# Chapter 4

# Further studies of the spot model

## 4.1 Do voids exist?

Other models of granular and glassy materials often make use of the concept of particle-sized voids in the material. Motion can then be described by a single neighboring particle jumping into the free space. As a test of these theories, it is interesting to ask whether such voids actually exist. In experiment, confirming or disproving the existence of these is difficult, but in the DEM and spot simulations this can easily be carried out.

A code was written to analyze the free space size distribution for the granular packing. The entire container is first covered with a fine grid with spacing $\lambda = \frac{1}{50}d$, and at each point on the grid the maximum size of sphere that can fit there is calculated. Of course, we may miss larger voids which are centered at points not on the grid. However, by geometrical considerations, we see that if the maximum radius found on the lattice is $r$, then the maximum radius for a void anywhere in the packing is bounded above by $r + \sqrt{3}\lambda/2$.

For the initial random packing, the maximum size of sphere that could be found had radius $0.466207d$. Using the above calculations the absolute maximum size for a void in the packing is $0.483527d$, and thus no particle sized voids exist in the packing.

Furthermore, of the 36 regions in the packing where a void of radius $0.4d$ or greater can be fitted, 31 of these lie within a particle diameter of the walls, and of

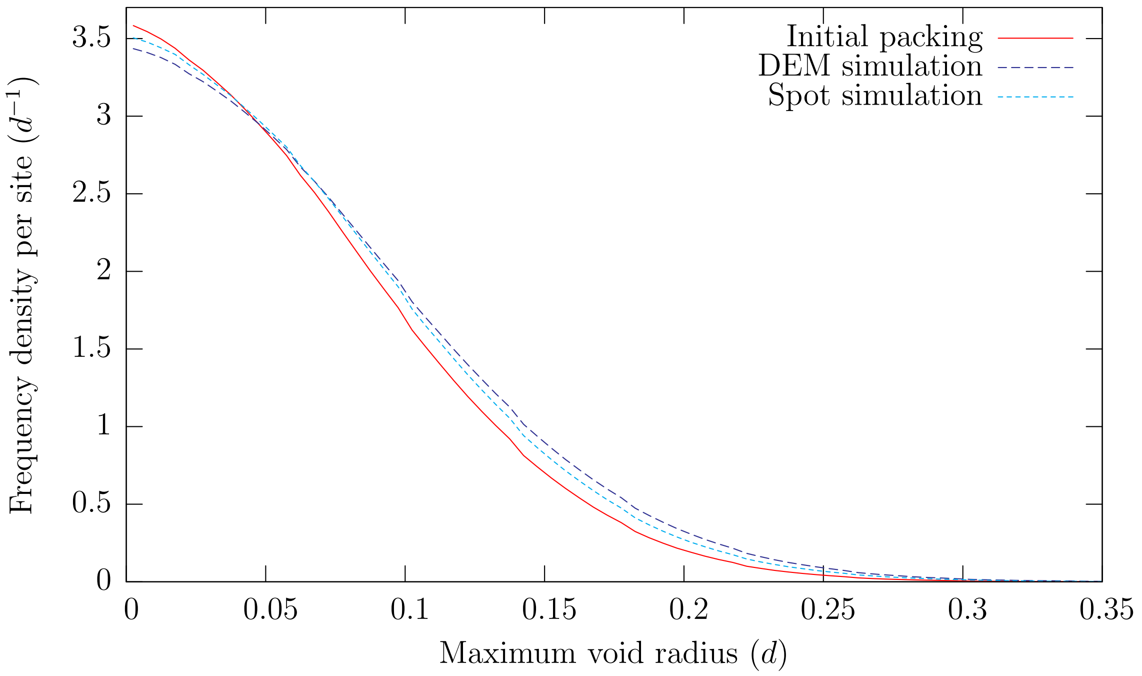


Figure 4-1: Distribution of radii of free spaces for the DEM and spot simulations, time averaged over the interval $80\tau \leq t < 100\tau$.

| Time ($\tau$) | 80 | 82 | 84 | 86 | 88 | 90 | 92 | 94 | 96 | 98 |
|---|---|---|---|---|---|---|---|---|---|---|
| MD | 0 | 3 | 2 | 0 | 2 | 1 | 2 | 1 | 1 | 1 |
| Spot | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 4.1: Number of particle-sized free spaces present in the spot and DEM simulations for ten simulation snapshots starting from $t = 80\tau$.

the remaining five, the largest void that can be fitted has radius $0.421792d$. Thus in a fully three dimensional granular packing, the likelihood of finding significant areas of empty space may be even less.

The above code was also applied to analyze the free space size distribution in the flowing random packings generated by the spot and DEM simulations. Since this requires testing many frames, the grid spacing was doubled to $\frac{1}{25}d$. Also, attention was restricted to the strip $20d < z < 60d$ to avoid orifice and free surface effects. Furthermore, to reduce lattice effects, only voids whose nearest contact was with a particle and not a wall were counted.

Figure 4-1 shows the distribution of hole sizes in the DEM and spot simulations during the flow. The vertical axis has been scaled to represent the frequency density

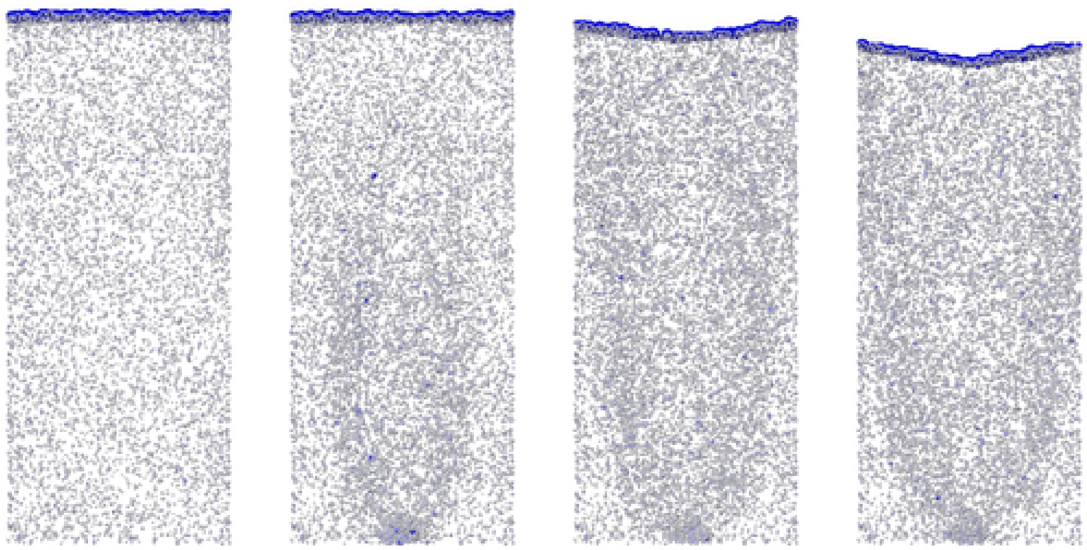


Figure 4-2: Free space plots for the DEM simulation for $t = 0, 30\tau, 60\tau, 90\tau$ showing all voids with radius $d/4$ or larger. Small voids are plotted in white, larger voids are plotted in blue, and the blue line at the top represents the free surface. Images generated using *Raster3D* [86].

of finding a hole of a given size per test site. Thus, the total area under the graph is approximately 40%, which is in rough agreement with proportion of free space in a typical granular packing.

As expected, the flow causes a general increase in the size of free spaces in the packing. This alteration in the distribution takes around twenty-five frames to occur, and remains roughly in equilibrium from then on. During the flow it is also possible to find voids in the material that are more than particle in radius – table 4.1 shows the number of such for ten simulation snapshots starting from $t = 80\tau$. Compared to the amount of flow, the number of particle-sized empty spaces is extremely small, providing further evidence that the void-based models are a very poor description of the microscopic particle dynamics. Also, every single void in the table was located either at the front or back wall of the container, suggesting that their effect may be even less in a fully three dimensional simulation.

