A fast triangle-based discrete element code

Konstantinos Krestenitis Tobias Weinzierl
Tomasz Koziara*

August 12, 2016

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

Abstract

Nothing written yet. Nothing written yet.

1 Introduction

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. Lorem ipsum. Lorem ipsum.

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?

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

Major objective/contribution of this paper.

Tomek: citation.

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

```
            force = granular(velocity(z), position(z), contacts(z).getcontact(k))
        end for
    end for
    t ← t + Δt
end for
end
```

We focus on DEM with explicit time stepping (Algorithm 1). A straightforward implementation consists of one outer time stepping loop hosting three [illegible] The first inner loop detects contact points between [illegible] angles and check each triangle vs. the others. It thus has to loop over the 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 and [illegible] the particles.

Contact detection. We employ a visco-elastic particle model with spring forces here, where the actual particle is incompressible. However, it is equipped with a small halo layer of size $\epsilon > 0$: While the particles are rigid bodies, 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 where the actual particles are blown up by a circle with radius ϵ and may penetrate each other up to depth of ϵ.

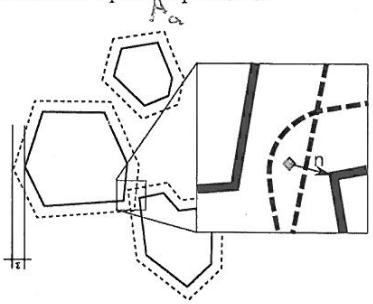


Figure 1: Left: Three particles with their ϵ environment. The particles do not penetrate each other, but two particles plus their ϵ environment penetrate and create one contact point with a normal.

As we apply a solid particle model, contact always is a unique point. We define it to be the centre of an overlap region (Figure 1), i.e. the distance d between a particle A and its contact point with a second particle is $0 < d \leq \epsilon$. Each contact point is equipped with a normal n pointing from the contact point to the surface of the contacting body's surface. As the distance is positive, the normal is well-defined and we have $|n| \leq \epsilon$. Please note that the normal direction depends on whether we make particle A being hit by particle B or the other way

I think Chapter 2 has to describe our benchmark experiments and
- show that our particles give qualitatively different results to spheres
- quantifies how much more expensive (in floating point ops) our approach is
- ~~describe~~ compare total kinetic energy of spheres vs. polygons

scope [?], our outlook does discuss more sophisticated and adaptive time step choices which translate the present time-driven scheme into an event-driven algorithm [?].

3 Grid meta data structures

Various speedup techniques such as linked-cell lists [?] and Verlet lists [?] reduce the quadratic complexity in DEM codes. ~~Notably~~ inspiration in this context stems from the molecular dynamics community [?, ?]. ~~In the present paper,~~ we propose to rely on a generalised tree-based linked-cell technique that allows us to efficiently treat particles ~~of~~ from a vast range of diameters. Three observations support this design decision: First, particles colliding with other particles are close to these particles. It ~~thus~~ is sufficient to scan a certain environment around each particle for potential collision partners. ~~We do not have to run through all particles per particle~~. 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 uneccessary 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 against 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.

Alternative approaches select the mesh size to accomodate the minimal particle diameter [?]. In this case, larger particles overlap multiple cells. Such an approach requires more sophisticated bookkeeping of particle-cell relations. The idea is not followed up here. Instead, we prioritise algorithm simplicity—a property that is notably enabled through trees decomposing the computational domain.

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 [?]. Bipartitioning, i.e. the classic octree, works

Please add

Can you remove the second item completely?

