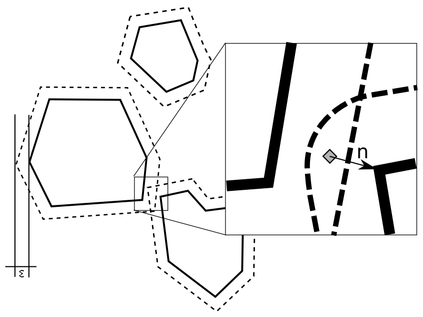


Figure 1: Three particles with their ϵ environment. The particles do not penetrate each other, but two particles plus their ϵ environment penetrate and create one contact point (diamond point) with a normal.

Each contact point is equipped with a normal n pointing from the contact point to the surface of the contacting body's surface. The normal direction depends on whether we make particle A being hit by particle B or the other way round: each normal associated with a particle points along the outer normal of the body. Despite the fact that we employ a rigid body model, multiple contact points between two bodies may exist as particles may be concave and their Minkowsi sum may overlap.

For two given particles A and B with corresponding sets of triangles $\mathbb{T}_A$ and $\mathbb{T}_B$, the contact detection thus reads: Loop over all triangle pairs $t_i \in \mathbb{T}_A, t_j \in \mathbb{T}_B$ and identify those pairs that are closer than 2ϵ. For those close, determine the middle point along the closest distance. The point equals the contact points while the distance vector identifies its normal. Add the contact point to a contact point set $\mathbb{C}_A$ and to the set $\mathbb{C}_B$. The normal in the latter is inverted. Such an algorithm has the complexity $\mathcal{O}(|\mathbb{T}_A| \cdot |\mathbb{T}_B|)$ and makes the overall naive implementation a member of $\mathcal{O}(|\mathbb{T}|^2_{max})$ if $|\mathbb{T}|_{max})$ is the maximum number of triangles per particle.

Force model. We apply the linear spring-dashpot force model from Cundall and Strack [**?**] to study the efficiency of our implementation. Per particle A, it accumulates all contact points in $\mathbb{C}_A$ into one translational and one rotational repulsive force with some damping. We make $\mathbb{C}_A$ subject to a postprocessing which eliminates all collision point duplicates which is all duplicates that are closer than $min(h_{A,min}, h_{B,min})$. It is the smallest face length of the meshes representing A and B. No contact point may be closer than this value. On the preprocessed contact point set $\mathbb{C}_A$ we then determine

$$f_{trans} = \sum_{c \in \mathbb{C}_A} \min \left(0, -k_{repulsive}(\epsilon - |n|) + k_{damp} \frac{(v_B - v_A, n)}{|n|} \right) \frac{n}{|n|} \tag{1}$$

$$f_{rot} = \sum_{c \in \mathbb{C}_A} \tag{2}$$

$k_{repulsive}$ and k_{damp} are material coefficients. We use the normal's norm to determine the model's penetration depth, while the change rate of this depth is derives from the projection of the relative velocity between the two particles onto the normal vector. We use the the particles' velocities v_A and v_B here. The min function ensures that the force always pulls particle away from each other. There is no particle attraction.

Our Newtonian DEM code relies solely on pair-wise interactions so far. With a modification of (2,2), it is however possible to introduce more sophisticated force patterns. We note that the complexity of the force computation depends solely on the size of $\mathbb{C}$ if we anticipate that we have to run over all particles to update their position in space due to momentum anyway. $\mathbb{C}$ typically is very small if particles are of reasonably the same size. If particle sizes vary dramatically, comparably large particles experience contact forces from many small particles. In this case, there are however few large particles compared to the overall number of particles.

3 Grid meta data structures

Various speedup techniques such as linked-cell lists [2] and Verlet lists [5, 2] reduce the quadratic complexity in DEM codes. We propose to rely on a generalised tree-based linked-cell technique that allows us to efficiently treat particles from a vast range of diameters. Three observations support this design decision: First, particles colliding with other particles are close to these particles. It is sufficient to scan a certain environment around each particle for potential collision partners. We thus split up the domain into control volumes. They are cubic as this simplifies the implementation compared to control volumes of more flexible shapes. Second, we may choose these control volumes to be larger than the biggest particle diameter. For a particle held in a particular control volume (cell), it is thus sufficient to check the $3^d - 1$ neighbouring cells whether they host other particles that might collide. d is the spatial dimension. Third, the previous decision is problematic if the particles are of extremely different size. The cell size is determined by the largest particle diameter. If we use a uniform cell size, many unnecessary collision checks are performed for small particles. If we use an adaptive grid, it is tricky to design the grid such that only direct neighbouring cells have to be studied. We thus, third, observe that a cascade of grids might be useful: If we have several grids embedded into each other, we can store each particle in the grid suiting its diameter. Particles of one grid then have to be checked agains particles in their neighbouring cell as well as neighbouring cells on coarser grid resolution levels. There is no need to check a particle of one grid resolution with particles of a finer grid resolution—if a particle A collides with a particle B, particle B also collides with particle A and such relations thus are already detected.

A spacetree is a space-partitioning data structure constructed recursively. The computational domain is embedded into a unit cube. We cut the unit cube into three equidistant pieces along each coordinate axis. This yields 27 new cubes. They are called children of the bounding box cube which is the root. For each of the children, we continue recursively to evaluate the split decision. The decision to cut into three parts results from the fact that we rely on a code base based upon three-partitioning [14]. Bipartitioning, i.e. the classic octree, works as well.

The construction scheme yields a cascade of ragged regular Cartesian grids that are embedded into each other. Each cell besides the root has a unique parent cell. While we could make the cells hold particles, we propose to use a multiscale, vertex-based scheme spanning a dual meta grid [16]. A vertex is unique through its spatial position plus its level. The level is the number of refinement steps required at least to create one of its adjacent cells. Each vertex holds a list of particles. A particles is always stored on the finest grid level where the cells' edge length is still bigger than its diameter. A particle is always associated to the vertex next to its geometric centre, i.e. any vertex has a list of all particles close to it on the same level. Links from the vertices to the particles are realised as pointers. If a particle moves, we have to update the links, but we do not move geometric data in memory.

Grid traversal. With a grid at hand, we may map the algorithmic steps from Algorithm 1 onto a grid traversal. For the traversal, we rely on a combination of a depth-first order with space-filling curve [13, 15]. From a DEM point of view, the exact traversal realisation however is not that relevant as long as the traversal is a real tree traversal, i.e. runs through all levels of the underlying spacetree.

We then write down the algorithm as a set of events, i.e. we specify which operations are performed if a vertex is read for the very first time (`touchVertexFirstTime`), if a cell is entered (`enterCell`), and so forth. Be-

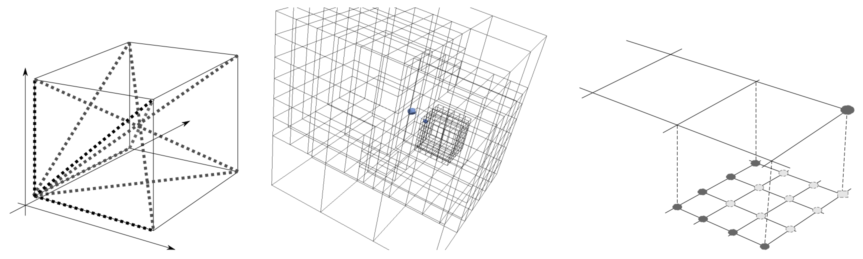


Figure 2: Left: Whenever the grid traversal enters a cell, it checks whether particles assigned to one vertex do collide with particles assigned to another vertex. To avoid redundant collision computations, we check only some vertex pairs (dotted, larger lines). Middle: Two particles approach each other. As they are of different size, they might be held on different spacetree resolution levels. Right: In the adaptive case, particles are dropped from the coarse levels into the fine grid (rectangular marker) if new grid levels are added. The bright round vertices are children of the marked coarse grid vertex. The bright and the dark round markers' vertices together are the descendants of the marked coarse grid vertex.

sides the actual grid hosting the particles, the grid sweeps build and maintain two further sets of collision points (algorithm 2).