Since the position of free space is very important to the spot model, it was decided to go further and explicitly track its position during the flow. The previous program was modified to dump the positions of all voids that have radii greater than $d/25$

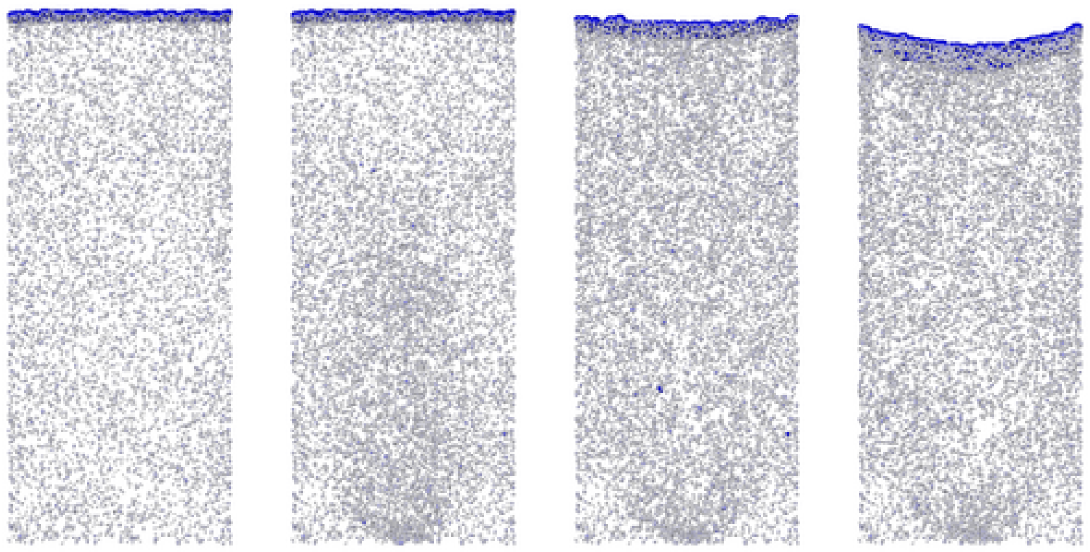


Figure 4-3: Free space plots for the spot simulation for frames $t = 0, 30\tau, 60\tau, 90\tau$.

to a file. These were then rendered, coloring small voids white, and larger voids progressively more blue. The results for four frames covering the transition from the static packing to the steadily flowing state are shown in figures 4-2 and 4-3 for the DEM and spot simulations respectively.

In both simulations, we see that the amount of free space increases first in the neighborhood around the orifice before spreading out into a roughly parabolic region, as expected. However, one very interesting feature of the DEM simulation is that the free space tends to be concentrated in two bands to either side of the orifice. This feature is completely absent in the spot simulation – according to the spot model, there should be a direct correspondence between free volume and downwards velocity. The result suggests that free volume may be better associated with shear, and a more complete and quantitative comparison of density fluctuations in the two models is considered in the following section.

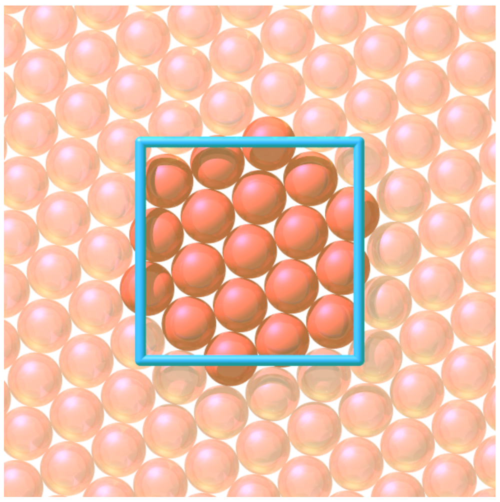
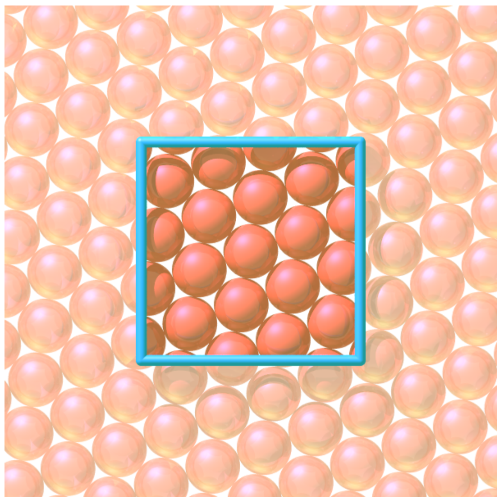
(a)  (b) 

Figure 4-4: Two simple methods of estimating packing fraction in a two dimensional packing, based on a $4d$ square test region. In (a), the packing fraction is computed by counting up the volume of all particles whose centers are in the test region, and dividing by the square's area. In (b), particles which overlap the boundary of the region contributed only their volume which is within the square.

## 4.2 Computing packing fraction using a Voronoi tessellation

### 4.2.1 The problems of accurately tracking free volume

While the snapshots shown in figures 4-2 and 4-3 were originally created to search for voids, they suggest that the distribution of free volume in the spot and DEM simulations may be considerably different, and since spots carry free volume, and it would be beneficial to accurately track this quantity. Unfortunately, accurately measuring the distribution of free volume in the simulation is difficult, since the changes in packing fraction may be very small. In a monodisperse static granular material, the structure will be similar to a Random Close Packing (RCP) of spheres, with a packing fraction of 63%. During flow, this may decrease to approximately 58%, a change of only 5%. If we were interested in a time-averaged packing fraction, or the packing fraction in a large spatial region, then we could get reasonably accurate results on this scale by averaging over a large amount of particle data. However, to potentially track spots, we would like to measure the instantaneous packing fraction in regions on the same size as the spot, and given the particle discreteness at this level, there is not enough data to achieve an accuracy level of less than 5%.

To illustrate this, consider the two dimensional regular hexagonal packing of particles shown in figure 4-4, where the orientation of the packing has been offset by 10° from the vertical. The packing fraction of this lattice should be a constant, with value

$$\frac{\pi(d/2)^2}{6 \times d \times (d/\sqrt{3})} = \frac{\pi}{2\sqrt{3}} = 90.689968\%.$$

Suppose now that we wish to estimate the packing fraction based on the particles in the blue square, which has side length $4d$. A first attempt would be to count up the number of particle centers which lie within the box (as shown in figure 4-4(a)) and then estimate the packing fraction by dividing the area of those particles by the area of the square. Unfortunately this leads to very large errors: by moving the box around by small amounts on the lattice, it is possible to have between 17 and 21 particle centers within the box, leading to a packing fraction estimate between 83% and 103%. A second approach is shown in figure 4-4(b). In this scheme, particles which are at the edges of the square contribute only their area which is within the square to the density estimate. This gives better accuracy, but as shown in figure 4-5 can still give errors on the order of 1%.

It is possible to remove this error by making use of Voronoi cells. In a given packing, the Voronoi cell [140] for a particle is defined as all the volume which is closer to that particle than any other. Using this prescription, it is possible to define a packing fraction at the level of a single particle, as the ratio of a particle's volume to the volume of its Voronoi cell. A local estimate on the scale of a spot can be obtained by averaging over the particles and cells in a small region.

In two dimensions, the Voronoi cells are polygons, the sides of which are the perpendicular bisectors between neighboring particles. In the current example, the Voronoi cells would be regular hexagons. Thus, as shown in figure 4-6(a), the packing fraction could be computed by summing up the volume of all the Voronoi cells in a small region. Regardless of precisely how the cluster of particles is chosen, the method

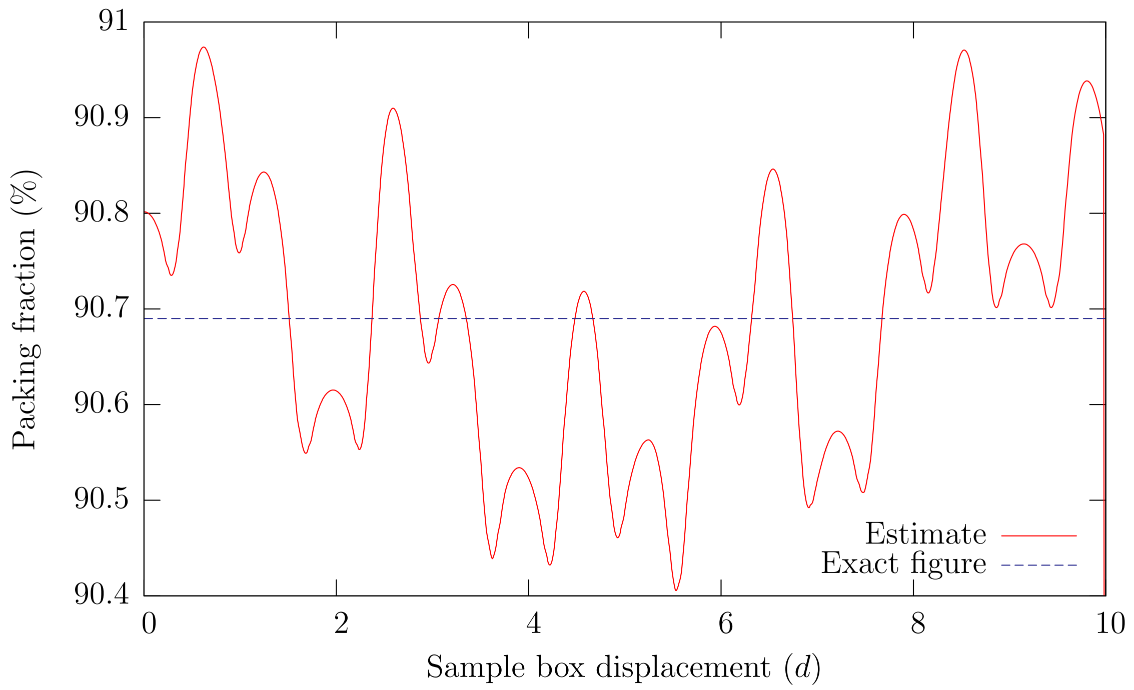


Figure 4-5: Graph showing how the local density estimate would change if the test region in figure 4-4(b) is continuously moved upwards.

will always give the exact answer for the packing fraction of $\pi/2\sqrt{3}$.