Not required in a paper but perhaps in a thesis

Please add

Such an algorithm may not consume more than half a page

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

```
1: function TRAVERSEGRID(C)
2:     C_old ← C
3:     C ← ∅
4:     while traversal continues do
5:         if touchVertexFirstTime then
6:             for all particles p associated to vertex do
7:                 for all contact points c ∈ C_old associated to p do
8:                     Update f_trans(p) through c
9:                     Update f_rot(p) through c
10:                end for
11:            end for
12:            for all particles p associated to vertex do
13:                Update particle incl. its triangles
14:            end for
15:            for all particle pairs (p_i, p_j) associated to vertex do
16:                C ← C ∪ FINDCOLLISIONS(p_i, p_j)
17:            end for
18:        end if
19:        if enterCell then
20:            for all 2^d vertices adjacent to cell do
21:                for all particles p associated to vertex do
22:                    if particle should be associated to different vertex then
23:                        Reassign particle
24:                    end if
25:                end for
26:            end for
27:            for all (p_i, p_j) associated to different vertices (cmp. figure) do
28:                C ← C ∪ FINDCOLLISIONS(p_i, p_j)
29:            end for
30:        end if
31:    end while
32: end function
33: function MAIN(T)
34:    C ← ∅
35:    t ← 0
36:    for t < T do
37:        TRAVERSEGRID(C)
38:        t ← t + Δt
39:    end for
40: 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 asso-

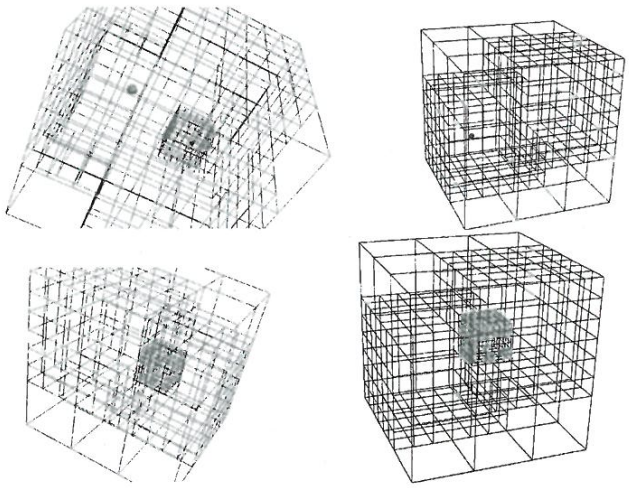


Figure 3: Two particles crash into each other. The adaptive grid refining around each particle while its diameter constrains the mesh size (left column). The reluctant adaptive grid works with a coarser resolution as long as particles are far away from each other (right column). Just before they collide,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

only if we shift them by half a grid traversal, i.e. perform the prediction right after a corrector in the same traversal. We introduce such a shift for equation system solvers in [?] and extend them to hyperbolic equation system solvers in [?].

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

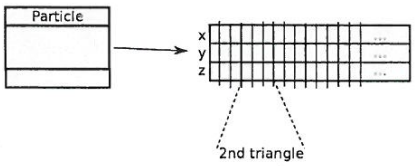


Figure 4: Right: The two layer data layout of our DEM code with an AoS on the particle level but SoA for the vector entries with replicated vector entries.

collision detection is. Collision detection is the computationally heavy activity in the algorithm. While other DEM discussions speak of a computational phase [?], we prefer 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. 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 particles or~~ two particle meshes ~~respectively~~ and identifies all contact points. Such an operation ~~with~~ quadratic complexity is often wrapped into an additional check that compares bounding boxes and thus may skip comparisons [?]. We abstain from such a check as the spacetree realises a related optimisation. The following text thus discusses ~~a nested loop with the outer loop~~ running over all triangles in $\mathbb{T}_A$ and ~~the inner loop running over~~ all triangles in $\mathbb{T}_B$.

4.1 Brute force geometric comparison

4.1.1 Segment to Segment Minimum Distance

The calculation of the distance between segments [8,9] in three dimensions is a geometric calculation that involve getting the closest points by extending the

is a unit square in (s,t) space. The four edges of the square are given by point at $s = 0, s = 1, t = 0, t = 1$. If $C = (sC, tC)$ is outside the G area, then it can "see" at most two edges of G. If $sC < 0$, C can see the $s = 0$ edge, if $sC > 1$, C can see the s = 1 edge, and similarly for tC, so in this way there is an enforcement of the required constraints. When C is not in G, at minimum 1 and at maximum 2 of constraints are active, and they determine which edges of G are candidates for a minimum of $|w|^2$.