Algorithm 2. *A grid-based DEM implementation.*

```
1: function TRAVERSEGRID($\mathcal{C}$)
2:     $\mathcal{C}_{old} \leftarrow \mathcal{C}$
3:     $\mathcal{C} \leftarrow \emptyset$
4:     while traversal continues do
5:         if touchVertexFirstTime then
6:             for all particles p associated to vertex do
7:                 for all contact points $c \in \mathcal{C}_{old}$ associated to p do
8:                     Update $f(p)$ through c
9:                 end for
10:                Update particle incl. its triangles
11:            end for
12:            for all particle pairs $(p_i, p_j)$ associated to vertex do
13:                $\mathcal{C} \leftarrow \mathcal{C} \cup$ FINDCOLLISIONS($p_i, p_j$)
14:            end for
15:        end if
16:        if enterCell then
17:            for all $2^d$ vertices adjacent to cell do
18:                for all particles p associated to vertex do
19:                    if particle should be associated to different vertex then
20:                        Reassign particle
21:                    end if
```

```
22:                 end for
23:             end for
24:             for all $(p_i, p_j)$ associated to different vertices (cmp. figure) do
25:                 $\mathcal{C} \leftarrow \mathcal{C} \cup$ findCollisions$(p_i, p_j)$
26:             end for
27:         end if
28:     end while
29: end function
end
```

Our grid-based realisation is characterised by few properties:

- When we load a vertex for the very first time, we make all particles associated to this vertex move.
- When we load a vertex, we do compare all the particles associated to this vertex with each other and identify collision points. The comparison of particles associated to different vertices is realised when we enter a cell. Here, we do compare the vertex pairs from Figure 2—left bottom front vertex with right bottom front vertex, all diagonal combinations, and so forth—such that we avoid multiple evaluations of vertex pairs. For boundary cells, some special case distinctions yielding additional checks are added. If a cell is traversed, we assume that all its adjacent vertices are available, i.e. have been read before.
- Whenever we run into a cell, we run through all the adjacent vertices and their particles. They all already have an updated position, as any vertex has been loaded before and thus been subject to `touchVertexFirstTime`. If a particle is associated to the 'wrong' vertex as it has moved and should be assigned to another vertex of the cell, we do the reassignment. A particle may move at most one cell of its corresponding level a time.
- The vertex comparisons yield a set of collision points. This set is kept persistent for the subsequent traversal: one time step of the scheme is realised per two grid traversal. The amortised cost however still is one time step per grid traversal. This scheme picks up the idea of pipelining [8]. Collision data is mapped onto the particles when we read a particle for the first time in the subsequent traversal. Immediately after the forces acting on a particle are determined, we do update the particle's property and also update the geometric data due to translation and rotation.

The extension of the present scheme to multiscale trees is detailed below. To make the algorithm correct, each contact point set between two particles does exist twice with inverse normals. Each set furthermore is augmented with a copy of the corresponing partner particles' global data (mass, velocity, momentum). Otherwise, we would have to search for this data and build up global indices, and have to be aware that the particle's data already might have been subject to the next time step.

Observation. The algorithm realises a single-pass policy. Geometric data is written only once per traversal, and is read only when we read in a vertex for the first time and when we run through a cell. The implementation's pressure on the memory subsystem thus is minimalistic as long as the spatial and temporal grid access locality [**?**] is high which yields high cache hit rates. We rely on a space-filling curve to run through the grid [13, 15] and thus guarantee such a two-fold locality.

More sophisticated explicit schemes can be realised with the same single-touch policy if we hold additional data per particle (acceleration, e.g.). Such an approach picks up ideas of pipelining [8].

Regular grid. The spacetree formalism allows us to realise at least three grid variants. A very simple refines all spacetree nodes all the time as long as the resulting cell mesh size is bigger than the largest particle diameter. Such a strategy yields a regular Cartesian grid.

Dynamically adaptive grid. Our dynamically adaptive grid is characterised by two ingredients: mesh refinement control and inter-grid particle treatment. In `touchVertexFirstTime`, our code analyses what the smallest diameter of all the particles held by the vertex is. If this diameter is smaller than 1/3 of the mesh width corresponding to the vertex's level, then the region around the vertex is refined. If we run into a vertex, we check whether there are any spatially coinciding vertices in the tree on a coarser level. If such vertices do exist, we run through all of their particles and move them one level down if the diameter permits. The scheme successively drops the particles down the grid hierarchies.

We define two multiscale relationships between vertices (Figure 2). A vertex a is a child of a vertex b if all adjacent cells of a are children of adjacent cells of b. b is the parent vertex of a. A vertex a is a descendant of b if at least one adjacent cell of a is a child of an adjacent cell of b. b is an ancestor of a. If we delete a vertex a that holds particles, its particles are moved to the next coarser level and assigned there to the nearest parent of a. Each vertex holds a boolean marker that is set $\perp$ before the vertex is read for the first time. If a vertex holds a particle, all the markers of the vertices where it is a descendent from are set to $\top$. If a vertex whose adjacent cells all are refined holds $\perp$ at the end of the multiscale traversal, we coarsen these refined adjacent cells. We rely on a top-down tree traversal. The refinement/coarsening procedure then can be evaluated on-the-fly. It equals an analysed tree grammar [**?**].

For the multiscale contact detection, we extend the list of particles associated to a vertex. There is the list of actually held particles and a list of virtual particles. We run through the grid top down, i.e. a vertex always is read for the first time before any of its descendants, and clear virtual particle list first. Then, we add all particles held in the particle or the virtual particle lists of any ancestor to the local virtual list. If we compare all particles with each other that are held by the same vertex, we do compare the actual particles with all other real particles as well as all virtual particles.

A reluctant adaptive grid. The dynamically adaptive grid refines rather

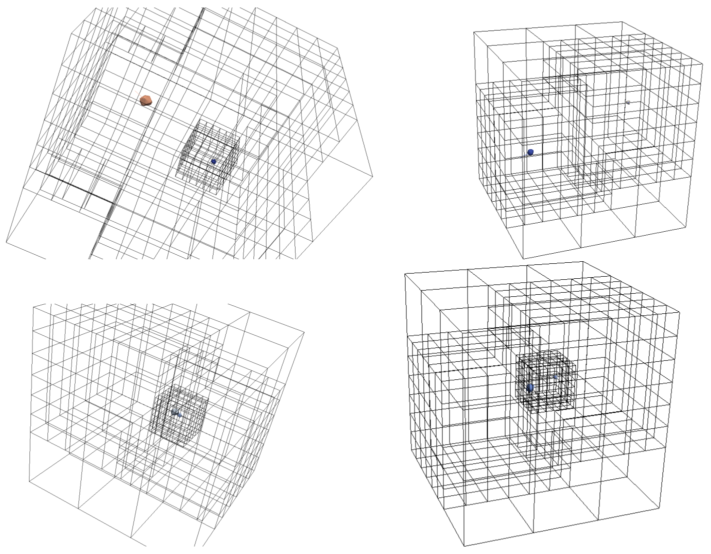


Figure 3: Two particles crash into each other. The adaptive grid refining around each particle while its diameter constrains the mesh size (left column). The reluctant adaptive grid works with a coarser resolution as long as particles are far away from each other (right column). Just before they collide, the grid is refined and particles are dropped down the resolution levels.

aggressive: Particles are always dropped to their corresponing refinement level immediately. Fine grid regions thus follow 'their' particles (Figure 3). This might introduce finer grids than actually required for contact detection which is an overhead. Given the one-cell-per-time-step constraint on the particle velocity, we also restrict the maximum velocity or time step rigorously. For the present work, this does not have a major impact as we apply uniform small time step sizes globally. For schemes with local time stepping, it however is important, besides overhead discussions, to keep the grid as coarse as possible as this facilitates big time step sizes.

Our reluctant adaptive grid works with coarser adaptive grids than the plain variant through two modifications of the refinement procedure: On the one hand, we refine the region around a vertex if the previous criterion holds and the vertex holds at least two particles. If only one particle is holds, we stick locally to the grid no matter what the particular diameter is. Throughout the inter-vertex contact detection in `enterCell`, we further bookkeep the minimum diameter of all the particles involved. If this minimal diameter is smaller than the cell, we do refine this cell, too.

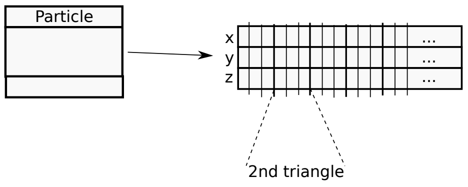


Figure 4: Right: The two layer data layout of our DEM code with an AoS on the particle level but SoA for the vector entries with replicated vector entries.

4 Particle storage and vectorisation

There are three different types of data sets in our minimalistic DEM codes: geometric data, behavioural data and meta data. Behavioural data is collision data. It is modelled as struct with location and a normal vector. Sets $\mathbb{C}$ of collision points are modelled as (dynamic) array of structs (AoS) augmented with a copy of the collision partner as detailed before. Spacetrees are our meta data. The geometry data finally determines how straightforward and fast the collision detection is. Collision detection is the computationally heavy activity in the algorithm. While other DEM discussions speak of a computational phase [11, 12, 9, 7], we prefer the term activity, as the collision detections are split among the grid traversal and interwoven with other activities.