Although this technique was demonstrated on a regular packing, it will also work on a random packing as well (figure 4-6(b)). The method seems theoretically advantageous, since to estimate packing fraction, it respects the packing structure, rather than cutting particles.

### 4.2.2 Computation of three-dimensional Voronoi cells

The computation of Voronoi cells is a well-studied problem, particularly in the computer science literature [11]. Several methods are readily available, and one is built into the popular mathematical package Matlab, which makes use of the dual Delaunay triangulation, computed via Qhull [3, 14]. However, to provide greater control and flexibility, it was chosen to create an algorithm specifically for the computation in granular simulation, that could directly compute cells individually.

In a three dimensional random packing, the Voronoi cell of a particular particle will be a convex irregular polyhedron, the faces of which are the perpendicular bisectors of

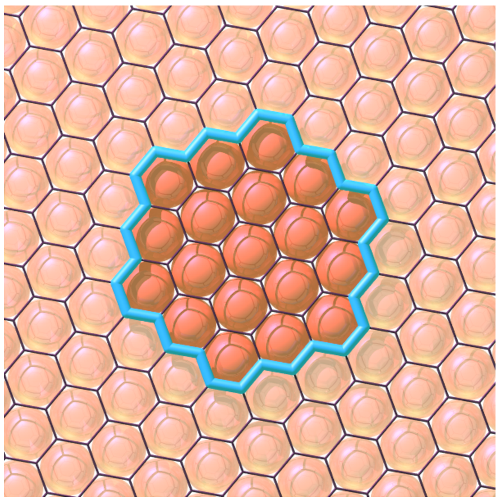
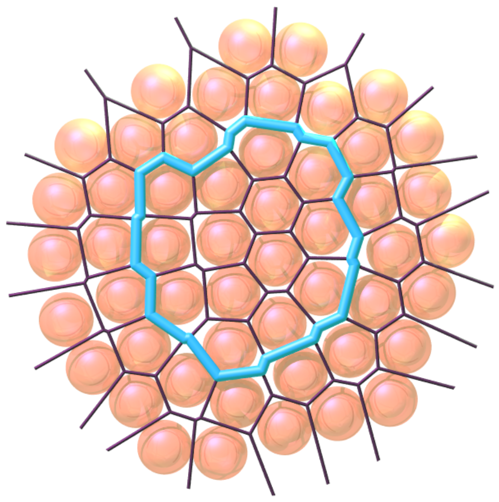
(a)  (b) 

Figure 4-6: The density of the local region can be accurately found by computing the Voronoi cells of all the particles, and then dividing the particle volume be the Voronoi cell region in a small cluster. For the two dimensional hexagonal packing, the Voronoi cells are all regular hexagons, and the computation yields the exact packing fraction of $\pi/2\sqrt{3}$. The algorithm can also be applied to an arbitrary amorphous packing (b), in which case the Voronoi cells will be irregular polygons (in two dimensions) or polyhedra (in three dimensions).

its neighbors. While in practice, we expect the Voronoi cells will be reasonably simple polyhedra, in a general situation, there is no theoretical limit on how complicated they may be. For example, consider a single particle surrounded by a densely packed spherical shell of particles, all at some large radial separation $R$. The Voronoi cell will be approximately a sphere with radius $R/2$, with many facets due to the particles in the shell. As $R$ increases, the number of facets will increase without limit. While a Voronoi cell of this type may be unlikely to occur in practice, it highlights the fact that our algorithm cannot make assumptions about the form of the polyhedra – we must be able to handle completely arbitrary convex polyhedra.

To compute a Voronoi cell for a particle with position $\mathbf{x}_p$, the algorithm takes the following approach:

1. Surround the particle with an initial Voronoi cell consisting of the entire container volume.

2. Set $r_{\text{test}} = \frac{d}{2}$.

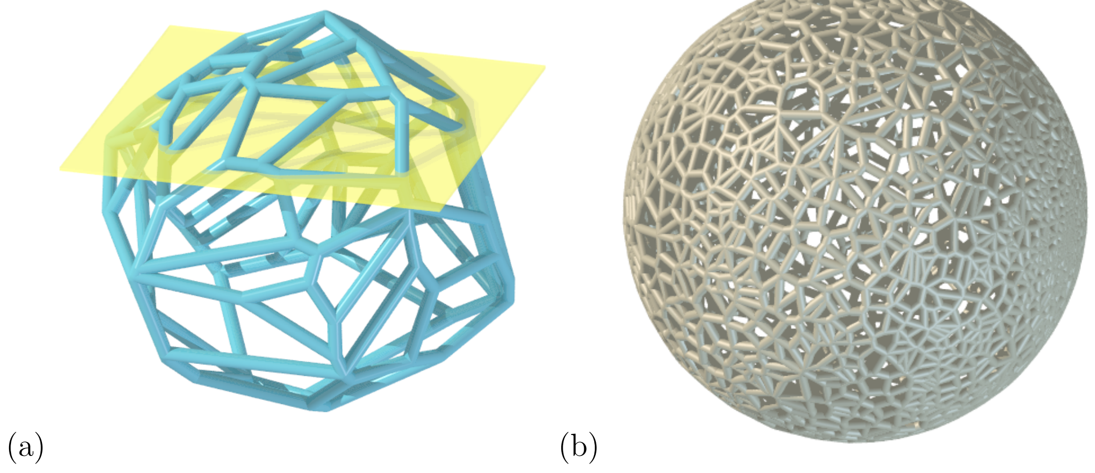


Figure 4-7: The basic computational challenge in computing three dimensional Voronoi cells is to take an irregular polyhedron, and recompute its vertices based on cutting off a plane (a). The algorithm developed in this thesis can be tested on arbitrarily complicated polyhedra, such as this approximation to a sphere, made by cutting many planes all of unit distance from the origin (b).

3. Find all the neighboring particles with positions $\mathbf{x}$ in the spherical shell defined by $r_{\text{test}} < |\mathbf{x}_p - \mathbf{x}| < r_{\text{test}} + d$.

4. For each of these particles, cut the Voronoi cell by the plane which is the perpendicular bisector between $\mathbf{x}$ and $\mathbf{x}_p$.

5. Compute the maximal distance $R$ of a vertex of the Voronoi cell to its center $\mathbf{x}_p$.

6. If $2R > r_{\text{test}}$, then increase $r_{\text{test}}$ by $d$, and go back to step 3. If $2R \leq r_{\text{test}}$, the computation is complete.

By looping over concentric spherical shells, we can efficiently test those particles nearest $\mathbf{x}_p$ which will most likely create the facets of the Voronoi cell. When we know that all particles which are left in the packing are further than $2R$ from the particle, then we can be sure that the cutting planes of those particles (which at the very least, are a distance of $R$ from $\mathbf{x}_p$) would not intersect the Voronoi cell, and thus do not need to be tested.

The remaining computational challenge comes from item 4, which is shown schematically in figure 4-7(a): we must be able to computationally represent an arbitrary convex polyhedra, and be able to recompute the polyhedra based on cutting off an arbitrary plane. To represent the polyhedra, three pieces of information were stored:

- A list of polyhedra vertices $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$.
- A table of edges: for each vertex $i$, store the three connected vertices $e(0,i)$, $e(1,i)$, and $e(2,i)$. The three edges are stored in a handed order, so that they trace around the vertex according a right hand rule with respect to an outward-pointing normal.
- A relational table describing how vertices are connected back to each other: for each vertex $i$, store three additional numbers $l(0,i), l(1,i), l(2,i)$ that satisfy the property
$$e(l(j,i), e(j,i)) = i.$$

The first and second items are obvious requirements, although it should be noted that in this representation, vertices always have three edges. The case of four or more edges (as would occur in a octahedron or icosahedron) is treated as degenerate, since it requires that the random cutting planes precisely intersect with an existing vertex. In practice this is not a problem, since due to numerical inaccuracy, any intersection between four edges is represented by two proximate, connected vertices.