For each candidate edge, to compute where the minimum that occurs on that edge, either in its interior or at an endpoint, it is possible to solve for the other unknown parameter since at minimum one is to be found (either t or s). So using the derivative of the $|w^2|$ equation it is always possible to solve for the parameter that is in the interior of G. For example when s = 0, $|w|^2 = ((P0 - Q0) - tv) \cdot ((P0 - Q0) - tv)$. Taking the derivative with t it possible get a minimum when: $0 = \frac{d}{dt}|w|^2 = -2v \cdot ((P0 - Q0) - tv)$. This gives a minimum on an edge at (0, t0) where $t0 = \frac{(v \cdot (P0 - Q0))}{(v \cdot v)}$ If $0 <= t0 < 1$, then this would be the minimum of $|w|^2$ on G, and P(0) and Q(t0) are the two closest points of the two segments. But in the case where t0 is outside G, then either (0,0) or (0,1), would be the minimum along that edge (since $s = 0$). Using this method it is possible to perform only a couple of checks to find the minimum of w(s,t) that correspond to the minimum distance between the segments.

4.1.2 Point to Triangle Minimum Distance

The other calculation required by the naive approach is to perform the check of point to triangle distance[12, 10, 11]. This test complete the brute force approach since it doesn't leave any space for cases where triangles are in a configuration where a point - triangle plane distance is the shortest distance in the initial triangle pair minimum distance problem.

To understand the problem of computing the minimum distance between a point P and a triangle T, it is necessary to understand the parameterised coordinate system of triangles. A triangle can be described by its barycentric coordinates in a way that any point on it can be dependent on two parameters, such that

$$T(s,t) = B + s \cdot E0 + t \cdot E1$$

where $B = point1$, $E0 = point2 - point1$, $E1 = P3 - P1$ for

$$(s,t) \in D = \{(s,t) : s \in [0,1], t \in [0,1], s + t1\}$$

The minimum distance is computed by determining the values $(s,t) \in D$ in the squared-distance equation $Q(s,t) = |T(s,t) - P|^2$ where $T(s,t)$ correspond to a point on the triangle closest to P. The function is quadratic and can be written as:

$$Q(s,t) = as^2 + 2bst + ct^2 + 2ds + 2et + f$$

where $a = E0 \cdot E0$, $b = E0 \cdot E1$, $c = E1 \cdot E1$, $d = E0 \cdot (B - P)$, $e = E1 \cdot (B - P)$, and $f = (B - P) \cdot (B - P)$.

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 region three and region five (Figure 6). In the case where (s, t) occur in region three then the minimum has to occur at (0, t0) following the process like in region one where $t0 \in [0,1]$. If (s,t) is located in region five (Figure 6), then the minimum occurs at $(s0,0)$ where $s0 \in [0,1]$.

When (s,t) is located inside region two, there are three possibilities for the elliptic level curve that contact or intersect the boundaries of $D - region0$ area, the contact of the levels with the triangular region may occur in:

1. edge where $s+t=1$
2. edge where $s=0$
3. at point where $t=0$ and $s=1$.

That is because although the global minimum occurs in region two, there is a level curve of Q that contact the D but the contact point and the region inside the level curve does not overlap. At these occurrences then the negative of the contacting level curve of Q cannot point inside D. For example for region two could be the direction of $-\nabla Q(0,1), -\nabla Q(s,1-t) and -\nabla Q(0,t)$, which points towards the inside of the level curve instead of inside D.

In order to determine which of the three cases occur, it is possible to check which areas are negative so in a way it is possible to eliminate the cases where the contact doesn't occur and where it does. So if $\nabla Q = (Qs, Qt)$ and Qs and Qt are the partial derivatives of Q, it should be the case where $(0,-1)\cdot\nabla Q(0,1)$ and $(1,-1)\cdot\nabla Q(0,1)$ are not both negative. The two vectors $(0,1)$ and $(1,1)$ are directions that correspond to the edges $s=0$ and $s+t=1$, respectively. The choice of edge $s+t=1$ or $s=0$ can be made based on the signs of $(0,-1)\cdot\nabla Q(0,1)$ and $(1,-1)\cdot\nabla Q(0,1)$. Similarly as for region three, the same calculation technique is used for the regions four and six.