We propose to realise the geometric data in two layers (Figure 4). A hull struct holds all particle properties such as velocities, rotation, mass, geometric centre and mass centre. Vertices refer to these hulls with arrays of structures (AoS). Hulls link with pointers to the actual geometric data. This data is realised as structure of array, i.e. there is a sequence of x-coordinates, a sequence of y-coordinates and a sequence of z-coordinates. These sequences are blown up with redundant data. The first three entries in the x array hold the x coordinates of the three vertices of the first triangle of the particle mesh. The entries four through six hold the coordinates of the second triangle and so forth. The degree of redundancy is determined by the particle mesh.

We accept the increased memory consumption of such a structure but in return are able to avoid any indirect addressing, process all geometry data in the collision checks in a stream-like fashion and can align all vector entries. SoA data are notouriously difficult to handle if subsets of a dataset are to be transferred or data is to be reordered [3]. In our particle handling, a particle is an atomic unity. It is never teared apart or resorted during the simulation run. A particle mesh is topologically invariant.

The remainder of this section discusses a function `findCollisions` that is passed two particle meshes and identifies all contact points. Such an operation runs over all triangles in $\mathbb{T}_A$ and the inner loop running over all triangles in $\mathbb{T}_B$. A sphere-to-sphere pre-check is often added to the computation as a cheap overhead that shortens simulation runtime during sparse-particle scenarios. The center of the sphere is defined to be the center of the non-spherical particle while the radius of the sphere is at least the radius of the particle plus the epsilon margin.

4.1 Brute force geometric comparison

Our default brute force approach implementation runs per triangle pair through all possible geometric configurations (Algorithm 6). Firstly, we determine the distances between each line-segment to line-segment combination (segment-to-segment stage). Second, with all six involved vertices we compute the distance to the other triangle (point-to-triangle stage). This yields fifteen comparisons in total. The distance between two triangles is the minimum over all computed distances. The brute force method is a robust approach, it always yields the correct answer after the evaluation of all steps. A fusion of multiple steps is not possible because all configurations have to be evaluated naively as seen in Appendix:Algorithms 7, 8.

The calculation of distance between line segments [4] in 3D involves extending the line segments until intersection, then the closest point on the two segments is on the boundaries. The segment $S1 = [P0, P1]$ can be formulated as $P(s) = P0 + s(P1 - P0) = P0 + su$ with a constraint, $0 <= s <= 1$. Similarly, $S2 = [Q0, Q1]$ is written as $Q(t) = Q0 + t(Q1 - Q0) = Q0 + tu$ with constraint $0 <= t <= 1$. So, for sC and tC being the closest points on the corresponding extended segments lines L1 and L2, then if both sC and tC are within the boundaries of the segments then the closest points are also the closest points on the respective segments. In the case where sC and tC correspond to points on L1 and L2 outside the range of either segment $S1$ and $S2$, then sC and tC do not also define the closest points on the segments $S1$ and $S2$. So it is necessary to determine points that minimize $w(s, t) = P(s) - Q(t)$ over the ranges of the segments using the corresponding constraints. The problem can be formulated into a minimization problem where the equation w is the same as minimizing $|w|^2 = w \cdot w = (P0 + su - tv) \cdot (Q0 + su - tv)$. The relation of $|w|^2$ define a equation over a (s,t)-plane (See figure 5) with a minimum at $C = (sC, tC)$, it is strictly growing along the (s,t)-plane with starting point from C. The required minimum region is not C but it is located over a subregion G of the (s,t)-plane. Using this method it is possible to perform checks to find the minimum of w(s,t) that correspond to the minimum distance between the segments.

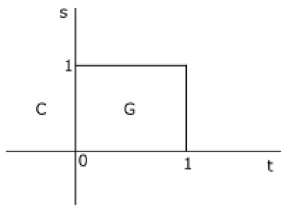


Figure 5: (s,t) parameter space with G boundary box C for global minimum

The next stage of the brute force algorithm requires checking the point-to-triangle distance **??**. Using triangle T barymetic function such that $T(s,t) = B + s \cdot BA + t \cdot DA$ where $B = A$, $BA = B - A$, $DA = D - A$ for $(s,t) \in D = \{(s,t) : s \in [0,1], t \in [0,1], s + t1\}$. The minimum distance is computed by the values $(s,t) \in D$ in distance equation $Q(s,t) = |T(s,t) - P|^2$ where $T(s,t)$ correspond to a point Q. The function can be written as: $Q(s,t) = as^2 + 2bst + ct^2 + 2ds + 2et + f$ where $a = BA \cdot BA$, $b = BA \cdot DA$, $c = D - A \cdot DA$, $d = B - A \cdot (B - P)$, $e = DA \cdot (B - P)$, and $f = (B - P) \cdot (B - P)$. So for function Q, $ac - b2 = (B - A \cdot B - A)(E1 \cdot D) - (B - A \cdot B - A)^2 = |B - A \cdot D - A|^2 > 0$. The positivity is based on the assumption that the two edges $B - A$, $D - A$ are linearly independent. The minimum occurs at an interior point of D where the gradient $\nabla Q = 2(as + bt + d, bs + ct + e) = (0,0)$ or at a point on the boundary of D. The gradient of Q is zero only when $s = (be - cd)/(ac - b^2)$ and $t = (bd - ae)/(ac - b^2)$. If $(s,t) \in D$ (See Figure 6) then minimum of Q is found.

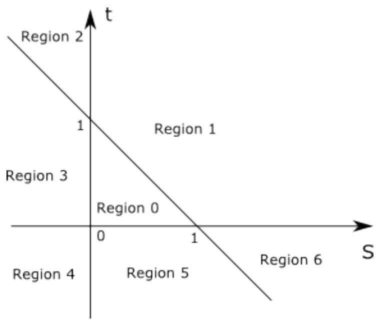


Figure 6: Regions based on the (s,t) parameters plane

On the other hand if the minimum is not in D then it is on the boundary of the triangle. Using the active constraints we restrict the solution so if (s, t) is in region one then the level curves of Q are constant. As the level value V increase from V_{min} as they growing away from (s,t) there is a smallest level value that is tangent to the domain D with $s + t = 1$. So for any level values $V < V_{min}$, the corresponding do not intersect D. However for any portion of D that intersect levels V must be $V > V_{min}$. Therefore in this case point $(s0, t0)$ provides the minimum distance between P and the triangle where $t0$ is known $t0 = 1 - s0$ and $s0$ is the only unknown to be solved. Moreover, when minimization is happening at $\nabla Q(s, 1s) = 0$ then there are three cases where $s > 1$ and s has to be restricted to $s = 1$ and the minimum occurs at $\nabla Q(1,0)$ because of the constraints. Similarly if $s < 0$ then minimum occurs when $\nabla Q(0,1)$ or lastly $s \in [0,1]$. Similarly the same technique for determining whether the minimum occurs at the endpoints or at the interior interval of the corresponding constraints is performed for all regions.

4.2 Penalty-based formalism

The alternative approach is by parameterisation of the triangles and the distance between them is approximated using the Newton method. Let x and y be two points on triangle T_1 and T_2 respectively, points A, B, C are vertices of T_1 while points D, E, F are vertices of T_2, then x and y can be specified using the following equations: $T_1 : x(a,b) = A + (B - A) \cdot a + (C - A) \cdot b$, $T_2 : y(g,d) = D + (E - D) \cdot g + (F - D) \cdot d$ To find the minimum distance line (P, Q) of T_1 and T_2 we minimize $f\left(a,b,c,d\right) = \left\|x\left(a,b\right) - y\left(c,d\right)\right\|^2$. For x and y to be within the feasible area of the two triangles, six inequality constraints are added:$min_{a,b,g,d} f(a,b,g,d) such\ that :\ \{a \geq 0, b \geq 0, a + b \leq 1, d \geq 0, g \geq 0, g + d \leq 1\}$

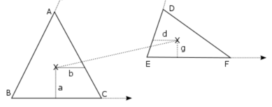


Figure 7: Triangle X:T1 (Vertices A, B, C) with barymetric parameters a,b that point x point on the triangle. Triangle Y:T2 (D,E,F) with g,d parameters. The two barymetric points define the P, Q points of the minimum distance between the two triangles in 3D.

$$P(x) = f(x) + r \sum_{i=1...6} max(0, c(x_i))^2 \tag{3}$$

Where r is the penalty parameter (figure **??**) that augments f. Convergence can be controlled by the r parameter that controls the sharpness of the constraint curve. One aspect that requires care however is the invertibility of the Hessian $\nabla\nabla P$ due to ill conditioning. This illustrates the fact that f has multiple minima and $\nabla\nabla f$ is singular. Consequently, $\nabla\nabla P$ is also singular inside of the feasible region. The cause is due to multiple possible solutions when the orientation of the two triangles is parallel. This is also revealed by the two zero eigenvalues of the Hessian. We use a quasi-Newton approach, where the Hessian is approximated by a perturbation operator $\nabla\nabla P + \epsilon I$. I is an identity matrix and ϵ is suitably small.