The third requirement listed above does not appear obvious, but is useful during computation. A common task during the plane-cutting involves breaking and reforming edges. Suppose we consider a vertex $i$, that is connected to a vertex $j$, so that $e(\lambda, i) = j$. We know that one of the edges of vertex $j$ points back to vertex $i$, but without the relational table, we do not know which one, without searching over the three elements $e(0,j), e(1,j), e(2,j)$. The relational table tells us which edge this is immediately.

Given this representation of a polyhedron, we now consider cutting it by an arbitrary plane, $\mathbf{x} \cdot \mathbf{n} = a$. We want to remove any part of the polyhedra which intersects the half-space $\mathbf{x} \cdot \mathbf{n} > a$.

The algorithm is described in detail below. The process starts by picking a vertex, and moving across the polyhedra to search for an edge which intersects the plane (items 1 to 6). To do this, the property of convexity is exploited. If the plane intersects the cell, then one of the three neighbors to a vertex will always be closer to the plane; the only time this fails is if the cell and the plane are disjoint. Once an intersected edge is found, the algorithm traces around intersected facets to find all intersected edges (items 7 to 9). On each intersected edge, a new point is created at the intersection point, and these points are then connected to create a new facet (items 10 to 11). The algorithm then deletes all the old vertices which were in the removed half-space $\mathbf{x} \cdot \mathbf{n} > a$ (items 12 to 13).

1. Pick an arbitrary vertex $i$.

2. Let $a_i = \mathbf{x}_i \cdot \mathbf{n}$. If $a_i > a$, skip to step 5.

3. Compute $a_j = \mathbf{x}_j \cdot \mathbf{n}$ for $j = e(0, i), e(1, i), e(2, i)$. If all three computed values are less than $a_i$, the half-space does not intersect the polyhedron; exit. Otherwise, pick a $j$ such that $a_j > a_i$.

4. If $a_j > a$, skip to step 7. Otherwise redefine $i$ to be $j$, and go back to step 3.

5. Compute $a_j = \mathbf{x}_j \cdot \mathbf{n}$ for $j = e(0, i), e(1, i), e(2, i)$. If all three computed values are all more than $a_i$, the polyhedron lies wholly within the half-space; exit. Otherwise, pick a $j$ such that $a_j < a_i$.

6. If $a_j > a$, redefine $i$ to be $j$, and go back to step 5. Otherwise, swap $i$ and $j$.

7. We have now found an edge, between $i$ and $j$, that intersects the cutting plane, so that $a_i < a < a_j$, meaning that $i$ is not in the half-space to be removed, and $j$ is. Consider an arrow from $i$ pointing $j$.

8. Trace around the facet to the right of the arrow, by following the arrow and going right at every vertex, until reaching a point which lies outside the half-space. This point is on an intersected edge.

9. Starting from this intersected edge, trace around the next facet, until another intersected edge is found. Repeat this process, finding all the intersected edges, until returning to the edge between $i$ and $j$. Define $(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)$ to be all the intersected edges, where all the $i$ vertices lie outside the half-space, and all the $j$ vertices lie inside the half space.

10. Define new vertices $k_\alpha$ at the points where the edges $(i_\alpha, j_\alpha)$ intersect the cutting plane.

11. For each $\alpha = 1, \ldots, n$ redefine the edge from $i_\alpha$ to $j_\alpha$ as going from $i_\alpha$ to $k_\alpha$. Connect $k_\alpha$ to $k_{(\alpha+1)}$ for $\alpha = 1, \ldots, (n-1)$, and connect $k_n$ to $k_1$.

12. Search the vertices $j_1, j_2, \ldots, j_n$, and find all the vertices which are connected to these; call these $j_{n+1}, \ldots, j_m$.

13. Delete vertices $j_1, j_2, \ldots, j_m$.

Before using this in the granular simulation, the algorithm was tested by constructing many arbitrary polyhedra. The cell used to make figure 4-7(a) was created using the algorithm. Figure 4-7(b) represents a difficult test of the algorithm, creating an approximation to a sphere by intersecting many planes all of the form $\mathbf{x} \cdot \hat{\mathbf{n}} = d$, where $\hat{\mathbf{n}}$ is a random unit vector. This polyhedron was then intersected with the plane $x = 0$, which removes approximately half of the vertices, and is a good test of the deletion step.

Figure 4-8(a) shows the algorithm results in a local region when applied to the DEM simulation data from figure 3-5. In this picture, the front wall has been rotated to the top, and the Voronoi cells mesh together to form a planar surface. This demonstrates one of the advantages of the direct computation: it is very easy to correctly mesh the cells to walls by cutting with the appropriate planes. Figure 4-8(b) shows an example of this, where cells have been cut to correctly handle a packing at an oblique corner. Figure 4.2.2 is a more complicated example, looking up from below at a conical funnel.

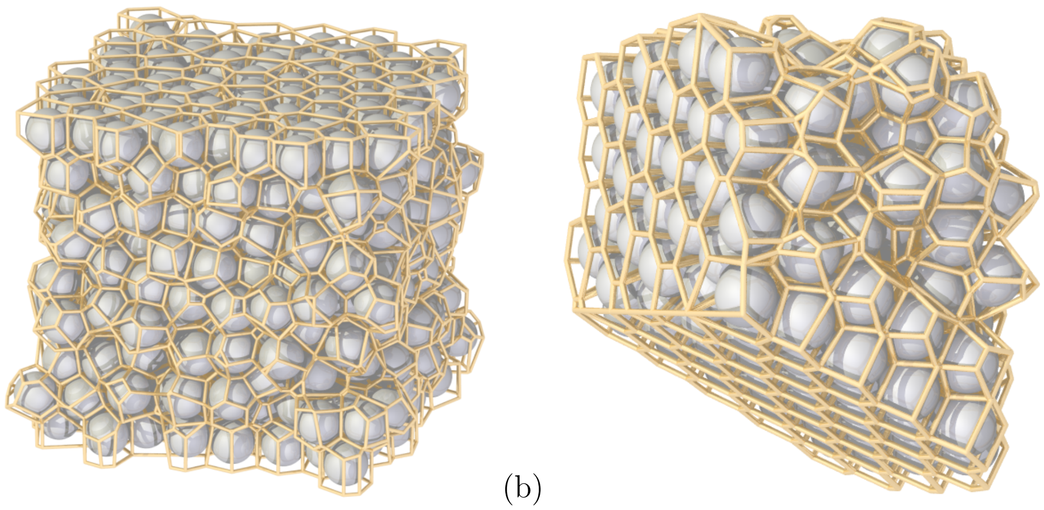


Figure 4-8: A small region of particles and their computed Voronoi cells, taken from the DEM simulation data of chapter 3, and rotated so that the front wall is facing upwards. The plane-cutting algorithm can also be applied to handle packings of particles at oblique boundaries (b).

