A Parallel Grid-based Approach for Multiscale Non-Spherical Geometry Impact Dynamics

Konstantinos Krestenitis[1]

February 16, 2017

[1]School of Engineering and Computing Sciences, University of Durham, DH1 3LE, Durham
konstantinos.krestenitis@durham.ac.uk
Mechanics Research Group

Supervised by
Dr Tobias Weinzierl
Dr Tomasz Koziara (former)
Professor Jon Trevelyan

Contents

1 Introduction

We present a Discrete Element Method (DEM) contact detection code that simulates rigid non-spherical particles on manycore shared memory machines and distributed memory computers. DEM is used to study granular particles in fields like soil mechanics. We rely on triangulated particle meshes to model surfaces. Spherical or multi-sphere models currently are state-of-the-art - due to a lack of well-suited software and runtime demands. Non-spherical particles promise to facilitate more accurate physics than sphere-based approaches [**?**]. The focus on triangles for rigid contact mechanics facilitates memory layouts allowing vectorised computation [**?**, **?**, **?**]. It is vital to yield high performance on current and upcoming processor architectures to enable engineers to simulate more realistic materials.

It is important to investigate the problem of finding the minimum distance between triangles because of the changes in the computational hardware [**?**]. The central processing unit architecture today and the future upcoming hardware support Single Instruction Multiple Data (SIMD)[**?**] parallelism which allow data level parallelism. These speed-ups are enabled because of new instruction sets and wide vector register. Furthermore, shared memory parallelism at the node level as well as distributed memory for supercomputers can provide significant speed-ups. It is vital to extract resources available on current and upcoming hardware, the speed-ups enable more triangles to be used to describe surfaces. Consequently, our objective is to enable engineers to use an algorithm that is capable to handle the maximum amount of triangles per time step to do better engineering and science.

In the present work we propose a hybrid method that combines the advantages of two optimised triangle-to-triangle distance computation methods. It benefits from both performance and robustness of an iterative Newton-penalty and a brute force solver. We exploit shared memory parallelism and SIMD (Single Instruction Data) to perform contact detection at the node level. On distributed memory, we use spatial domain decomposition whilst with the Message Passing Interface (MPI), we exploit asynchronous non-blocking communication to overlap data exchange with computation.

The state-of-the-art large scale DEM work, to the best of my knowledge, relies on sphere-based or multi-sphere particles [**?**, **?**]. Large scale applications of spherical particles can be found in the field of molecular dynamics. Large scale non-spherical particle based DEM simulations in literature [**?**, **?**, **?**, **?**] often describe particles as convex polyhedra and contact detection is resolved with GJK (Gilbert-Johnson-Keerthi) [**?**] variants which require interpenetration and access to all vertices of the particle to detect contact on the surface. Interpenetration introduces contact point clustering and the problem of divergence [**?**, **?**, **?**]. In addition, convex polyhedra based contact detection methods require undeterministic memory access to all particle simplices [**?**] which is contradictory to the aligned memory access imposed by SIMD for data locality. Triangle-based contact detection retain locality for vectorisation as well as increases accuracy of contact point generation [**?**]. Furthermore, the method can be extended to run on shared memory manycore and distributed memory machines.

The remainder of the paper is structured as follows. In Section 2, we describe the serial DEM algorithm. In Section 3 we review two triangle-to-triangle distance algorithms.

We create a new hybrid shared memory method that benefits from fast convergence and robustness. Next, in Section 3, we propose a distributed memory algorithm for contact detection that use asynchronous communication to overlap computation over communication. Section 4, discusses the future research directions.

2 Outline Summary of Thesis

2.1 Overview

- Completed since last report: Implemented serial DEM simulation with triangle-to-triangle contact detection using spring-dashpot contact forces, explicit time step.
- New concepts: New shared memory hybrid method for computation of triangle-to-triangle distance.
- Open questions: Best-case design of distributed memory implementation is not clear yet, real-world experiments are missing.

In contact mechanics, fluid-stucture interaction or other fields, it is an essential task to compute the distance between geometries to determine contact. Contact detection also is the most expensive algorithmic step [**?**, **?**]. We present a Discrete Element Method (DEM) code that simulate particles that interact with spring-based contact. Non-spherical particles promise to facilitate more accurate physics than sphere-based approaches [**?**, **?**]. The contact detection routine determines contact based on the distance between the triangles of every particle against all other. If two triangles are closer than a prescribed threshold, they are considered to contact each other. The use of triangles for rigid body contact dynamics rather than arbitrary polygons simplifies geometric checks and facilitates memory layouts that allow vectorised computation [**?**, **?**, **?**, **?**].

2.1.1 Geometric Approximation

*polygons, sphere, multisphere

To create objects that

2.1.2 Interaction Model

Algorithm 1 Spring-dashpot force algorithm

```
0 FUNCTION force = penaltyForce(normal, relativeVelocity, depth, massA, massB)
1  mass = 1.0 / ((1/massA) + (1/massB));
2  velocitymagnitude = DOT(relativeVelocity, normal);
3  magnitude = SPRING*depth + DAMPER*sqrt(SPRING*mass)*velocitymagnitude;
4  force = magnitude*normal;
5 ENDFUNCTION
```

Algorithm 1 shows the spring-based force derivation approach [**?**, **?**, **?**, **?**]. To define contact points without penetration, we use the boundary layer margins to extend the surface boundary. Margin size is set based on velocities and time step size. When in contact, the margin overlap is used as the interpenetration depth. The contact point is at the middlepoint of the distance and normal direction is either side of the distance. At line 0, given the contact point normal, relative linear velocities, penetration depth and the mass of the two bodies we derive the interaction force to update the position of particle triangles on every time step.

2.2 Algorithmic Overview

The DEM Algorithm 2 demonstrates the execution of contact detection, an O(n^2) operation. In line 3 and line 4 the nested for loop indicate the complexity of iterating through all triangle pairs. In line 5 TTD(i, j) invokes a Triangle-to-Triangle Distance (TTD) algorithm that determines the distance between triangle i and triangle j. Based on the boundary layer margin contact model [**?**] to avoid penetration, we check if distance between Triangle i and j is smaller than a set margin (line 6). If the parent particle of triangle i and j is not the same then this indicate a contact point between the two particles (line 7). The contact point information at line 13 and line 14 is used to derive the forces. The forces accelerate the particles that are then integrated by an explicit time stepping scheme such as explicit Euler.