```
20:     if PointProjection in Region 6 then
21:         determine (s,t) parametric space values
22:     end if
        return Qpointontriangle
23: end function
end
```

Algorithm 5. *Segment to segment distance algorithm.*

```
1: function PT(segment1, segment2)
2:     if segment1 and segment2 are parallel then
        return any point P on segment1 and any point Q on segment2
3:     else
4:         Get the closest points sC and tC on the infinite lines L1, L2
5:         if sC ¡ 0, the s = 0 edge is visible then
6:             s = 0
7:         end if
8:         if sC ¿ 0, the s = 1 edge is visible then
9:             s = 1
10:        end if
11:        if tC ¡ 0, the t = 0 edge is visible then
12:            t = 0
13:        end if
14:        if tC ¿ 0, the t = 1 edge is visible then
15:            t = 1
16:        end if
17:    end if
        return P, Q points on two segments
18: end function
end
```

Our default brute force approach implementation runs per triangle pair through all possible distance configurations (Algorithm 5). First, we run through all six involved vertices and compute the distance to the other triangle. Second, we determine the distances between each line segment-to-line segment combination. This yields 9+6=15 comparisons in total. The minimum distance results from a minimum over all computed distances. It is a robust approach, it always yields the correct answer.

The algorithm is ...

A fusion of multiple steps is not possible since all parametric/geometric scenarios have to be evaluated separately and naively. As shown the algorithm in Figure (x) the geometric checks and the brute force character of the algorithm does not allow computational fusion of execution branching.

4.2 Penalty-based formalism

The second approach to solve the triangle-to-triangle distance problem by parameterisation of the triangles such that the distance between them is formu-

Same game: One page!