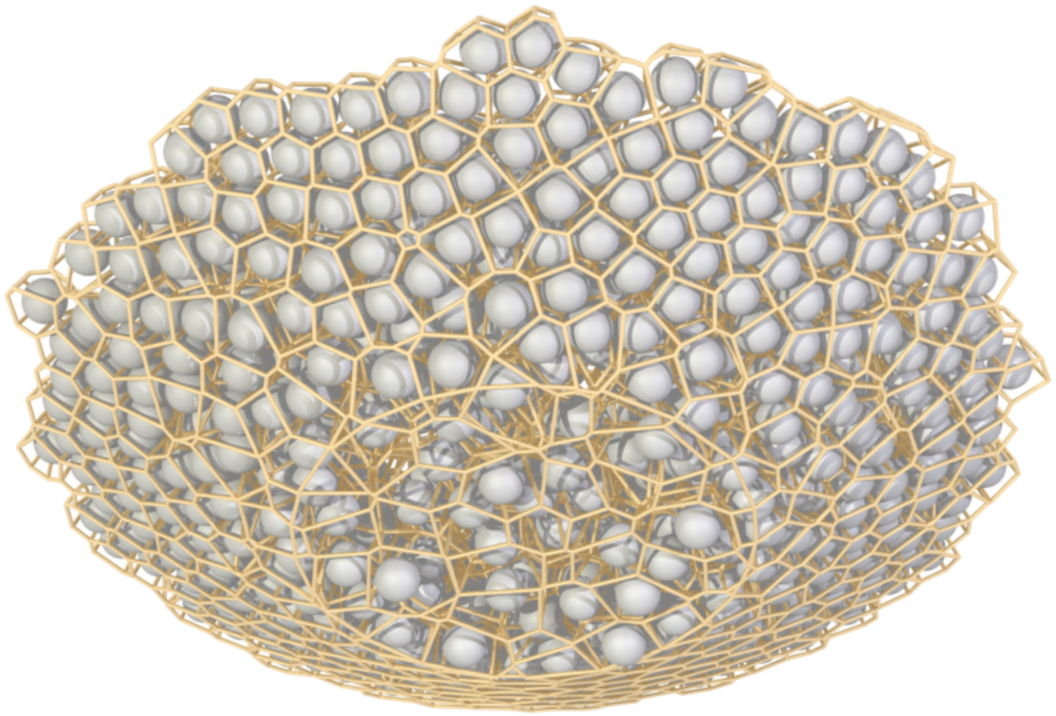


Figure 4-9: An example result of the Voronoi cell algorithm, looking up from below at particles falling out of a conical funnel (using data from chapter 5). For the Voronoi cells next to the wall, the cells are cut by approximating the conical wall as locally planar, and cutting the cell with that plane. This yields very satisfactory results, although small discrepancies are visible, particularly at the circular interface between the cone and the exit pipe.

### 4.2.3 Local density computation

To calculate the local density requires one additional computational step: we must be able to take the computed polyhedra of the previous section and find their volume. This is done by the following procedure:

1. Pick an arbitrary vertex $\mathbf{v}$.
2. Choose a facet, whose vertices are $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$.
3. Compute the volume of the pyramid whose base is the facet, and whose apex is $\mathbf{v}$, by dividing it into tetrahedra with vertices $(\mathbf{v}, \mathbf{x}_0, \mathbf{x}_\alpha, \mathbf{x}_{\alpha+1})$ for $\alpha = 1, 2, \ldots, (n-2)$.
4. Return to step 2, and repeat over all facets.

An excellent test of this algorithm was to compute the Voronoi cells for an entire simulation snapshot, and then check that the total volume of all the Voronoi cells equalled the volume of simulation container. Calculating the Voronoi cells for all 55,000 particles in the container took approximately 9.5 s on a 2GHz Athlon machine. Thus, as a rough guide, the algorithm computes 5000 cells per second although many factors such as system geometry or computer architecture could significantly alter this.

Using this algorithm, the local packing fraction at a point was defined by finding all the particles whose centers were within a distance $s = 2.2d$ of the point. Figures 4-10 show snapshots of local packing fraction in the DEM and spot simulations from the previous chapter. In both simulations, the initial packing fraction is approximately 63%, although we can see that at the boundaries the computed packing fraction decreases, since the Voronoi cells which are pressed against the walls have slightly more volume.

The snapshots highlight very large differences between the packing fraction distributions in the two simulations. The upwards velocity of the drop in density is roughly equal, as would be expected from the calibration of spot velocity. However in the DEM simulation, the drop in density takes place in two bands either side of the central region, which correspond to the particles undergoing the highest amount of shear, and is a direct observation of the well-know concept of shear dilation. The presence of these bands is in good agreement with the spatial distribution of free volume that was seen in 4-2.

In the spot simulation, the largest density drop is located in the central region above the orifice, in the region of highest velocity. This suggests a fundamental problem with the formulation of the spot model up to this point: since spots each move at constant velocity, the only way to make a packing move faster is to send more spots through, resulting in a larger density drop. The above results suggest that density drop may better be linked to shear, and not velocity. This subject will be returned to in later chapters.

## 4.3 Alternative spot models

### 4.3.1 A model of the free surface

Describing the behavior of the free surface of a granular pile has received much attention from the literature. Typically the free surfaces of dry granular materials will form at a critical angle, known as the angle of repose. If perturbed, they will tend to reform at this angle again. This became a useful metaphor for "Self-Organized Criticality", a notion applicable in describing many natural processes [12, 13]. Recently, work on the evolution of the free surface has concentrated on the rotating drum geometry [109, 68, 40, 101, 22].

In granular drainage, the free surface also tends to form heaps with slopes at the critical angle. This can be seen in an hourglass, where the free surface takes the form of an indented cone. Particles on the surface of the cone fall radially inwards to the cone's apex, before moving downwards through the packing. While the spot model was originally proposed in the context of bulk flow, it is interesting to see whether it could be used in any way to describe the free surface also. In the previous simulations the free surface was ignored, and the figures only shown the bulk region, well below

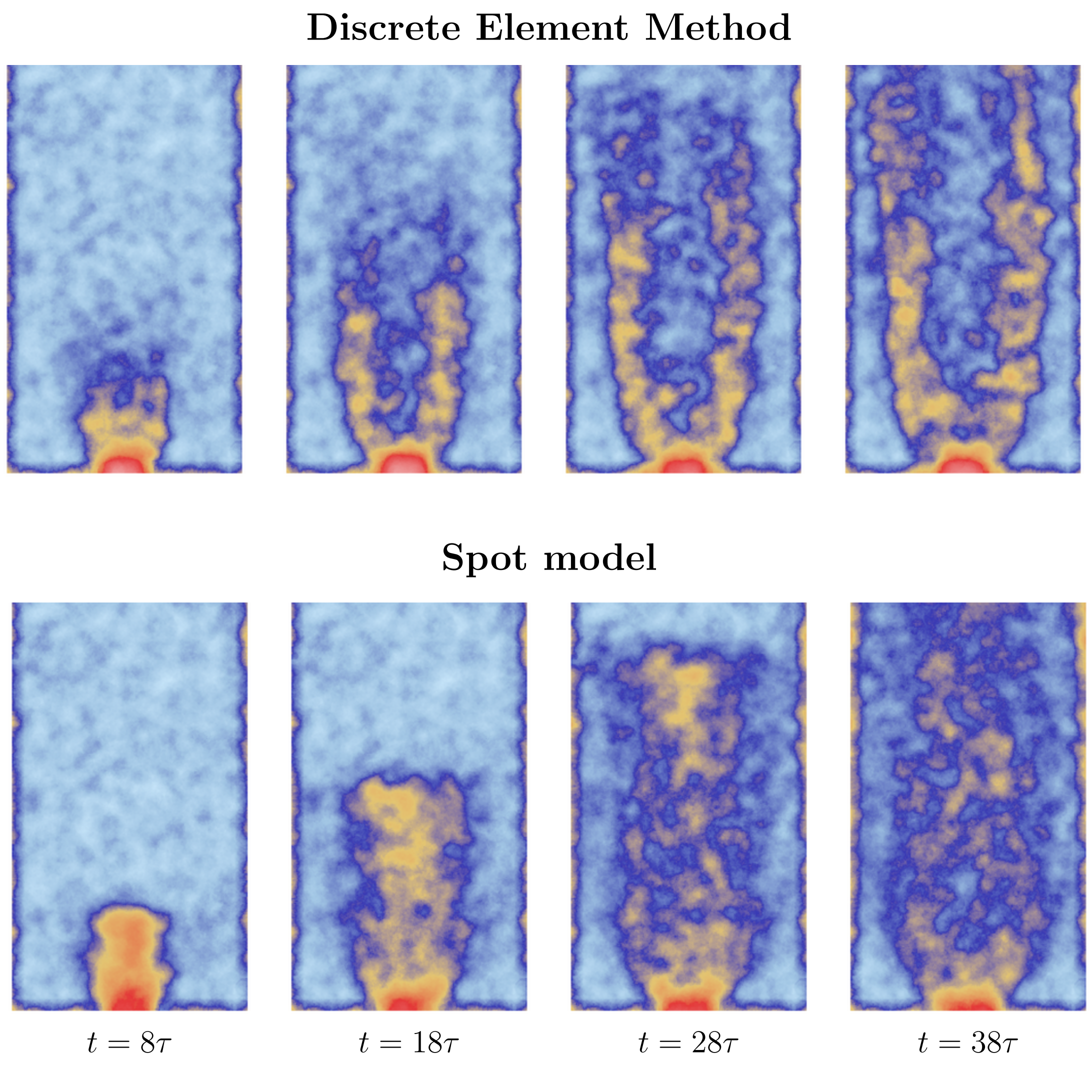


Figure 4-10: Plots of instantaneous local packing fraction computed using the Voronoi algorithm for the two simulations shown figure 3-5. Volume fractions of 50%, 57%, 60%, and 63% are shown using the colors of red, yellow, dark blue, and cyan respectively. Colors are smoothly graded between these four values to show intermediate volume fractions.

the top of the packing. The approach of the previous chapters was for the spots to carry out a completely unbiased random walk, and as shown in figure 4-11, this does not capture the behavior seen in DEM. Particles at the free surface essentially follow the parabolic velocity field in the bulk, which is inappropriate and does not capture the particle avalanching.