2.3 Domain Decomposition

2.3.1 Vectorization

**interaction models exploit SIMD architecture **properties of SIMD, bit precision **implementation, memory layout, implications

Algorithm 2 DEM Serial Simulation Pseudo code

```
1 FOR time = 0; time < simulation time; time+=step
2   //contact detection
3     FOR i = 0 to N triangles
4       FOR j = i+1 to N triangles
5         distance = TTD(i,j)
6         IF (distance < margin) AND ParticleID(i) != ParticleID(j)
7           contact(PID(i)).add(point, normal)
8         ENDIF
9       ENDFOR
10    ENDFOR
11  //force derivation
12    FOR z = 0 to NB particles
13      FOR k = 0 to contacts(z).size()
14          force = granular(velocity(z), position(z), contacts(z).getcontact(k))
15      ENDFOR
16    ENDFOR
17  //explicit time stepping
18 ENDFOR
```

2.3.2 Shared Memory

In the shared memory chapter of the thesis

2.3.3 Distributed Memory

In chapter distributed memory computation, the areas covered will be the feeding of compute nodes with dense arithmetic operations and the communication behaviour that emerges from this.

2.4 Case Study

*two case studies, study behaviour of grid and paradigm and effect on performance (memory, intensity, etc)

2.4.1 Hopper Flow

*hopper flow scenario description

2.4.2 AGR Seismic Shake

*seismic scenario, setup of objects/geometry, experiment setup and properties

3 Progress Report

3.0.3 Overview

3.0.4 Geometric Approximation

We utilise particles approximated by a random spherical point cloud, arbitrary particle surfaces are generated with the Delauny triangulation algorithm. The algorithm is scaled in three respects; number of non-spherical particles, irregularity of particle radius size, and mesh size.

3.0.5 Interaction Model

The brute force is the naive approach to contact detection, it computes all distances between geometrical primitives of the triangles. The method computes the distance between all line segments of one triangle with the line segments of the other using the segment-to-segment (Algorithm 3, line 22-30) distance function [**?**, **?**]. Then for every point of one triangle it computes the distance against the other triangle using the point-to-triangle (Algorithm 3, line 3-8) function [**?**, **?**]. The method requires evaluating the minimum distance among all comparisons performed to find the overall minimum distance in the end.

The algorithm is straight-forward because it relies on only two geometric comparisons. As seen in Algorithm 3 in lines 3 to 21 the six comparisons yield one that defines the distance. For segment-to-segment, let v_{1i}, e_{1i}, v_{2i}, e_{2i} be the vertices and edges of triangles T_1 and T_2. The brute force approach computes all fifteen distances v_{1i}-to-T_2, v_{2i}-to-T_1, e_{1i}-to-e_{2i}, and then we select the shortest distance. As seen in Algorithm 3 in line 22 to 37 the algorithm determines the segment-to-segment distance between all segments of the triangle pair. Finally the algorithms finds the minimum of the two sets of comparisons to get the P and Q vertices that define the distance between two triangles.

Optimization of the brute force method can at most rely on avoiding redundant evaluations parts of the computation. It is not possible to avoid logical branching because we have to evaluate all if statements to arrive to the solution. Therefore it is not possible to optimize the method for SIMD parallelism because of the branching.

Algorithm 3 MATLAB Brute Force Solver.

```
0  FUNCTION [min,P,Q] = bf(A,B,C, D,E,F)
1  T1=[A;B;C;];  T2=[D;E;F;];
3  [ptlist(1), P0(1,:)]= pt([D;E;F], T1(1,:));
4  [ptlist(2), P0(2,:)]= pt([D;E;F], T1(2,:));
5  [ptlist(3), P0(3,:)]= pt([D;E;F], T1(3,:));
6  [ptlist(4), P0(4,:)]= pt([A;B;C], T2(1,:));
7  [ptlist(5), P0(5,:)]= pt([A;B;C], T2(2,:));
8  [ptlist(6), P0(6,:)]= pt([A;B;C], T2(3,:));
9  ptmin = min(ptlist);
10  FOR i=1:6
11    IF(ptmin==ptlist(i))
12      IF(i¡4)
13        p1 = T1(i,:);  p2 = P0(i,:);
15      ELSE
16        p1 = P0(i,:);  p2 = T2(i-3,:);
18      END;  break;
20    END;
21  END;
22  [ss(1),sp1(1,:),sp2(1,:)]=segseg(T1(1,:),T1(2,:),T2(1,:),T2(2,:));
23  [ss(2),sp1(2,:),sp2(2,:)]=segseg(T1(1,:),T1(2,:),T2(2,:),T2(3,:));
24  [ss(3),sp1(3,:),sp2(3,:)]=segseg(T1(1,:),T1(2,:),T2(3,:),T2(1,:));
25  [ss(4),sp1(4,:),sp2(4,:)]=segseg(T1(2,:),T1(3,:),T2(1,:),T2(2,:));
26  [ss(5),sp1(5,:),sp2(5,:)]=segseg(T1(2,:),T1(3,:),T2(2,:),T2(3,:));
27  [ss(6),sp1(6,:),sp2(6,:)]=segseg(T1(2,:),T1(3,:),T2(3,:),T2(1,:));
28  [ss(7),sp1(7,:),sp2(7,:)]=segseg(T1(3,:),T1(1,:),T2(1,:),T2(2,:));
29  [ss(8),sp1(8,:),sp2(8,:)]=segseg(T1(3,:),T1(1,:),T2(2,:),T2(3,:));
30  [ss(9),sp1(9,:),sp2(9,:)]=segseg(T1(3,:),T1(1,:),T2(3,:),T2(1,:));
31  ssmin = min(ss);
32  FOR i=1:9
33    IF(sslist(i) == ssmin)
34      csp1 = sp1(i,:);  csp2 = sp2(i,:);
36    END;
37  END;
38  min = min(ssmin, ptmin);
39  IF min == ptmin
40    P = p1;  Q = p2;
41  ELSEIF min == ssmin
42    P = csp1(:,:);  Q = csp2(:,:);
43  END;
```

The second approach to solve the triangle-to-triangle distance problem is by parameterising of the triangles such that the distance between them is formulated by a quadratic function and a solvable minimization problem. This approach is iterative compared to brute force. The method approximate the solution using Newton iterations.