Algorithm 3. *Penalty Solver.*

```
1: function PENALTY(A, B, C, D, E, F, rho, tol)
2:     $BA = B - A$; $CA = C - A$; $ED = E - D$; $FD = F - D$;
3:     $hf = [2*BA*BA',\ 2*CA*BA', -2*ED*BA', -2*FD*BA'$;
4:     $2*BA*CA',\ 2*CA*CA', -2*ED*CA', -2*FD*CA'$;
5:     $2*BA*ED', -2*CA*ED',\ 2*ED*ED',\ 2*FD*ED'$;
6:     $2*BA*FD', -2*CA*FD',\ 2*ED*FD',\ 2*FD*FD']$;
7:     $x = [0.33;\ 0.33;\ 0.33;\ 0.33]$;
8:     for i=1:99 do
```

```
 9:     $X \ = \ A + BA{*}x(1) \ + \ CA{*}x(2);$
10:     $Y \ = \ D + ED{*}x(3) \ + \ FD{*}x(4);$
11:     $gf \ = \ [2{*}(X-Y){*}BA';\ 2{*}(X-Y){*}CA';\ -2{*}(X-Y){*}ED';\ -2{*}(X-Y){*}FD'];$
12:     $h \ = \ [-x(1);\ -x(2);\ x(1)+x(2)-1;\ -x(3);\ -x(4);\ x(3)+x(4)-1];$
13:     $dh \ = \ [-1,\ 0,\ 1,\ 0,\ 0,\ 0;\ 0,\ -1,\ 1,\ 0,\ 0,\ 0;$
14:     $0,\ 0,\ 0,\ -1,\ 0,\ 1;\ 0,\ 0,\ 0,\ 0,\ -1,\ 1];$
15:     $mask \ = \ h'$ ¿= $0;$
16:     $dmax \ = \ dh.{*}\ [mask;\ mask;\ mask;\ mask];$
17:     $gra \ = \ gf \ + \ rho * dmax * max(0, h(:));$
18:     $hes \ = \ hf \ + \ rho{*}dmax{*}dmax' \ + \ eye(4,4)/rho^2;$
19:     $dx \ = \ hesngra;$
20:     $DX \ = \ BA{*}dx(1) \ + \ CA{*}dx(2);$
21:     $DY \ = \ ED{*}dx(3) \ + \ FD{*}dx(4);$
22:     $error \ = \ sqrt(DX{*}DX' + DY{*}DY');$
23:     if $error < tol$ then
24:         $BREAK;$
25:     end if
26:     $x \ = \ x \ - \ dx;$
27:   end for
28: end function
end
```

The penalty algorithm as shown in Algorithm 3 accepts vertex coordinates for triangle T1(A, B, C), T2(D, E, F), *rho* for the penalty parameter, and *epsilon* for the perturbation, *tol* is the tolerance error for convergence. At line 23 of Algorithm 3 an initial guess is set to be the center of the triangles, the For loop initiates the Newton solver to find the solution on the X, Y triangle, under constraint c. For each constraint (line 12) the max function is evaluated to detect the active ones. At line 17 and line 18 the gradient and Hessian is evaluated for the Gaussian elimination direct solver (MATLAB backslash) to get *dx* Newton direction

4.3 Hybrid approach

To create a hybrid solver we first assume that on average there are for each triangle pair distance computation only a few iterations that are required to arrive to a solution 8. Based on empirical studies and tuning of the penalty parameter and the regularization variable, the majority of triangle pairs are solved within four iterations as shown in Figure . Secondly we set a user defined tolerance of error to the method to act as the switching point for the falling-back to brute force solver. If the number of fall backs does not overtake the number of penalty-based solutions then the method is a compromise between brute force and penalty both in terms of performance but also in terms of error of solution.

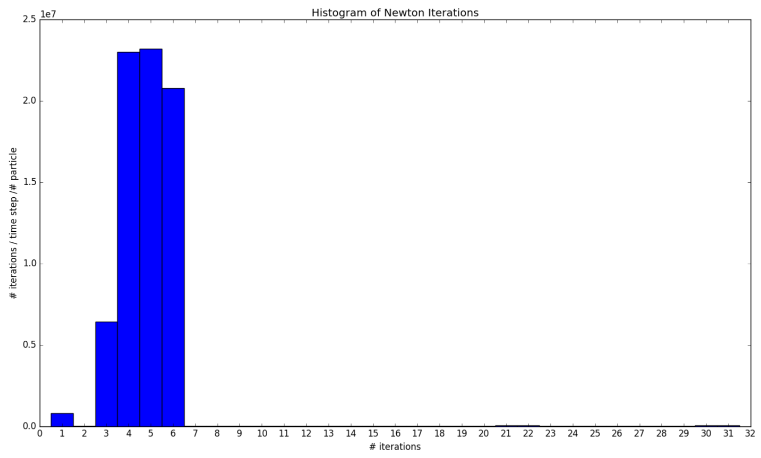


Figure 8: Histogram of number of iterations required by Newton Method for convergence on a sample of twenty four million triangle pair configurations.

There are two variants of hybrid method; the first is the hybrid-on-triangle-pairs and the second one is hybrid-on-triangle-batches. Both variants are developed to exploit the robustness of brute force while keeping the arithmetic intensity of the penalty method. The implementation of both methods takes into account the potential for shared memory scaling as well as data access continuity. In addition SIMD vectorised performance of the underlying methods and memory alignment are critical for the execution of both methods.

The first variant is hybrid-on-triangle-pairs where it divides the computational workload to be per each triangle pair. The hybrid-on-triangle-pairs method runs first the penalty solver on one triangle pair, if the solution is not within the user specified tolerance, we fall back to brute force to solve the problem naively.

Algorithm 4. *Hybrid On Triangle Pairs*

```
1: function HYBRIDONTRIANGLEPAIRS(PARTICLEA, PARTICLEB, TOLERANCE)(
       )
2:     for i = 0 to n particleA.triangles do
3:         for j = 0 to n particleB.triangles do
4:             triangleError = penalty(particleA[i], particleB[j])
5:             if triangleError ¿ tolerance then
6:                 bruteForce(particleA[i], particleB[j])
7:             end if
8:         end for
9:     end for
```

```
        return P, Q points on two triangles
10: end function
end
```

The other variant is the hybrid-on-batches where it is hybrid per triangle batches. This method is checking the error less frequently than the previous variant and uses an error for the whole batch. As in the hybrid-on-triangle-pairs variant; we run then penalty method on one batch of triangles and then fall back to brute on the whole batch if the error tolerance is not satisfied. The batch size can be set by the user to be of any arbitrary size. For our application we set it to be the number of triangles of our non-spherical particles (tessellation size of 60 triangles).

Algorithm 5. *Hybrid On Triangle Batches*

```
1: function hybridOnTrianglePairs(particleA, particleB, tolerance)(
        )
2:     for i = 0 to n particleA.triangles do
3:         for j = 0 to n particleB.triangles (batchSize) do
4:             batchError = penalty(particleA[i], particleB[j])
5:         end for
6:         if batchError ¿ tolerance then
7:             for j = 0 to n particleB.triangles do
8:                 bruteForce(particleA[i], particleB[j])
9:             end for
10:        end if
11:    end for
        return P, Q points on two triangles
12: end function
end
```

Both hybrid methods suffer by their nature by the granularity of the grain size whether that is singular size (triangle pair) or greater (batches of pairs). The memory distribution of pairs of triangles that do not converge within the mean number of Newton iterations are not known a priori because the solution depends on the underlying geometry. Triangle pairs or triangle batches that do not converge within the set tolerance skew the overall error distribution margin, potentially creating the worse case scenario where the method becomes a worse than brute force solver with both penalty and brute force being executed in sequence. It is not possible to predict the sparsity/distribution of non-convergent triangle pairs/batches during run-time so the tolerance value, penalty, regularization parameters are vital. In our experiments those parameters are set based on empirical tuning and trial and error.

5 Shared memory parallelisation

We introduce three levels of shared memory parallelisation. On the highest level, we exploit the cell level decomposition and assign work within each cell to cores. Within the cell we assign work per particle pair to an inner fork-level of threads. Lastly within each particle pair the innermost level of parallelisation is utilised to exploit tessellation level parallelisation by the contact solver.