In the void model, the evolution of the free surface has been addressed, by proposing a very simple modification to the walk process [25]. In the bulk of the packing, when a void always generally has two particles in the lattice points above it, and in this situation it picks one at random. However, in the case when only one of these two sites is filled with a particle, the void always moves in the direction of the particle. A void is only removed from the simulation when both of the sites above it are vacant. This simple modification suffices to create heaps and avalanching at the free surface as when a void reaches the heap, it travels diagonally upwards along the heap surface.

This, in effect, creates a simple biasing of the random walk: there are two locations it can jump to, and it chooses randomly amongst the available options. This idea can be adapted to the spot model. Suppose that a spot can potentially move to $N$ locations, and it would influence $p_i$ particles if it moved to position $i$. Let $q = \sum_{i=1}^{N} p_i$. If $q = 0$ then remove the spot from the simulation. Otherwise, let

$$\mathbb{P}(\text{Spot moves to } i) = \frac{p_i}{q}.$$

In the bulk, where the density of particles is approximately constant, this does not alter the process by a large amount, but at the free surface, spots will bias their motion to create heaps. As shown in the left snapshot in figure 4-12 this correction dramatically improves the free surface behaviour in the spot model. However, it can also be seen that the packing of the particles at the free surface in this model is unphysical: individual particles are floating, and not in contact with their neighbors. There is no explicit gravity in the spot model, but in the bulk, particles are kept in contact by geometrical constraints from their neighbors. At the free surface, the particle rearrangement during avalanching causes them to separate.

This can be corrected by a further modification of the spot model. In the previous implementation, when a spot moves by $\mathbf{r}_s$, then the particles experience a displacement $-w\mathbf{r}_p$, where $w$ is a fixed quantity. Suppose a spot is going to influence $p$ particles, each of volume $V_p$. If spots are thought of as carrying a completely fixed amount of free volume $V_s$, then another possible approach would be to let $w = V_s/pV_p$, so the spot's influence is divided equally among the particles in range. In the bulk, where the particles are roughly at constant density, this modification has little effect. However, at the free surface, where $p$ is lower, the spots give a larger downwards push, and stop particles from drifting upwards.

When this model was implemented, an additional constraint was needed: if $p < 20$, then the spot displacement was calculated based on $p = 20$. This removes the possibility of a spot only in range of a few particles giving an extremely large push to them. Physically, one can think of this constraint as saying that spots start to partially "evaporate" once they get close to the limit of the free surface.

The results of this model, shown in the right snapshot of figure 4-12, appear very promising. It has all the qualitative features of the DEM simulation, and the particles near the free surface remain packed. The angle of repose of the free surface does not precisely match DEM, but this could be tuned by altering the details of the random walk weighting. We leave this as a promising direction for further study.

### 4.3.2 A two dimensional spot simulation with relaxation

In figure 2-9, a spot model without relaxation was carried out on a two-dimensional regular hexagonal packing, and it is interesting to see whether how a spot model with relaxation will work for this case. Particles in a two-dimensional packing have even stronger geometrical constraints, and have a tendency to crystallize in monodisperse situations, and it is not clear *a priori* whether the spot model microscopic mechanism will be able to handle this case.

Figure 4-13 show two snapshots from a simulation of this type. The spot simulation has no problem running in this geometry, and generates valid, non-overlapping packings, although several of the features are quite surprising. Large areas of crystal-

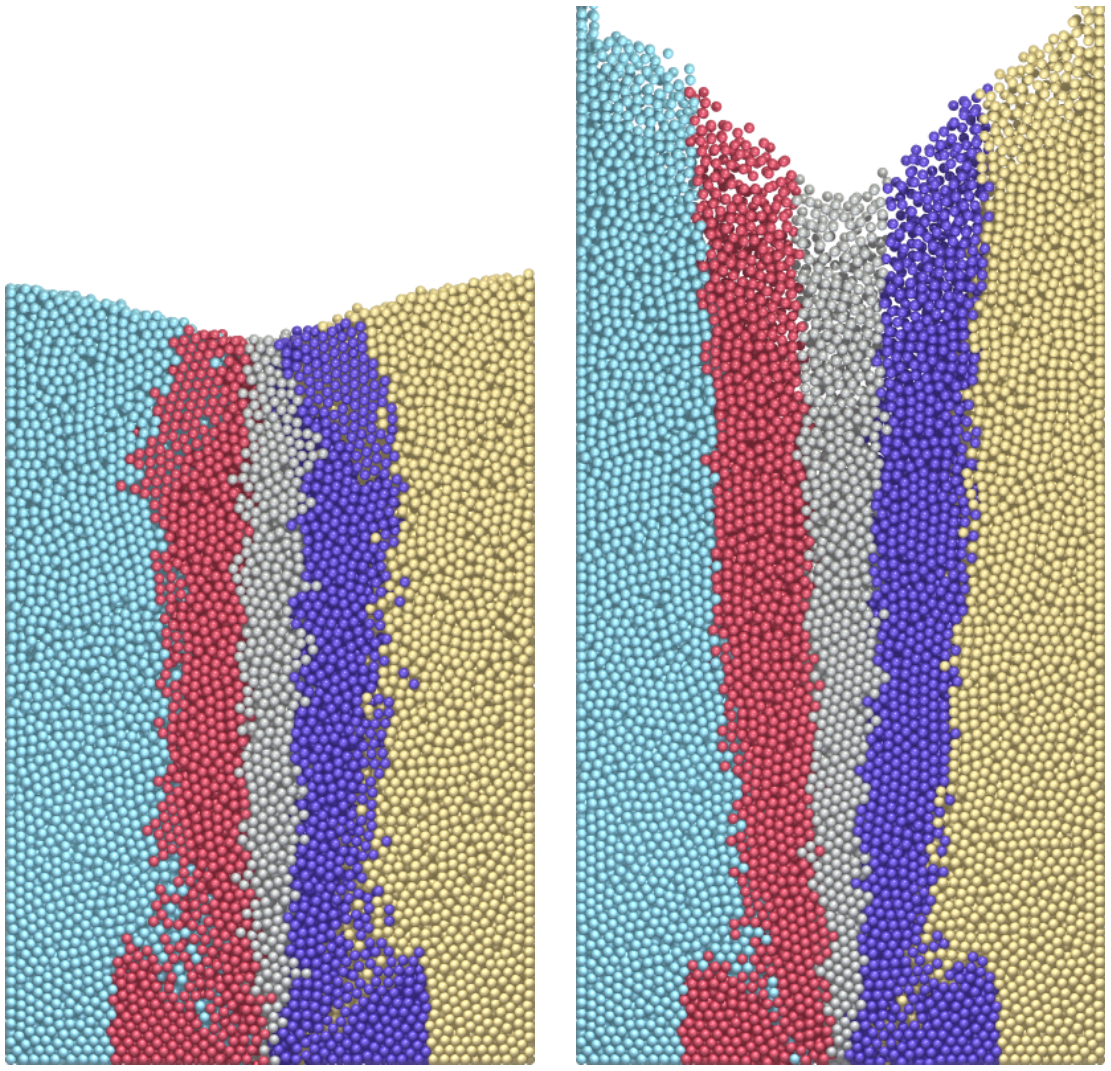


Figure 4-11: Snapshots of the DEM (left) and spot (right) simulation of chapter 3 taken at $t = 300\tau$ showing the evolution of the free surface. The colored particles were initially in equally-spaced columns of width $10d$, and serve as a guide to the eye. The right image shows that if spots follow a completely unbiased walk, then the free surface is not accurately modeled.