The method relies on an optimization-based formulation of the distance between two points. Let x and y be two points, belonging respectively to triangle T_1 and T_2. Assuming that points A, B, C are vertices of T_1 and that points D, E, F are vertices of T_2, x and y can be defined 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 between T_1 and T_2 and the corresponding two closest points P, Q on the two triangles we minimize

$$f\left(a, b, c, d\right) = \left\| x\left(a, b\right) - y\left(c, d\right) \right\|^2$$

What has to be noted is that x and y have to stay within the area of the two triangles. The four parameters of the function f have to comply with six inequality constraints:

$$min_{a,b,g,d} f(a, b, g, d)$$

$$such\ that:\ \{a \geq 0, b \geq 0, a + b \leq 1, d \geq 0, g \geq 0, g + d \leq 1\}$$

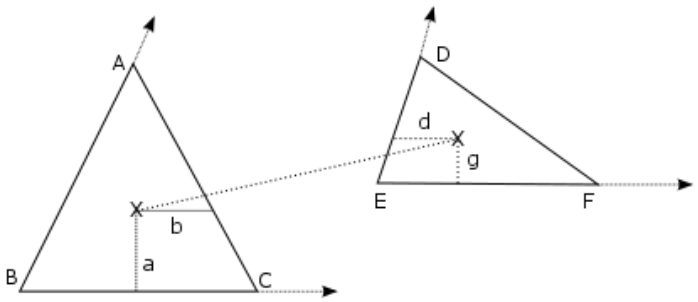

Figure 1: Example of minimum distance and the corresponding barycentric points (parameters of objective function) on a pair of triangles in 3D. Triangle X:T1 has points A, B, C where barymetric parameters a,b correspond to point x on the triangle. Triangle Y:T2 has points D,E,F where barymetric parameters g,d correspond to a point x. The two defined barymetric points define the minimum distance between the two triangles in 3D.

To find the minimum of the parabolic constraint problem the Newton-Raphson method is used. Newton method use information from the curvature of the problem using the Hessian and gradient to arrive to a minimum point. To enforce the constraints the problem is transformed into a series of unconstrained problems with the augmentation of the objective function f using the penalty-based method.

The penalty function penalizes the iteration so that the boundaries of the feasible region are valid; at least in a weak sense. Although there are factors that determine the number of iterations required the tuned-to-the-problem parameters yield a good number of Newton iterations that cannot be reduced to less than two; (initial guess to penalized boundary step, correction step).

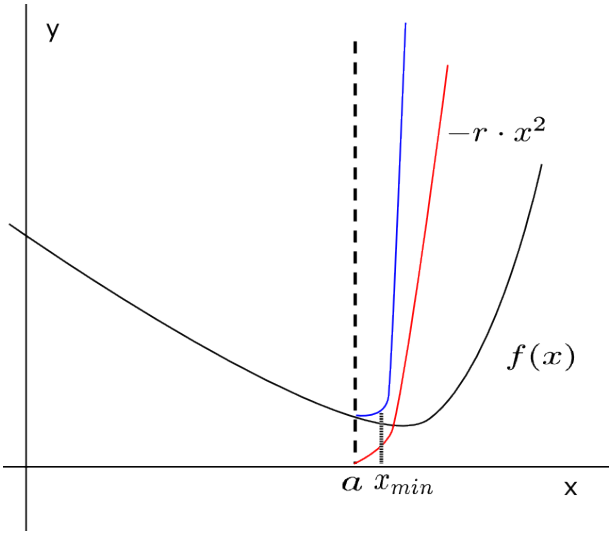

Figure 2: Illustration of a 2D problem showing the penalty function (red line) penalizing the objective function (black line) f(x) under a constraint a (dash line) to create the feasible region (blue line).

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

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

Where r is the penalty parameter (Figure 2). Newton iterations always converge to a solution slightly on the outside of the feasible region. Convergence can be controlled by the r penalty 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 that the problem is ill conditioned and a Quasi-Newton method has to be 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.

The penalty algorithm in pseudocode language is shown in Algorithm 4. It accepts A, B, C, D, E, F vector coordinates for triangle T1(A, B, C), T2(D, E, F) as well as the required parameters for the algorithm to be solved. Rho is the penalty parameter that controls the

Algorithm 4 MATLAB Penalty Solver.

```
0  FUNCTION x = penalty(A, B, C, D, E, F, rho, tol)
1  BA = B-A; CA = C-A; ED = E-D; FD = F-D;
2  hf = [2*BA*BA', 2*CA*BA',-2*ED*BA',-2*FD*BA';
3        2*BA*CA', 2*CA*CA',-2*ED*CA',-2*FD*CA';
4       -2*BA*ED',-2*CA*ED', 2*ED*ED', 2*FD*ED';
5       -2*BA*FD',-2*CA*FD', 2*ED*FD', 2*FD*FD'];
6  x = [0.33; 0.33; 0.33; 0.33];
3  FOR i=1:99
4    X = A+BA*x(1) + CA*x(2);
5    Y = D+ED*x(3) + FD*x(4);
6    gf = [2*(X-Y)*BA'; 2*(X-Y)*CA'; -2*(X-Y)*ED'; -2*(X-Y)*FD'];
7    h = [-x(1); -x(2); x(1)+x(2)-1; -x(3); -x(4); x(3)+x(4)-1];
8    dh = [-1, 0, 1, 0, 0, 0; 0, -1, 1, 0, 0, 0;
9           0, 0, 0, -1, 0, 1; 0, 0, 0, 0, -1, 1];
10   mask = h' >= 0;
11   dmax = dh.* [mask; mask; mask; mask];
12   gra = gf + rho * dmax * max(0,h(:));
17   hes = hf + rho*dmax*dmax' + eye(4,4)/rho^2;
18   dx = hes\gra;
19   DX = BA*dx(1) + CA*dx(2);
20   DY = ED*dx(3) + FD*dx(4);
21   error = sqrt(DX*DX'+DY*DY');
22   IF error < tol, BREAK; END
23   x = x - dx;
24 END
25 END
```

steepness of the P(x) function (equation 1) , eps is the perturbation parameter for the hessian matrix of the problem along its diagonal to make the matrix solvable. Tol is the tolerance for convergence (Floating point accuracy). In line 6 of Algorithm 4 an initial guess is chosen to be the center of the two triangles, then the for loop initiates the Newton iterations to find the points on the X, Y triangle planes under the constraints c. For each of the six constraints (line 12) the max function of the penalty is determined so that every possible active constraint is detected. In line 19 and line 20 the gradient and Hessian of P Penalty function is evaluated to be provided to the Gaussian elimination direct solver so that a Newton direction DX is solved. If the Newton step is large enough over the specified tolerance then the iteration is converged else the direction is used and the loop is executed once more recursively.

3.1 Algorithmic Overview

3.1.1 Space Decomposition

Various speedup techniques such as linked-cell lists [**?**] and Verlet lists [**?**, **?**] 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 [**?**]. Bipartitioning, i.e. the classic octree, works as well.