Three levels of multicore parallelisation:

a) on the cell level b) within one 'cell' do all the particles in parallel c) within one particle-particle sweep, do all the triangles in parallel

For the multi-core scaling experiments we setup runs that use the hybrid solver and the brute force solver. We base performance experiments on three main runtime setups. We use meshed non-spherical particles of approximately 60 triangles each based on the Delauny triangulation (see appendix) derived from a 50 sphere shaped point cloud. We scale the contact problem in two aspects; by number of non-spherical particles and by irregularity of particle radius size. We use regular grid, adaptive and reluctant grids for grid-based adaptivity. Lastly, the particles mass is homogeneous to the particles and the initial velocities are random for every particle. To enhance the time to solution we use spheres as a last stage boundary box to minimize the number of mesh to mesh contact solving per timestep per cell per vertex.

In the innermost level of parallesation we are experimenting with several solver methods to resolve contact points. At this level computation is parallelised based upon triangle pairs and triangle batches and assigned to threads statically. From the performance measurements in Figure 22 it is observed that Hybrid-on-triangle-pairs scales but it is slowest method to solution per timestep.

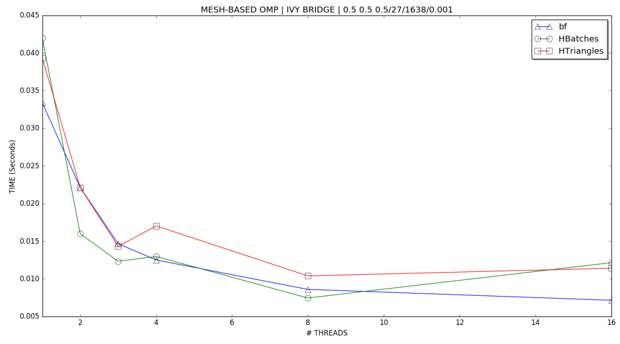


Figure 9: Triangle based shared memory parallelism using openMP running brute force (bf), hybrid-on-batches (HBatches) and hybrid-on-triangle-pairs (HTriangles) methods.

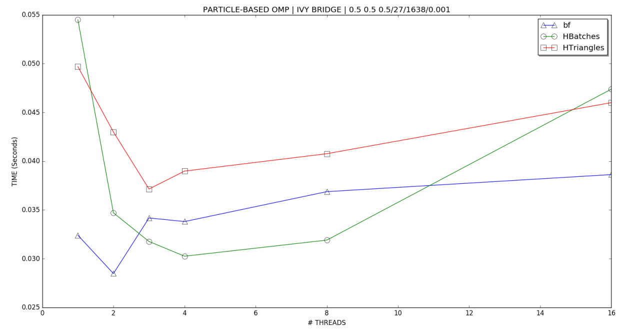


Figure 10: Particle based shared memory parallelism using openMP running brute force (bf), hybrid-on-batches (HBatches) and hybrid-on-triangle-pairs (HTriangles) methods.

The alternative method of parallelisation is based on whole particle pairs. Thread subscription is performed during the outer sweeping of particles where pairs-of-particles per grid-vertex are computed in parallel on each vertex visit. In Figure 10 we see that again hybrid-on-triangle-pairs is performing worst and hybrid-on-batches to scale better. Brute force does not scale as well as hybrid-on-batches but has faster time to solution on low number of threads.

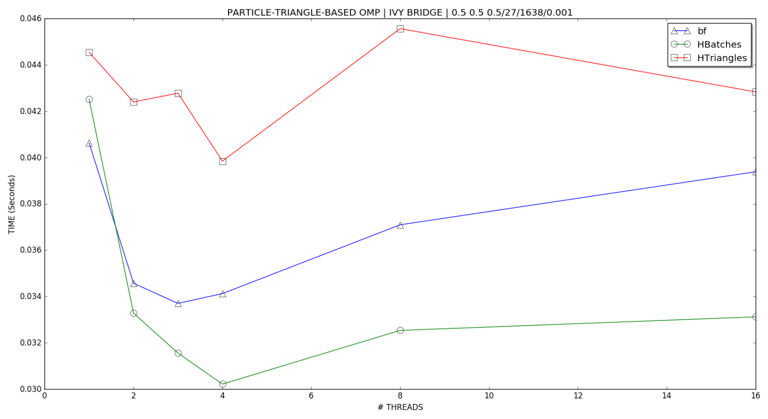


Figure 11: Particle and Triangle based nested shared memory parallelism using openMP running brute force (bf), hybrid-on-batches (HBatches) and hybrid-on-triangle-pairs (HTriangles) methods.

Particle and Triangle parallelism nests both particle and triangle shared memory threads. The runtime to solution is slightly slower than particle-based parallelism due to the thread/openMP overhead. Nevertheless it has an impact on brute-force method as it scales smoother than particle-based only parallelism due to overhead. In this case hybrid-on-batches is also the fastest while the hybrid-on-triangle-pairs the slowest.

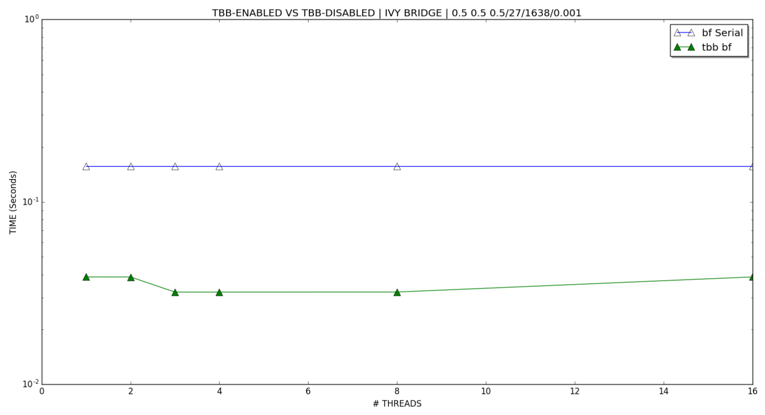


Figure 12: Cell based parallelism on Peano compared to serial runs using Intel TBB

Another aspect of shared memory performance is thread scheduling and balancing implications. The non-deterministic nature and error-prone distribution of triangles of the hybrid algorithm as discussed in Chapter -hybrid chapter-would suggest that alternative scheduling would pay off. But for our experiments dynamic and guided thread scheduling didn't not have a performance improvement on our tests but they rather create an scheduling overhead instead. That is partly due to the distribution of error in the triangle pairs and due to the granularity of our batches. Insignificant improvements were shown to our worst hybrid-on-triangle-pair method when using dynamic scheduling with OpenMP on the ivy bridge machine.

At the highest level of shared memory parallelism is the grid cell-based multicore processing. It is based on the peano-framework that is using Intel TBBs to assign cells on threads. This has significant impact on the overall runtime performance in Figure 12 it is compared with plain serial execution. Number of threads on the x axis refer to TBB Cell-based threads, at the contact detection method level the computation is performed without shared memory parallelism but with vectorization enabled. For the specified experiment in Figure we observe no cell-based thread scaling but only reduction of time to solution. But when we increase the problem size further (figure 13) we see that cell-based parallelism scales and it enhances execution time.

Overall shared memory parallelism on these three levels of parallelisation yield performance gain on manycore systems for our hybrid-on-batches method and in some cases brute force. For small problems triangle-based parallelism show good time to solution but for larger (**put plot) problem sizes particle-based parallelism pays off better. The finer/inner the thread forking the less significant scaling it is observed for larger problem sizes. For large problem sizes the combination of cell-based plus particle-based multi-core parallelism using

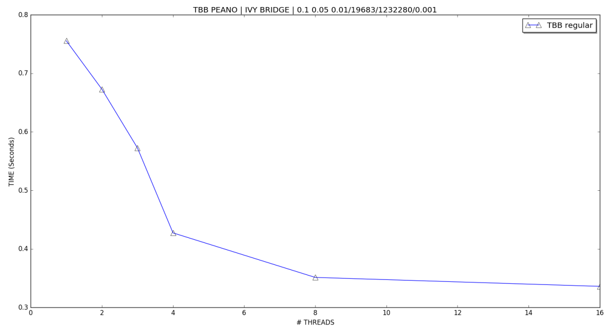


Figure 13: Cell based parallelism on Peano compared to serial runs using Intel TBB

hybrid-on-batches makes sense. Furthermore additional overall speedup can be gained if spheres are used as a filtering bounding box stage to our triangle-to-triangle contact detection.