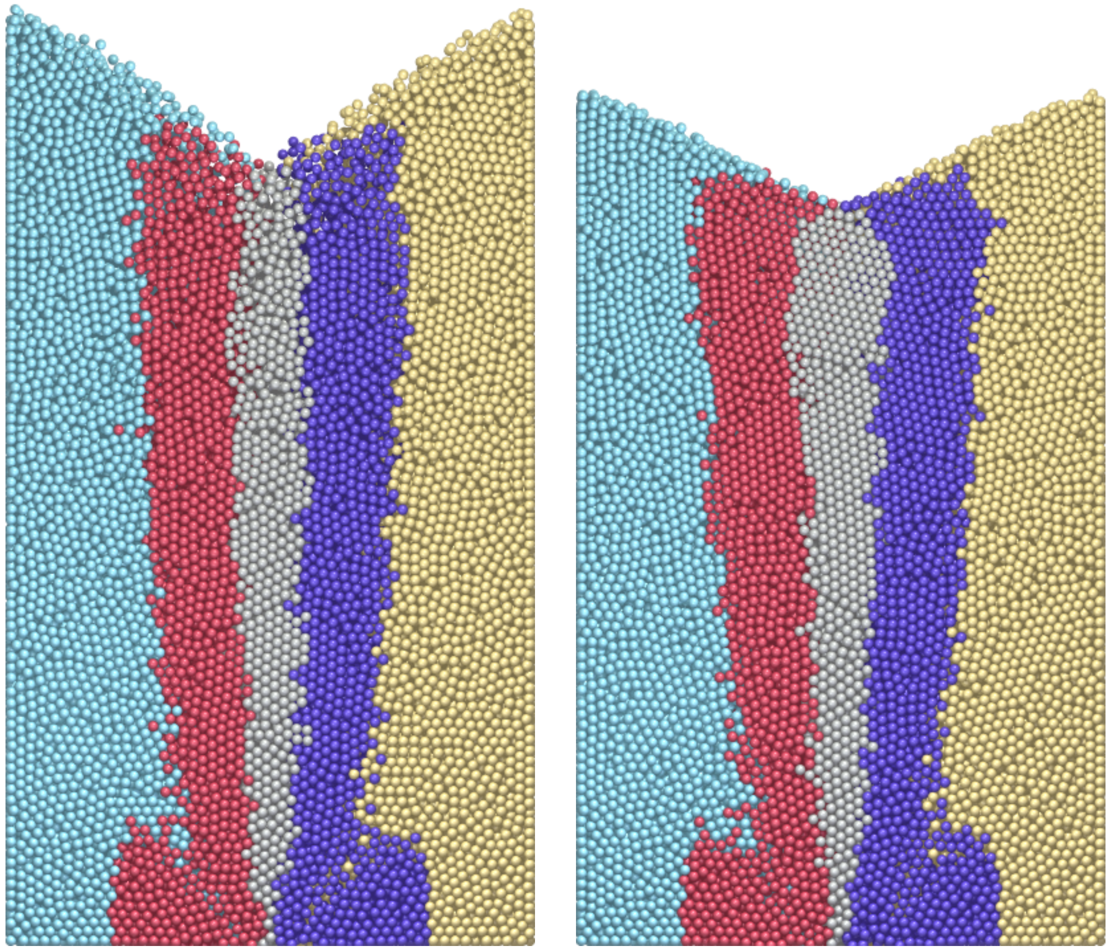


Figure 4-12: Two spot models with a modified random walk process that can qualitatively predict the features of the free surface seen in DEM, while leaving the bulk flow largely unaltered. In the left image, the random walk process was biased so that spots preferentially move towards regions with more particles. In the right image, the influence of the spot was also modified so that spots in range of fewer particles give them a larger displacement.

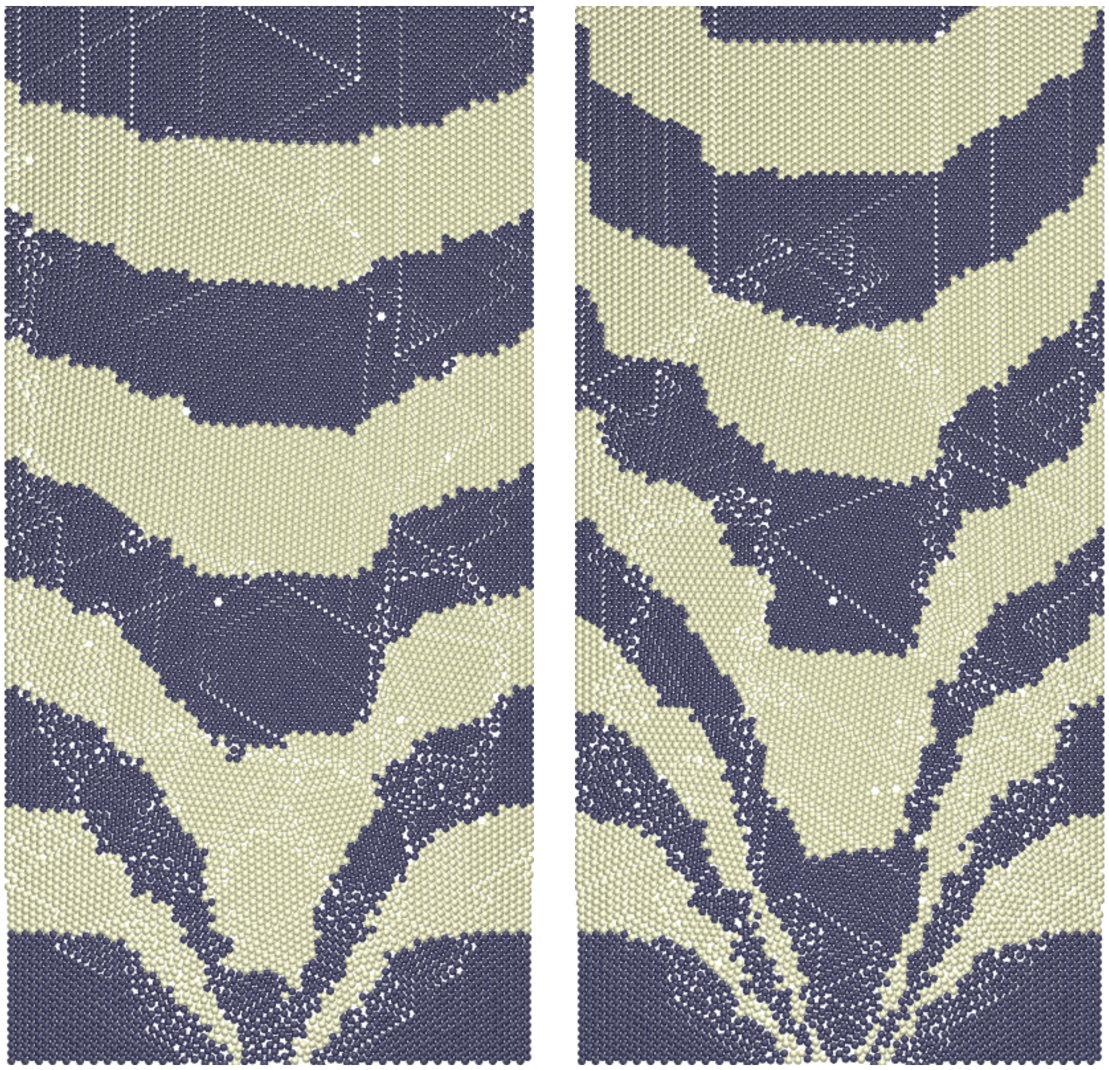


Figure 4-13: Snapshots of a spot simulation with relaxation on a two-dimensional regular hexagonal packing, taken at $t = 200\tau$ (left) and $t = 400\tau$ (right). Extended, linear dislocation defects are visible.

lized particles are visible, and the boundaries of these regions form linear dislocation defects. Many of these defects are extended over tens of particles, and it is interesting that the spot motion (which is only ever applied on an scale of $\sim 5d$) can produce these. Voids in the crystalline structures are also apparent and these play a large role on the rearrangement of the packing, by nucleating dislocations.

## 4.4 Parallelizing the spot model

### 4.4.1 Introduction

One of the major advantages of the spot model as a simulation technique is its speed. In the simulations above, the spot model typically executes many times faster on a single processor than a DEM simulation on twenty processors. However, since the spot model microscopic mechanism is local, and contains no long-range forces, it lends itself well to parallelization, allowing for still faster simulations, or the ability to handle much larger systems.

In this section, we consider two possible methods of parallelizing the spot model simulations considered in the previous section. The codes are written using the Message-Passing Interface (MPI), which is a popular library for development of parallel codes suitable for Beowulf clusters. It provides low-level routines for passing data between nodes, for node synchronization, and also for accurate parallel timing. All the codes discussed in this section were run on the AMCL, allowing for an in-depth comparison of running times.

The following subsection gives an overview of the serial spot model code, and highlights which parts need to be parallelized. In subsection 4.4.3 a parallel implementation using a master/slave architecture is discussed, while in subsection 4.4.4, an alternative code making use of a more distributed architecture is presented.