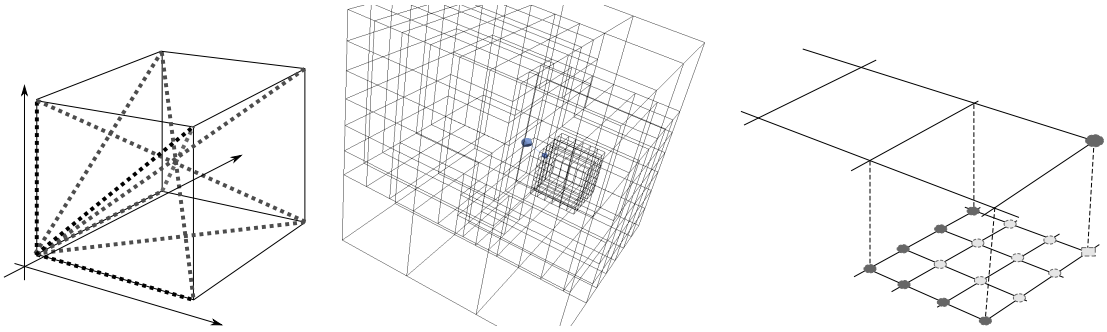

Figure 3: Left: Whenever the grid traversal enters a cell, it checks whether particles assigned to one vertex do collide with particles assigned to another vertex. To avoid redundant collision computations, we check only some vertex pairs (dotted, larger lines). Middle: Two particles approach each other. 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.

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 [**?**]. 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.

With a grid at hand, we may map the algorithmic steps from Algorithm **??** onto a grid traversal. For the traversal, we rely on a combination of a depth-first order with space-filling curve [**?**, **?**]. 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. Besides the actual grid hosting the particles, the grid sweeps build and maintain two further sets of collision points (algorithm 29).