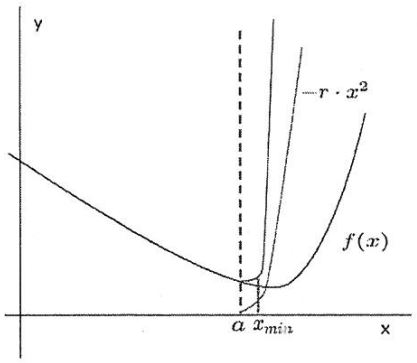


Figure 8: Illustration of a 2D problem showing the penalty function (red line) penalizing the objective function (black line) f(x) under a constraint a (dash line) to create the feasible region (blue line).

This approach adds a penalty term to the objective function to penalize the solution when outside of the feasible region:

$$P(x) = f(x) + r \sum_{i=1...6} max(0, c(x_i))^2 \tag{4}$$

Where r is the penalty parameter (Figure ??). Newton iterations always converge to a solution slightly on the outside of the feasible region. Convergence can be controlled by the r parameter that controls the sharpness of the curve for the constraints. One aspect that requires care however is the invertibility of the Hessian $\nabla\nabla P$.

Furthermore the problem is ill conditioned and a Quasi-Newton method is used. The Hessian matrix is not invertible so it is not possible to solve the direction of the search by computing the Hessian and gradient. 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 ill conditioning is caused by the problem definition itself, where there is a state where there are multiple solutions to the problem based on the orientation of the two triangles. This is also revealed by the two zero eigenvalues of the Hessian. Because of the ill-conditioning, we use a quasi-Newton approach, where the Hessian is approximated by a perturbed operator $\nabla\nabla P + \epsilon I$. I is an identity matrix and ϵ is suitably small.

Algorithm 6.
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'];

A high number of Newton steps can render the penalty method slower than the brute force approach.

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 ??. 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.

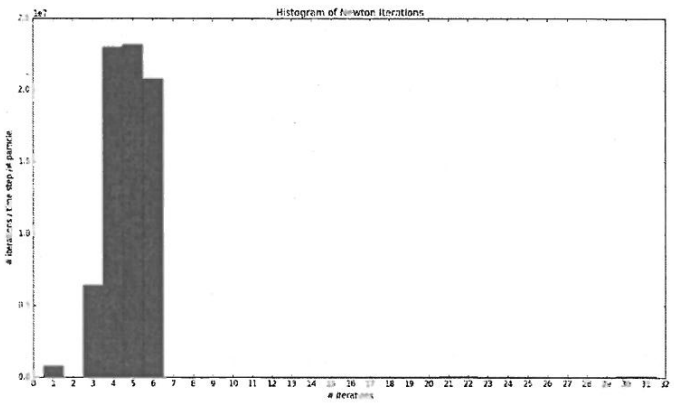


Figure 9: Histogram of number of iterations required by Newton Method for convergence on a sample of twenty four million triangle pair configurations.