6 Application benchmarks

7 Performance results

We study the behaviour of our algorithms at hands of an artificial granulate setup. Granulates of various diameters between $diam_{min}$ and $diam_{max}$ are randomly distributed in a unit cube. They are assigned random velocity and fixed density. If particles bump into the unit cube wall, they are reflected by applying an opposite direction in the velocities. We apply a full elastic collision model by allowing virtual penetration using the DEM spring-dashpot penalty force function (**??**). Each particle is created by a randomised algorithm that places a random number of 50 points within the unit sphere. We then use the hull algorithm [**?**] using a Delaunay triangulation to create the particle that is finally shrinked to fit into the prescribed diameter. On average, we obtain around 60-65 triangles per non-spherical particle.

The underlying spacetree is based upon tree-partitioning as we rely on the AMR framework Peano [14]. All experiments are realised with double precision. We note that the combination of the present work with reduced floating-point accuracy, notably following [1], however could be advantageous. All particle velocities and time step sizes Δt are chosen reasonably small such that particles do never penetrate or jump over more than one grid cell per time step. Higher velocities require more sophisticated collision models and a more sophisticated vertex association as discussed in [16]. Particles are assumed to collide as soon as they get closer than $\epsilon = 10^{-8}$. Adaptive or implicit time stepping [**?**] are not subject of study. Two different setups are realised: In the first experiment, we do not imply any external force. The particles fly through the unit cube and collide with each other. In the second experiment, we apply uniform gravity. The first setup yields an estimate how our code performs if the particle collission characteristics on the long term is time are invariant while the particle distribution is instationary. The second setup yields an estimate how the code performs if particles cluster and run into a stationary setup. Both experiments thus cover one extreme of a geometric challenge.

TW doublecheck ϵ

TK/KK citations

All single node experiments are ran on an Intel Xeon E5-2650 with two times 8 cores running at 2.0 GHz. On this system, we use Likwid [10] to read performance counters. All manycore experiments are done on an Intel Xeon Phi 5110P with 8 GByte of memory running at 1053 GHz in native mode. All parallel node experiments are ran on XXX nodes. Parallel experiments are ran on Durham University Hamilton supercomputer where per node there are 2 x Intel Xeon E5-2650 v2 (Ivy Bridge) 8 cores, 2.6 GHz processors, 64 GB DDR3 memory, 1 x TrueScale 4 x QDR single-port InfiniBand interconnect. We use Intel(R) MPI Library for Linux* OS, 64-bit applications, Version 5.0 Update 3 Build 20150128. For performance statements, we rely on the Intel 16 compiler.

penalty vectorization

hybrid vectorization

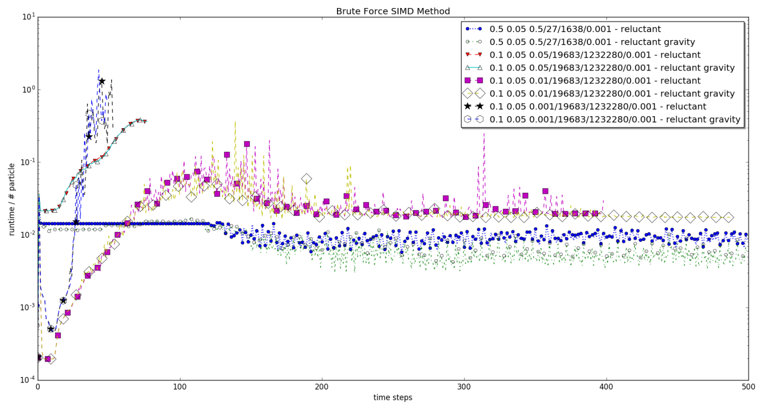


Figure 14: Brute Force SIMD runtimes.

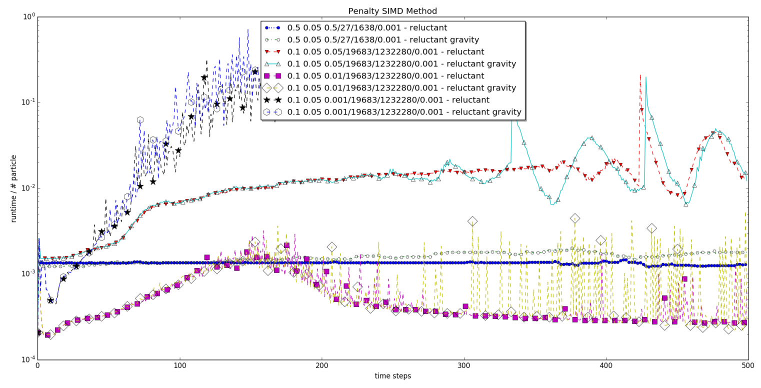


Figure 15: Penalty SIMD runtimes.

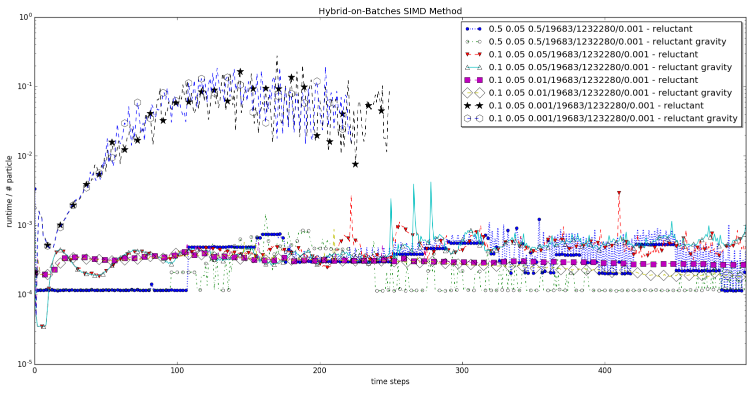


Figure 16: Hybrid-on-batches SIMD runtimes.

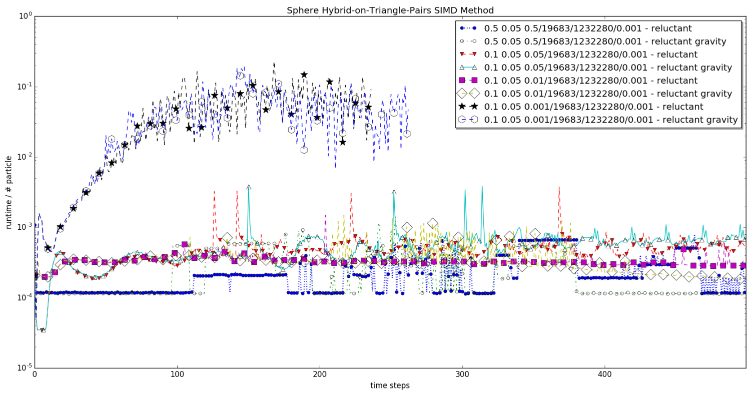


Figure 17: Hybrid-on-triangle-pairs SIMD runtimes.

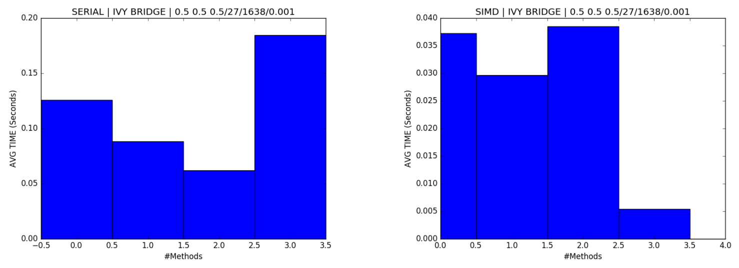


Figure 18: Left: Serial runtime comparison. Right: SIMD runtime comparison.

7.1 Impact of multiscale grid management

We start with single core experiments and $diam_{min} = diam_{max} \leq h_{max}$ where h_{max} is the grid spacing of a regular grid. Alternatively, we remove the grid completely. The number of triangle-to-triangle comparisons is reduced dramatically by the grid which reduces the runtime complexity (Figure **??**). Runtime measurements reflect this fact directly (not shown). If we release the equality constraint, select $diam_{min} < diam_{max}$ and allow particles to have an arbitrary diameter between $diam_{min}$ and $diam_{max}$, we retain the performance compared to the grid-free method. Once we however use a dynamically adaptive multiresolution grid with inter-grid particle comparisons, we can reduce the number of triangle-to-triangle comparisons further. From hereon, the characteristic triangle-to-triangle comparison counts from the multiscale algorithm are used. For the performance engineering, this is a worst-case choice as the arithmetic work is minimalised.

Cost Cache hit rate Stress on memory subsystem

8 Conclusion

Works also for FSI.

Maximale Zeitschrittweitenwahl. Dadurch uebergang zu event-driven scheme.