1: **function** TRAVERSEGRID($\mathcal{C}$)
2: $\quad\mathcal{C}_{old} \leftarrow \mathcal{C}$
3: $\quad\mathcal{C} \leftarrow \emptyset$
4: $\quad$**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(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:               C ← C∪ 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:               C ← C∪ findCollisions(p_i, p_j)
26:           end for
27:       end if
28:   end while
29: end function
```

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 3—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 [**?**]. 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.

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 [**?**, **?**] 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 [**?**].

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 3). 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 $\bot$ 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 $\bot$ at the end of

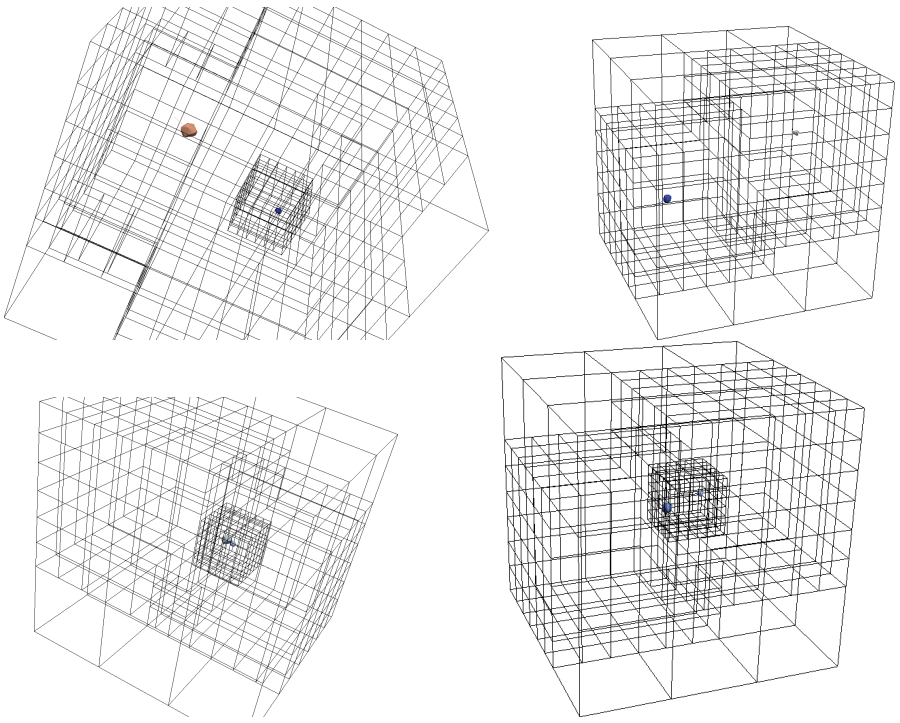

Figure 4: Two particles crash into each other. The adaptive grid refining around each particle while its diameter constrains the mesh size (left column). The reluctant adaptive grid works with a coarser resolution as long as particles are far away from each other (right column). Just before they collide, the grid is refined and particles are dropped down the resolution levels.

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 aggressive: Particles are always dropped to their corresponing refinement level immediately. Fine grid regions thus follow 'their' particles (Figure 4). 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.

3.1.2 Vectorization (SIMD)

All performance tests (Figure 5) used the latest AVX2 (Advanced Vector Extensions) instruction set on an Intel Sandy Bridge 2.0GHz i5 CPU with a memory of 600 MHz DDR3 RAM. For the barrier method isn't worth to do a performance comparison even though its C and ISPC version is implemented but not with great success due to its theoretical computational inefficiencies as discussed earlier in the report. Only the tuned penalty and brute force approaches are considered for real-world applications due their significant parallel speedup. The performance test comprises calculating the minimum distance between ten million pairs of triangles of length ten. The performance is scaled linearly for every magnitude length of pair triangle.

As it can be observed in Figure 5, the runtimes show that there is significant speedup when SIMD parallelism is used compared to sequential algorithm execution for all methods. For the brute force, serial Intel and GCC are the slowest with no significant difference in performance in using floats or double precision. That is due to little actual floating point computation. It has to be noted that our serial version is optimized considering memory bandwidth. For the Intel SIMD version of brute force, it can be seen that there

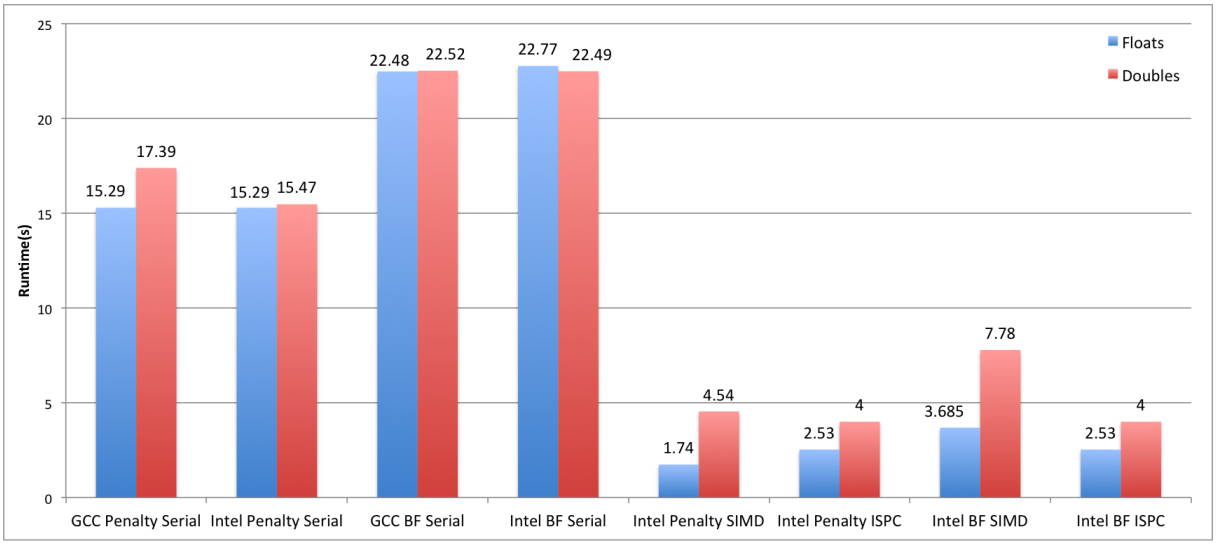

Figure 5: Run-time comparison of the sequential and parallel penalty and brute force methods using float and double precision, using ISPC, Intel, GCC compilers on a single AVX2 core.

is 6x speedup compared to its serial sequential variant using single precision. Speed up for double precision (3x times), both results reflect the logical branching involved in brute force. Nevertheless the speed up is significant. The ISPC compiler for brute force is capable to do vectorization better for double precision than the Intel compiler, which makes the new vectorization compiler well-suited for SIMD programming. As we haven't studied the results assembles codes, we cannot identify where the difference between ISPC and C come from.

As penalty solver is an iterative method, its performance is dependent on the tuned parameters. The r penalty parameter and *eps* are calculated automatically based on the tuning and algorithm study performed. Moreover, the technique discussed in the implementation chapter for converging in 4 iterations is used without losing significant numerical precision. The penalty method reaches the physical limits of SIMD architectures with 4x speed up for double precision and 8x speedup for single precision. This makes it the fastest algorithm. Intel compiled serial code is significantly faster than Intel and GCC versions of Brute force. GCC compilers for serial are slightly slower, but still faster than brute force GCC serial. ISPC, Intel compiler for SIMD show that they are very fast, slightly showing a speedup for penalty over brute force.

It can be observed see from performance measurements in Table 1 using likwid [49] that for serial double precision (same for single precision) both methods are not memory (Memory BW Bandwidth Megabyte per second) but compute-bound. For the machine used, the maximum memory throughput is 13 GB/s, so memory bandwidth is not used to the maximum. That is a result of careful programming, and as mentioned in the implementation section, due to the fact unnecessary memory operations are reduced extensively. The data volumn in Gigabytes is the same for both methods, since both methods use the same amount of input and output data. That is the triangle coordinates, P, Q point solution on the two triangles T1, T2 and the calculated distance which is derived from P and

Intel C++ Compiler Sequential Double Precision		
Algorithm	Penalty Method	Brute Force Method
Runtime unhalted (s)	15.47	22.49
MFLOPS/s	1073	962
CPI	0.49	0.48
Memory BW M/s	579	408
Data Volumn GB	6.7	6.77

Table 1: 64bit sequential computation; solving ten million random triangle pairs using the two methods.

Q vectors. The penalty method is managing to executes faster than brute force because it execute slightly more floating point operations than the brute force (Mega Flops per second MFLOPS/s).

Intel C++ Compiler SIMD Double Precision		
Algorithm	Penalty Method	Brute Force Method
Runtime unhalted (s)	4.54	7.78
MFLOPS/s	3202	2808
CPI	1.26	0.91
Memory BW M/s	1424	902
Data Volumn GB	5.15	5.2