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

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.

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.

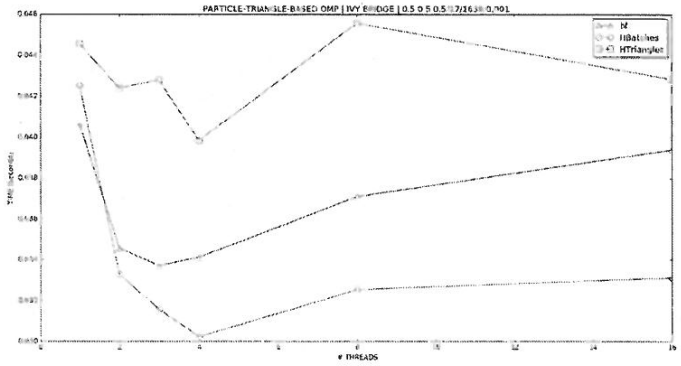


Figure 12: Particle and Triangle based nested shared memory parallelism using openMP running brute force (bf), hybrid-on-batches (HBatches) and hybrid-on-triangle-pairs (HTriangles) methods.

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.

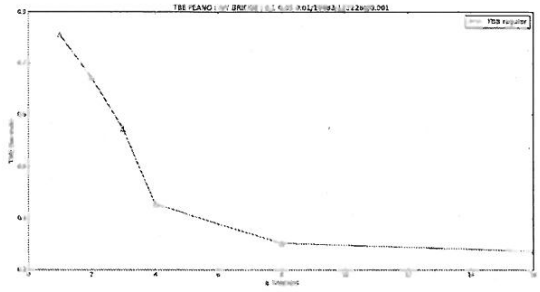


Figure 14: Cell based parallelism on Peano compared to serial runs using Intel TBB

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.

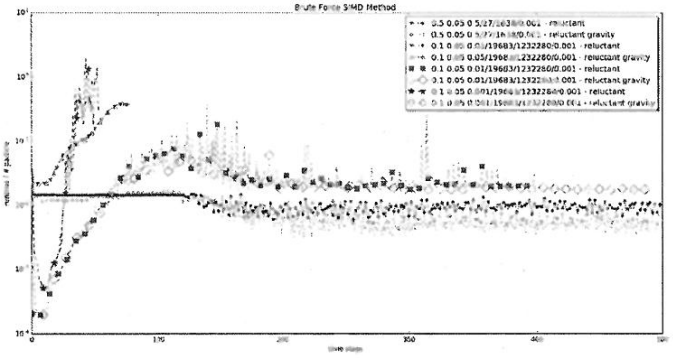


Figure 15: Brute Force SIMD runtimes.

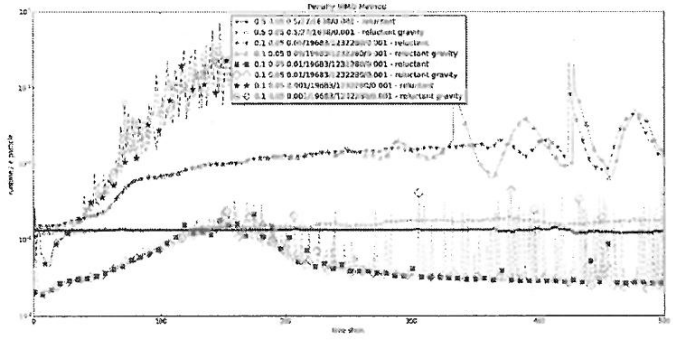


Figure 16: Penalty SIMD runtimes.

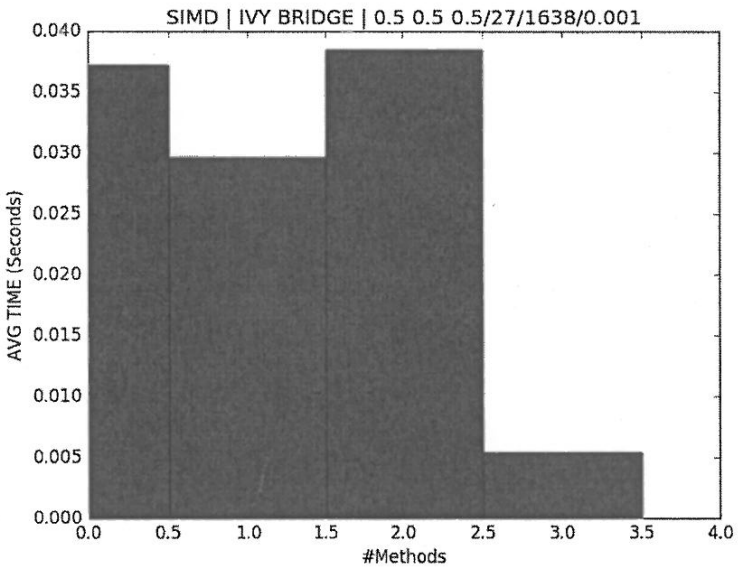


Figure 19: SIMD runtime comparison.

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.

A Software

All underlying software is free and open source C++ code. `Delta` Δ is available from [?] 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 [?, ?, ?]. 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.

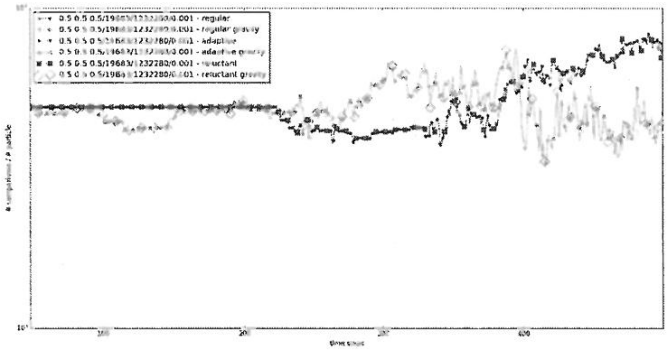
We offer `Delta` Δ with single (`float`) and double (`double`). The accuracy is controlled via a compile flag `-DiREAL=`. Subject of study here is exlusively `double`. Studies on reduced accuracy computations are beyond the scope of the present work.

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

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



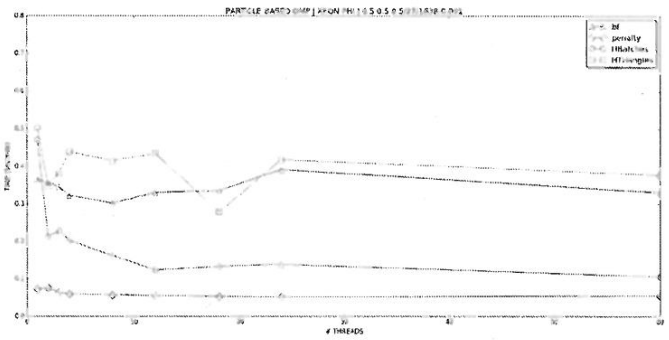


Figure 22: Xeon Phi scaling OpenMP static scheduling triangle-based shared memory.

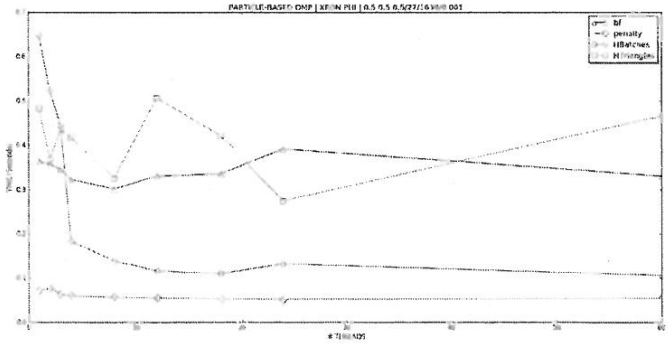


Figure 23: Xeon Phi scaling OpenMP dynamic scheduling particle-based shared memory.

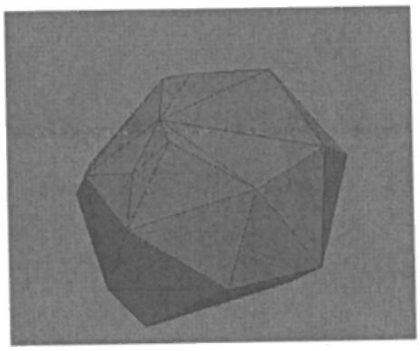


Figure 26: Fine spherical particle triangulation using Hull and Delaney algorithm.