References

[1] W. Eckhardt, R. Glas, D. Korzh, S. Wallner, and T. Weinzierl. On-the-fly memory compression for multibody algorithms. In *Advances in Parallel Computing*. IOS Press, 2015. (in press).

[2] Wolfgang Eckhardt. Efficient HPC Implementations for Large-Scale Molecular Simulation in Process Engineering. 2014.

[3] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. *ACM SIGPLAN Notices*, 39(6):82, 2004.

[4] C Ericson. The Gilbert-Johnson-Keerthi algorithm, 2005.

[5] Florian Fleissner and Peter Eberhard. Parallel Load Balanced Particle Simulation with Hierarchical Particle Grouping Strategies. pages 33–44.

[6] K. Krestenitis, T. Koziara, and T. Weinzierl. Delta, 2016. github.com/KonstantinosKr/delta.

[7] Eric J R Parteli. DEM simulation of particles of complex shapes using the multisphere method: Application for additive manufacturing. *AIP Conference Proceedings*, 1542:185–188, 2013.

[8] Steve Plimpton. Fast Parallel Algorithms for Short – Range Molecular Dynamics. *Jounal of Computational Physics*, 117(June 1994):1–42, 1995.

[9] Chris H Rycroft, Terttaliisa Lind, Salih Güntay, and Abdel Dehbi. Granular flow in pebble bed reactors : dust generation and scaling. pages 447–455, 2012.

[10] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216. IEEE Computer Society, 2010.

[11] Anthony Wachs, Laurence Girolami, Guillaume Vinay, and Gilles Ferrer. Grains3D, a flexible DEM approach for particles of arbitrary convex shape - Part I: Numerical model and validations. *Powder Technology*, 224:374–389, 2012.

[12] Anthony Wachs and Andriarimina Daniel Rakotonirina. A MPI / domain decomposition strategy for large-scale simulations of granular media made of particles of arbitrary shape. 130(2011):69360, 2012.

[13] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids.* Verlag Dr. Hut, München, 2009.

[14] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2016. www.peano-framework.org.

[15] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, October 2011.

[16] T. Weinzierl, B. Verleye, P. Henri, and D. Roose. Two particle-in-grid realisations on spacetrees. *Parallel Computing*, 52:42–64, 2016.

A Software

All underlying software is free and open source C++ code. `Delta` Δ is available from [6] and offers all the functionality introduced in Section 4 through **??**. All spacetree and adaptive mesh refinement routines (Section refsection:grid) used in the present work rely on the framework Peano [14, 13, 15]. All geometric operations as well as DEM-specific compute kernels however are independent of Peano and can be used with any other (spacetree-based) software.

We offer `Delta` Δ with single (`float`) and double (`double`). The accuracy is controlled via a compile flag `-DiREAL=`. Subject of study here is exlusively `double`. Studies on reduced accuracy computations are beyond the scope of the present work.

B Acknowledgements

This work made use of Durham University's local HPC facilities Hamilton. We appreciate the support from Intel through Durham's Intel Parallel Computing Centre (IPCC) which gave me access to latest Intel software. The work has been sponsored by EPSRC (Engineering and Physical Sciences Research Council) and EDF Energy as part of an ICASE studentship. It also made use of the facilities of N8 HPC provided and funded by the N8 consortium and EPSRC (Grant No. N8HPC _DUR _TW _PEANO). The Centre is co-ordinated by the Universities of Leeds and Manchester. We also thank University of Durham for the supercomputing resources and technical assistance.

C Experiment: Two particles

The present experiments study two particles that bump into each other and then move away because of the spring dashpot forces. The two particles are represented by 126 triangles. We simulate 15,000 time steps. A contact is detected first in iteration 9,541 and introduces a force making the two particles to move away from each other. The code however still needs another 5 iterations to make the particles be away from each other more than $\epsilon = 10^{-4}$. If the two particles are compared, this results in 3,944 triangle-triangle comparisons.

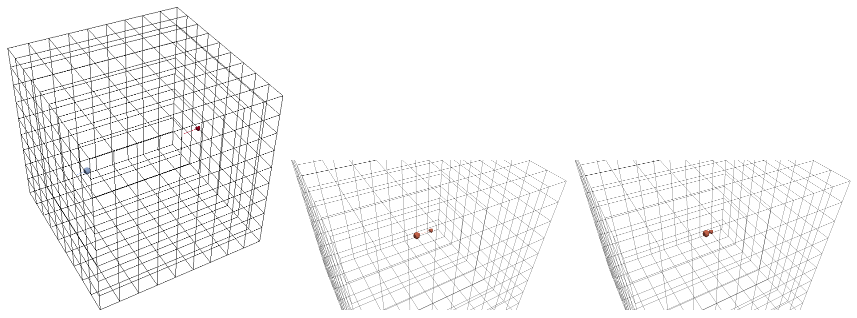
The experiments can be rerun with

```
./dem-3d-asserts 0.3 0.1 0.02 two-particles-crash 15000 regular-grid 0.0001 \
upon-change 0.0

./dem-3d-asserts 0.3 0.1 0.02 two-particles-crash 15000 adaptive-grid 0.0001 \
upon-change 0.0

./dem-3d-asserts 0.3 0.1 0.02 two-particles-crash 15000 reluctant-adaptive-grid \
0.0001 upon-change 0.0
```

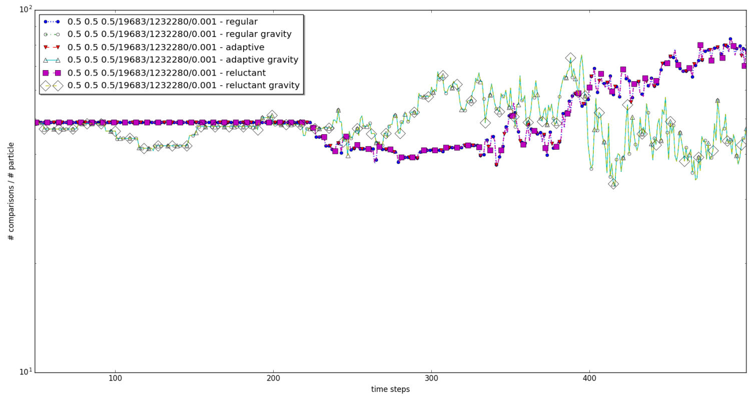
Regular grid. We obtain a grid with 1,032 vertices. The first 6,334 iterations, we do not perform any contact detection. The subsequent grid traversals perform all 3,944 comparisons till iteration 12,503. Starting from time step 12,504, the particles are far away from each other again and we do not compare anything anymore.

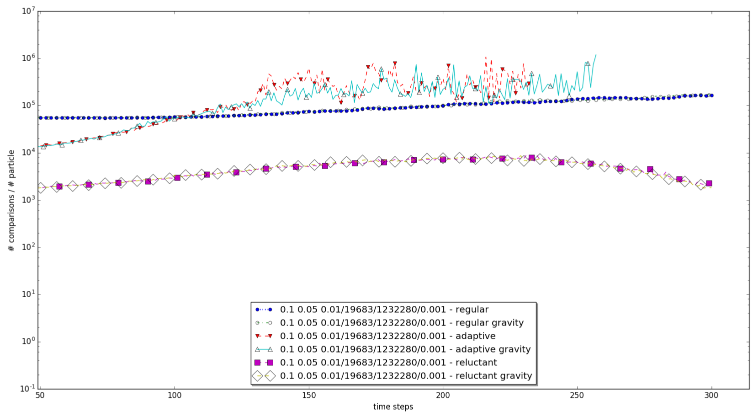


Adaptive grid. We obtain a grid with 747 vertices. Only few iterations (when one particle enters a neighbouring region) make the number increase to 980, but these grids are immediately after reduced to 747 again. The first 8,556 iterations, we do not perform any contact detection. The subsequent grid traversals perform all 3.944 comparisons. In iteration 9,539 we detect contact and the particles start to move away from each other, but we continue to check for further contacts till iteration 10,415. From here on, the particles are far away from each other again and we do not compare anything anymore.

Reluctant adaptive grid. The reluctant adaptive grid behaves qualitatively similar to the plain adaptive grid though yields quantitatively a smaller number of vertices. We start from a grid with 498 vertices, i.e. we do not refine the grid down to the same fine level as the adaptive approach. In iteration 6,334, both particles enter the same coarse grid cell and we run 3,944 triangle-triangle comparisions. They yield no contact yet. The reluctant criterion refines the grid that has now 980 vertices starting from iteration 6,338 on. The code continues to run without any further comparisons till iteration 8,556 when the particles again are close to each other (close this time w.r.t. finest grid level allowed by the particle diameters). Between iteration 8,557 and 9,533 we continue to run with 980 vertices and 3,944 comparisons. The code decides to reduce the grid in iteration 9,536 to 747 vertices when the particles are approach each other within a single cube that is resolved with a fine grid. We detects contact in iteration 9,541 after which the particles move away from each other. The 747 vertices are preserved and the code continues to run 3,944 comparisons. In iteration 10,452, the particles are far away from each other. No more comparisons are performed from hereon. In iteration 14,556 the particles are far away from each other, leave the fine tessellation around the previous contact point and we continue with a coarser grid of 498 vertices.