Table 2: 64bit SIMD computation; solving ten million random triangle pairs using the two methods.

In Table 2 it can be seen that the SIMD implementation for both methods increase performance significantly. Their serial counterparts they both achieve more than 2x times speedup, with penalty achieving 3.40x speedup, and brute force 2.89x speedup. The theoretical maximum speedup that is possible to achieve is 4x times for 64bit precision using a 256bit wide register, and even more speedup for wider 512bit registers that are available on Xeon Phi processors. Using ISPC compilers however as shown in Figure 5 double precision achieve 4x speedup. For 32bit precision the speedup using AVX 256bit wide instruction sets is near 8x times as it can be seen in Figure 5.

These performance comparisons are the result of extensive low level programming and tuning of parameters to optimize the solvers. The effort of looking into compiler specific instructions and architecture specific programming has resulted into optimal codes for this triangle to triangle minimum calculation. In practice the SIMD speedups show that both penalty and brute force solvers are suitable but in terms of performance penalty is more favorable, while brute force benefits its numerical robustness. Overall because of the reduced errors and tuning through experiments and testing in addition to performance, the penalty method is the best method to use using either ISPC or Intel compilers and up to eight core concurrency.

3.1.3 Shared Memory (SMT)

We introduce three levels of shared memory parallelisation on a single machine that exploit multiple levels of locality. The hierarchical categorisation of the methods is based upon the abstract algorithmic distance between shared resource parallel utilisation and actual arithmetic computation. At the highest level we exploit data access independence of vertex touches we assign cell block tasks to threads. Within each vertex touch particle pairs are grouped into aligned memory blocks thus tasks are launched between particle-to-particle comparisons. Lastly within each particle pair at the innermost level tessellation elements of the mesh are dichotomised and parallel locality is held for the underlying vectorized contact solver.

Three levels of multicore parallelisation:

a) cell level cell-to-cell thread units

b) cell vertex level particle-to-particle thread units

c) particle level mesh-to-mesh thread units

Respectively, we have three parameters to decompose the simulation domain. *Rmin*, *Rmax* parameter for max and min radius of particles, m for mesh density, p for number of particles in the domain. These allow observation of the interplay between performance and particle geometry and dynamics within a grid in a modern NUMA architecture. For scaling measuring we use the hybrid and the brute force solvers.

For the experiment the initial space configuration of the particles in respect to the grid is homogeneous and aligned per cell. Initial conditions for the particle dynamics are governed by only gravity and termination is defined by the number of iteration that the granulates are rest condition with energy dissipated on the ground. Termination phase is known a priori. We utilise regular grid and reluctant-adaptive grids for space decomposition and and grid adaptivity to granular kinematics.

Beginning at the innermost tessellation-level with a regular grid (figure **??**) on a NUMA ivy bridge system the code scales linearly up to the eight-core single die. As the size of the problem increase for weak scaling (figure **??**) the computation to memory bandwidth rate ratio is no longer sustainable computation to scale over multiple cores. Similarly on a small computational domain (figure **??**, blue) with minimum weighted threads grid adaptivity doesn't pay off as thread initialisation, scheduling, grid adaptivity and memory access overhead dominates over arithmetic intensity. Contrary to regular grid, adaptivity allows up-scaling towards both increasingly larger computations (figure **??**) and number of cores as long as bandwidth, compute bounds are not reached or thread overheads are not the bottleneck. In all cases the brute force solver does not scale pass the second die interconnect as communication to memory stagnates.

Arithmetic, bandwidth intensity and data access patterns vary among solvers in the innermost level and can yield different performance results. The hybrid solver is parallelised

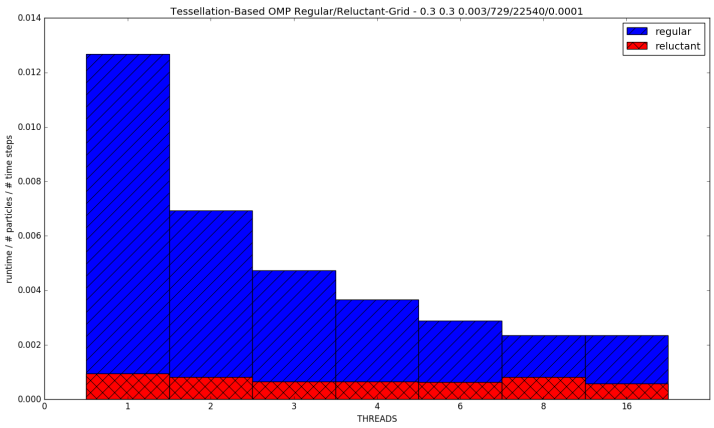

Figure 6: Grids shared memory scaling 22540 triangle elements (BF solver).

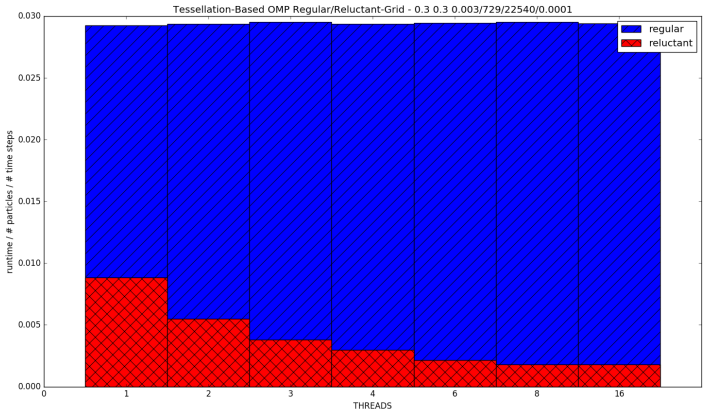

Figure 7: Grids shared memory scaling 98540 triangle elements (BF solver).

in two different hybrid schemes triangle-to-triangle pairs and triangle-to-triangle batches, both launch different types of threads. For the hybrid-on-batches solver the tessellation-based threads are n triangle batches wide while hybrid-on-triangles solver relies on fine couples of threads. 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 don't have a performance gain but they rather create scheduling overhead due to error distribution in triangle pairs and due to the granularity of our batches. For the hybrid-on-triangles that is also the case because although the arithmetic intensity is dense in the solver, it doesn't last long enough to significantly overlap the cost of threading overhead.

Figure **??** hybrid-on-triangle and hybrid-on-batches scales up to eight cores on the single dice but also neither gain from offloading to the second socket. In practice hybrids

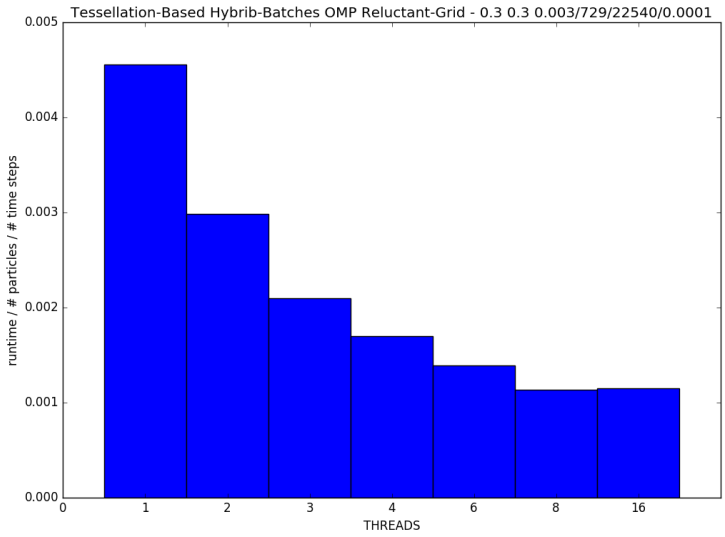

Figure 8: Triangle based shared memory running hybrid-on-batches (HBatches).

normalized time to solution is significantly greater than brute force as seen in Figure **??**

At the outer parallel blocks, particle blocks are threaded within each grid-vertex touch. In Figure **??** brute force does not scale at all and that is due to the adaptive domain decomposition and not due to the solver. The aggressive particle decomposition per grid cell since the preconditioning of the simulation creates a access pattern contradiction to the dense tile access patterns that tessellation method exploits. The overhead and the precondition enforced by the grid for minimum number of particles per vertex doesn't allow any scaling to occur on any method. Particle density is too low for thread launching and compute-wise scaling, memory communication overtakes the simulation time.

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 **??** 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 12) we see that cell-based parallelism scales and it enhances execution time.

Overall shared memory parallelisation yield good speed ups both for hybrid and brute force using adaptive grids with triangle-based parallelism show good time-to-solution For large problem sizes the combination of cell-based plus triangle-based parallelism using hybrid-on-batches is the preferred option. Additional speedup can be gained if spheres are used as a filtering bounding box stage to our triangle-to-triangle contact detection.

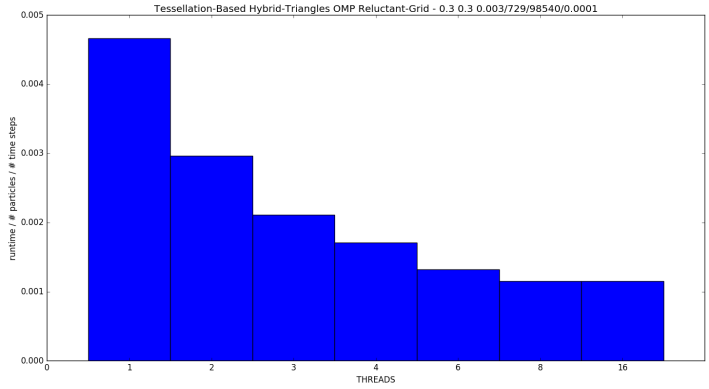

Figure 9: Triangle based shared memory running hybrid-on-triangle-pairs (HTriangles)

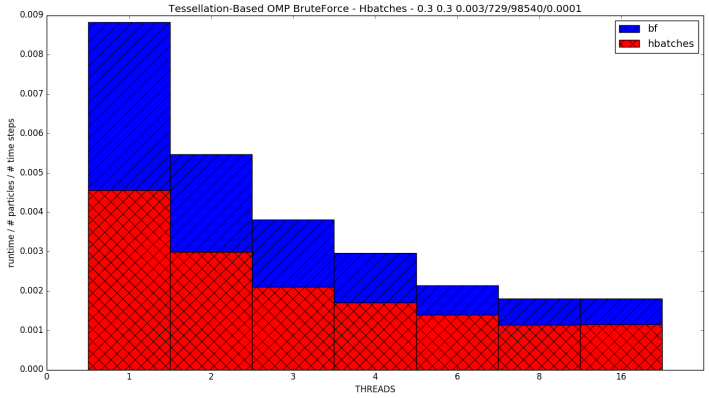

Figure 10: Triangle based shared memory running hybrid-on-triangle-pairs (HTriangles)

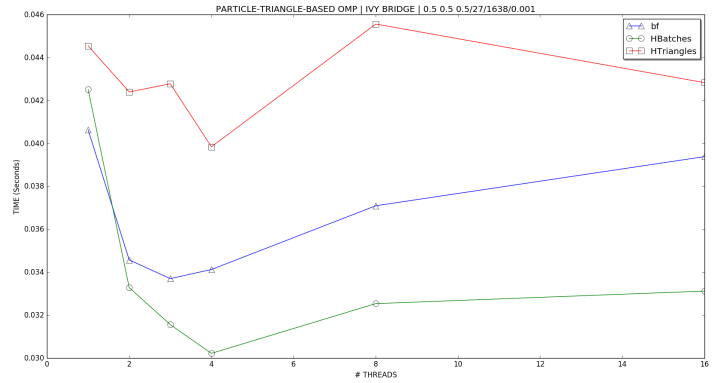

Figure 11: Particle based nested shared memory brute force (bf)

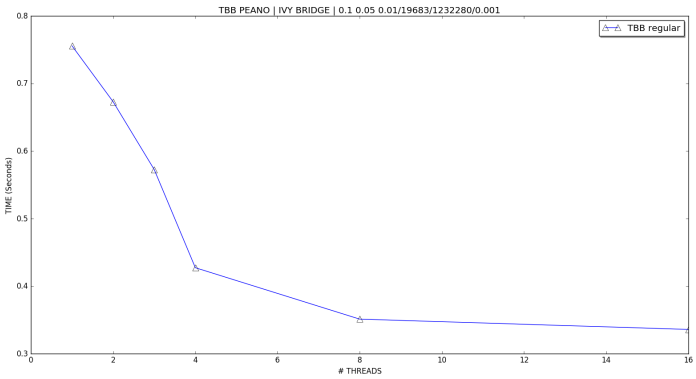

Figure 12: Cell based parallelism on Peano compared to serial runs using Intel TBB

3.1.4 Distributed Memory (SPMD)

We further scale the computation with the Message Passing Interface (MPI) [**?**] for distributed memory computation. Our DEM-based contact detection is inspired by molecular dynamics simulations, where millions of spheres are simulated. State-of-the-art MD computation is decomposed into smaller computational sub-domains and communication exchanges necessary boundary information. Similar computational stages are applied by us in the DEM-based contact detection algorithm. We study synchronous and asynchronous modes of data exchange and how two communication strategies can be exploited to increase performance and minimize computation.

The distributed memory DEM contact detection algorithm is performed in four stages. At stage one, data migration assigns the sub-domains to MPI instances. At stage two, we compute the distances to determine contact between particles. In stage four we accumulate the data and in stage four we perform the explicit time step integration.

The number of neighbours has severe implication on communication patterns between processes. It is an open question what are the performance implications on uniform spaced octree-based decomposition versus the non-uniform kd-tree based decomposition. It is an area of investigation since on an octree-based approach information about the level of refinement is known a priori by the sub-domain boundary size, which is an interesting application for multiscale simulations.

The triangles that overlap into a remote sub-domain due to the decomposition cuts are temporary copied to one or more sub-domains in order to perform a complete contact detection. In this section, we study MPI communication characteristics when we resolve data dependencies between the sub-domains caused by ghost triangles in order to maximize performance of distributed contact detection. We explore the implication of inter-process communication and local computational performance using two communication strategies that use asynchronous non-blocking communication.

Algorithm 5 Naive Asynchronous Data Exchange Pseudocode

1. Load balance triangles
2. Migrate triangles to MPI network using blocking communication
3. Initiate neighbourhood all-to-all asynchronous MPI send/receive
4. Wait for neighbourhood asynchronous communication to terminate
5. Contact detection
6. Derive contact forces from contact points generated
7. Explicit time integration

When spatial decomposition finishes we migrate the data to the processors (Algorithm 5 line 2) with blocking synchronous communication. At each time step the triangles migrate according to the DEM kinematics. In addition to migration, a local area data exchange is required to communicate the boundaries of the sub-domains that cut triangles at the boundary.

The first strategy exchanges local data to all neighbours (Algorithm 5 line 3). The goal is to utilize the communication bandwidth while minimise communication administration overhead. If the exchange does not reach the upper bandwidth limit then exchange of all

data is faster than filtering out the ghost triangles, i.e. doing any preprocessing that finds out which triangle from a sub-domain might be required from a neighbour. Alternatively, we can send out only triangles that overlap from one sub-domain into another sub-domain. This filtering of triangles is an $O(N)$ operation. As soon as MPI communication finishes, the algorithm invokes the existing contact detection routines exploiting vectorised floating point operations with a single contact detection routine. Exchange of all local data to all neighbours significantly increases the number of triangles to be processed from T_{local} to $(T_{local} * N_{ranks} * T_{remote})$ where N_{ranks} is the number of neighbours of neighbours. The increase of triangles to be checked increases the total computation performed locally because of the redundant triangles.

The disadvantage of such a naive method is that total MPI wait (figure 13) for an all-to-all neighbour data exchange increases with the number of processes. The asynchronous communication wait time results to time wasted with idle processors. The method is potentially useful with decomposition schemes where all data exchange can be processed in the background, i.e. enough bandwidth is available.

Figure 13: Waiting time (s) per MPI rank/node for all to all neighbour data exchange over 1000 time steps (25 mil triangles, 10k non-spherical particles).

Algorithm 6 Overlapping Asynchronous Data Exchange Pseudocode