D Experiment: Some comparison statistics





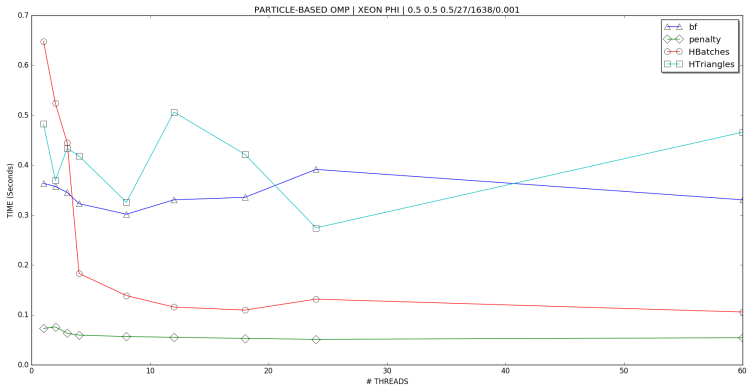


Figure 19: Xeon Phi scaling OpenMP dynamic scheduling triangle-based shared memory.

E Randomised particle generation

For our particle generation, we rely on the convex hull algorithm [**?**, **?**, **?**]. We place 50 points random on the unit sphere They act as input to the computation of a convex hull which yields around 60 triangles typically. Each vertex of the resulting mesh then is scaled with a random scalar from $[0.75, 1]$. Thus, we obtain distorted particles that do not resemble a sphere and can be slightly convex. The resulting mesh is finally suitably dilated and scaled.

KK: Can you please add here two or three pictures from typical particles?

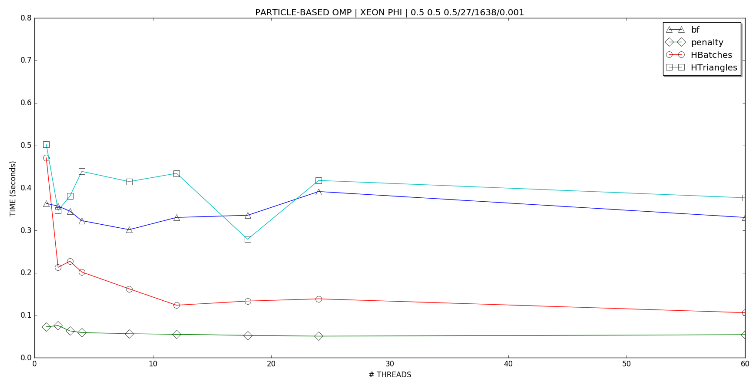


Figure 20: Xeon Phi scaling OpenMP static scheduling triangle-based shared memory.

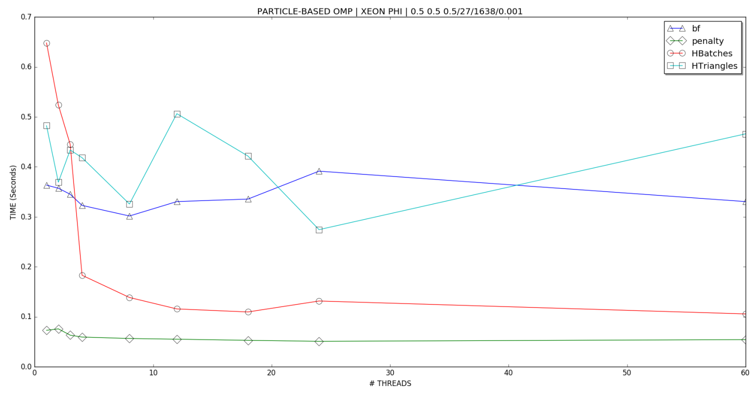


Figure 21: Xeon Phi scaling OpenMP dynamic scheduling particle-based shared memory.

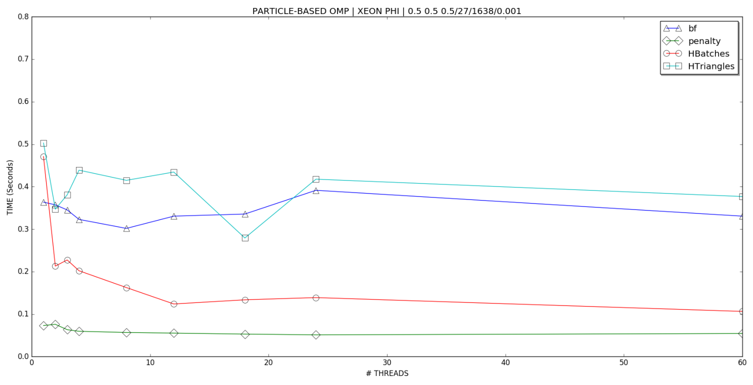


Figure 22: Xeon Phi scaling OpenMP static scheduling particle-based shared memory..

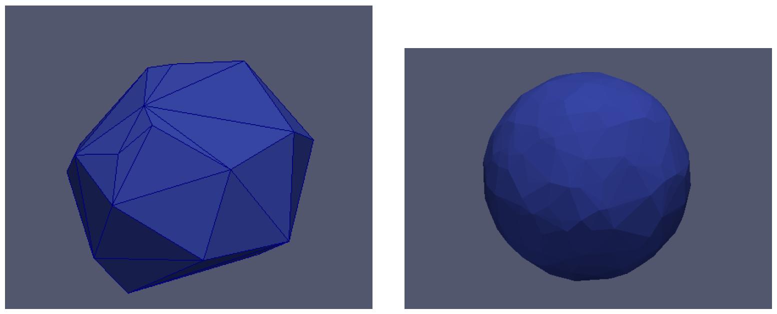


Figure 23: Left: Coarse non-spherical particle triangulation using Hull and Delaney algorithm. Right: Fine spherical particle triangulation using Hull and Delaney algorithm.

F Appendix Algorithms

Algorithm 6. *Brute-force distance calculation.*

```
function BF(A, B, C, D, E, F)
    T1=[A;B;C;];
    T2=[D;E;F;];
    list[0]= pt(T2, T1[0]);
    list[1]= pt(T2, T1[1]);
    list[2]= pt(T2, T1[2]);
    list[3]= pt(T1, T2[0]);
    list[4]= pt(T1, T2[1]);
    list[5]= pt(T1, T2[2]);
    ptmin = min(list);
    list[0]=segseg(T1[0],T1[1],T2(0],T2[1]);
    list[1]=segseg(T1[0],T1[1],T2[1],T2(2]);
    list[2]=segseg(T1[0],T1[1],T2[2],T2(0]);
    list[3]=segseg(T1[1],T1[2],T2[0],T2(1]);
    list[4]=segseg(T1[1],T1[2],T2[1],T2(2]);
    list[5]=segseg(T1[1],T1[2],T2[2],T2(0]);
    list[6]=segseg(T1[2],T1[0],T2[0],T2(1]);
    list[7]=segseg(T1[2],T1[0],T2[1],T2(2]);
    list[8]=segseg(T1[2],T1[0],T2[2],T2(0]);
    ssmin = min(list);
    min = min(ssmin, ptmin);
    P = min.p; Q = min.q;
end function
end
```

Algorithm 7. *Point to triangle distance algorithm.*

```
function PT(T, Point)
    for all regions i from 0 to 6 do
        if projection in Region i then
            determine (s,t) parametric values
        end if
    end for
     return Q point on triangle
end function
end
```

Algorithm 8. *Segment to segment distance algorithm.*

```
function PT(segment1, segment2)
    if segment1 and segment2 are parallel then
     return any point P on segment1 and any point Q on segment2
    else
        Get the closest points sC and tC on the infinite lines L1, L2
        if $sC \leq 0$ then
```

```
 6:           s = 0 edge is visible =¿ s = 0
 7:       **end if**
 8:       **if** $sC \geq 0$ **then**
 9:           s = 1 edge is visible =¿ s = 1
10:       **end if**
11:       **if** $tC \leq 0$ **then**
12:           t = 0 edge is visible =¿ t = 0
13:       **end if**
14:       **if** $tC \leq 0$ **then**
15:           t = 1 edge is visible =¿ t = 1
16:       **end if**
17:   **end if**
        **return** P, Q points on two segments
18: **end function**
end
```