```
1 Load balance triangles
2 Migrate triangles to MPI network using blocking communication
3 Search overlapping ghost triangles to send
4 Initiate neighbours asynchronous MPI send/receive
5       Local contact detection
6       Retrieve required ghost triangles from neighbours
7       Local against to remote ghost triangle contact detection
8 Wait for neighbourhood asynchronous communication to terminate (No Real Wait)
9 Derive contact forces from contact points generated
10 Explicit time integration
```

The second strategy filters out local ghosts from the data structure and sends them to

the overlapping neighbouring processes. Using the spatial decomposition information, we find the specific processes/boundary cells that the triangle bounding box overlaps. In this strategy we minimize data exchange at the cost of a filtering overhead and the allocation of buffers in memory. The method aims to minimise the waiting time of MPI processes by overlapping concurrent computation over communication. As shown in Algorithm 6 line 5 local contact detection is executed as soon asynchronous MPI communication is initiated overlapping communication. A second contact detection is initiated to determine contact between local and remote ghost triangles at line 7. The strategy is advantageous as neighbour waiting is always zero as long as local contact detection takes longer than the transmission of the data. As shown in Figure 13 the waiting time is increased proportionally to the number of MPI ranks, enabling us to increase overlapping computation per process with larger number of processes.

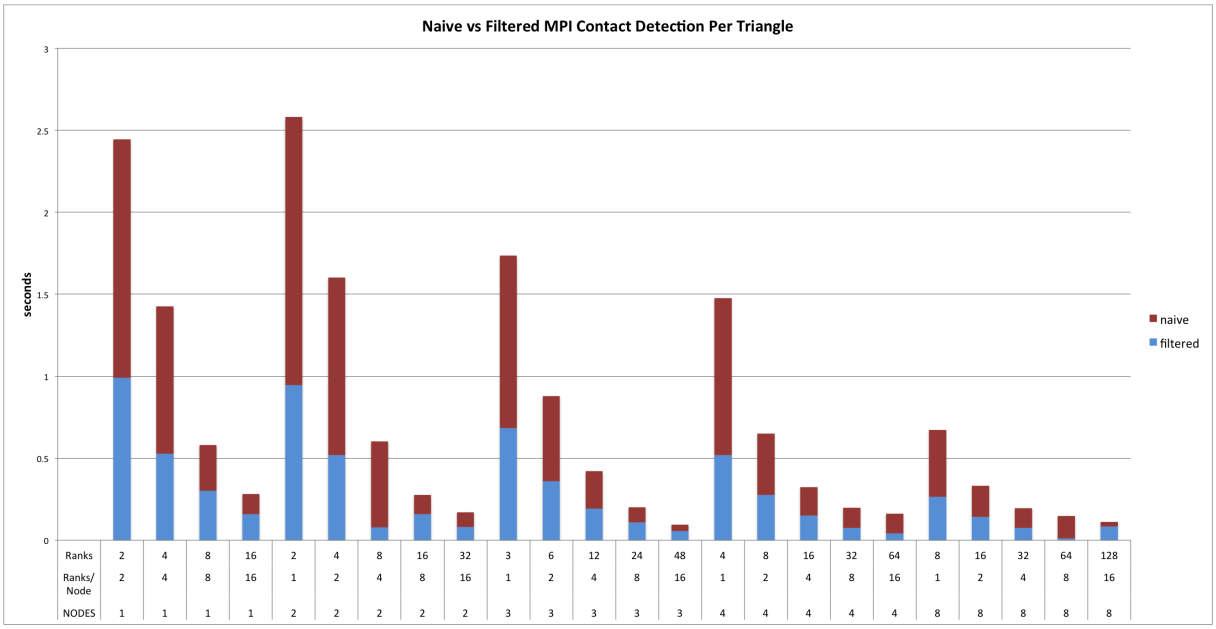

Figure 14: Normalized naive vs filtered contact detection performance per triangle pair running on multiple nodes.

We measure both strategies to determine the performance of normalised computation per triangle. The result in Figure 14 shows for each rank and compute node on the x axis, the time required to solve the distance of a pair of triangles. The blue bars show the average normalised time for the second strategy to process a triangle (filtering approach). The red bars show the difference between the overall naive communication time and the filtered method. The time is reduced linearly as the number of ranks increase.

For the measurement set-up in Figures 13 and 14, we use Durham University Hamilton supercomputer that has per node 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.

On distributed memory the number of neighbours has severe implication on communication patterns between processes. It is an open question what are the performance

implications on uniform spaced octree-based decomposition versus the non-uniform kd-tree based decomposition. It is an area of investigation since on an octree-based approach information about the level of refinement is known a priori by the sub-domain boundary size, which is an interesting application for multiscale simulations.

3.2 Case Study

3.2.1 Hopper Flow

3.2.2 AGR Seismic Shake

4 Timetable

5 Critical Analysis

5.1 Open Issues

5.2 Contribution

5.3 Overall Assessment

- Completed since last report: Implemented domain decomposition for distributed memory load balancing using library, MPI blocking communication for data migration.
- New concepts: Overlapping process communication for contact detection with ghost data. Asynchronous communication waiting time significant as number of ranks increase.
- Open questions: multilayer grid/spacetree implementation for DEM

We implement a new fast but also robust hybrid algorithm that exploits modern CPU architectures. In addition, we study different schemes of domain decomposition in literature. I implement the state-of-the-art spatial domain decomposition and load balancing using Zoltan. Manual migration has been implemented using blocking MPI communication. We implement an asynchronous communication for data exchange of ghost data and compare two possible strategies. Finally based on benchmark measurements we pick the best performing scheme that overlaps computation, increases performance and decreases data exchange volume.

The next stage of my research is to investigate contact detection strategies for large scale supercomputing on a multiscale setting using adaptive multiscale grids. I'm also interested in vectorized memory layout implications on MPI buffered communication. Lastly, the source code will be generalized to form a open source library that any simulation tool can use.

According to the project plan of the previous report, the work packages have been completed as projected.

5.4 Acknowledgments

This work has been sponsored by EPSRC (Engineering and Physical Sciences Research Council) and EDF Energy as part of an ICASE studentship. This work 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. All underlying software is open source and available at: https://github.com/KonstantinosKr/delta.