### 4.4.2 Overview of the serial code

The serial version of this code was written in object oriented C++, and is discussed in detail in appendix C. The bulk of the code is built into the `container` class, which has a constructor of the form

```
container::container(float minx, float maxx, float miny,
                     float maxy, float minz, float maxz,
                     int xn, int yn, int zn);
```

which initializes a simulation box. The first six parameters set the size of the box. The volume is divided up into smaller regions, each of which handles the particles within that region, and the number of subdivisions in each direction is given by the remaining three parameters, `nx`, `ny`, and `nz`. The container holds and manages all particles in the simulation. Simple routines such as `void put(vec &p, int n)` place a single particle with numerical identifier `n` and position `p` into the simulation, assigning it to the correct region. Other routines such as `void dump(char *filename)` dump the particle positions out to a file.

However, the two most important routines for the simulation are the ones responsible for the spot motion and the elastic relaxation. The routine `void spot(vec &p, vec &v, float r)` displaces the particles within a radius `r` of position `p` by an amount `v`. The routine `void relax(vec &p, float r, float s, float force, float damp, int steps)` carries out an elastic relaxation on those particles within radius `s` of position `p`, with those between radius $r$ and $s$ held fixed to prevent long-range disruptions. The quantities `force` and `damp` set the forcing and damping used in the relaxation, and the relaxation process is carried out `steps` times. Further details are listed in appendix C.

The relaxation step is typically the most computationally intensive part of the calculation. For the simulations in the previous chapter, we generally make use of just a single relaxation step, where `force=0.8`, `damp=0`, and `steps=1`, and for these parameters, the entire container can be drained in approximately twelve hours. In those simulations, the single step relaxation scheme was found to be sufficient, but one could easily imagine situations where a more complicated relaxation process was carried out (perhaps taking into account friction). Situations involving a more complex relaxation steps would benefit the most from parallelization, and thus in the timing comparisons given below, it is also helpful to consider a five step relaxation scheme using `force=0.6`, `damp=0.2`, and `steps=5`. The serial version of this code takes approximately two days to drain the entire container.

### 4.4.3 Parallelization using a master/slave model

#### Overview

In the serial spot simulation code, the elastic relaxation step is the computational bottleneck, since it requires analyzing all pairs of neighboring particles within a small volume. In a parallel version of the code, we would ideally like to distribute this computational load across many processors, and since each relaxation event occurs is a local area, we can pass out different relaxation jobs to different processors.

As a first attempt at this, a code was written using a master/slave configuration. During the computation, the entire state of the system (particle positions and spot positions) is held on the master node. The master node then sequentially passes out jobs to the slave nodes for computation, before receiving them back. Since the spot motion events occur at random positions within the container, multiple relaxation events can be computed simultaneously.

In the typical spot algorithm, every spot displacement is followed by an elastic relaxation at that location. For this implementation, it therefore made sense to combine the `spot()` and `relax()` routines into a general routine `void spotrelax(vec &p, vec &v, float sr, float r, float s, float force, float damp, int steps)` which carries out a displacement of `v` with radius `sr` at location `p`, and follows it with an elastic relaxation, using the same parameters as those described for `relax()`. For the parallel version of the code, the `spotrelax()` routine first sends out the parameters of the relaxation job to an available slave node, and then sends across all the particles (typically on the order of several hundred) which can be potentially be influenced. The slave carries out the relaxation, and waits until it receives a new relaxation job, at which point it passes the completed job back to the master node.

A job to the slaves takes the form of four MPI messages. A short header message is first sent, containing the number of particles that are involved in the relaxation and the number of fixed particles in the outer shell. Three buffers are then sent containing the fixed particle positions, the moving particle positions, and the numerical identifiers of the moving particles. The completed jobs are passed to the master node using two

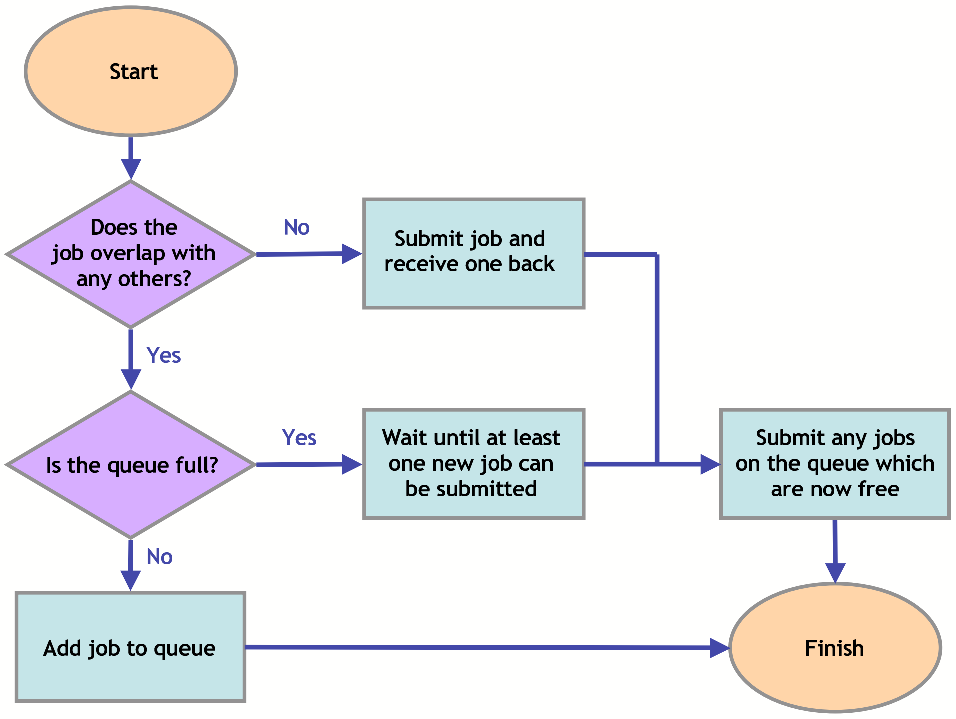

Start
Does the job overlap with any others?
No
Submit job and receive one back
Yes
Is the queue full?
Yes
Wait until at least one new job can be submitted
Submit any jobs on the queue which are now free
No
Add job to queue
Finish

Figure 4-14: Flow chart showing the queueing system used in the master/slave parallelization method.

MPI messages, one containing the numerical identifiers of the displaced particles, and one containing the positions of the displaced particles; the master node already knows how big these messages will be by remembering from when it sent out the job. To save on time the `MPI_Irecv` command is used, so that the new job is transmitted to a slave concurrently with the old job being sent back. However, the message passing in this algorithm could definitely be improved further, and reducing the total number of messages could significantly improve efficiency.

During a typical flow of the system, spot displacements will occur at different positions and as such, their effects can be computed independently. However sometimes cases will arise in which a spot motion takes place in a position which overlaps with a job already being computed on a slave node. A simple resolution to this problem is to wait until the conflicting job is finished before submitting the new one. However, in many cases this may result in a large amount of wasted computer time.

A queueing system was therefore implemented, and this is shown in detail in figure 4-14. If a job overlaps with any others which are already being computed by the slave nodes, then an attempt is made to queue it. If the queue is not full, then the job is added to the queue, and the routine exits. If the queue is full, then jobs are pulled back from the nodes until at least one job from the queue can be submitted, and the original job is then be added to the queue. Each time a job is submitted to the slave nodes, the queue is scanned to see if any new jobs are now free; when the routine exits, it is guaranteed that every job which is stored on the queue cannot at that time be submitted.

Another routine, `void queueflush()`, was written to flush all the jobs from the queue and from the slave nodes; this routine must be run before the particle positions are saved to file.

### Timing results

To test the algorithm, simulations were carried out using different numbers of slave nodes, for both the one-step relaxation process, and the five-step relaxation process. The first sixty snapshots of the drainage simulation were calculated, and the times to generate each snapshot (using `MPI_Wtime`) were logged to a file. Graphs of the progress of the simulation for the two relaxation schemes are shown in figures 4-15 and 4-16.

From the graphs, we see that the rate of computation is initially very rapid, before reaching a steady state after around fifteen frames. Since spots are injected at the orifice during computation, it takes several frames for them to propagate through the container, before the total number in the container reaches a steady state and the number being introduced at the orifice roughly balances the number reaching the top of the packing. In order to assess the rate of computation, linear regression was applied during the steady state region from frame thirty to frame sixty. Figure 4-17 and table 4.2 show the rates of computation compared to those from the serial code. Unfortunately, for the single step relaxation code, the parallel jobs are all slower than the serial code, suggesting that the amount of time spent communicating the particle information to the slaves is larger than the amount of time for the actual computation. However, for the five step relaxation method, a significant speedup is seen over the

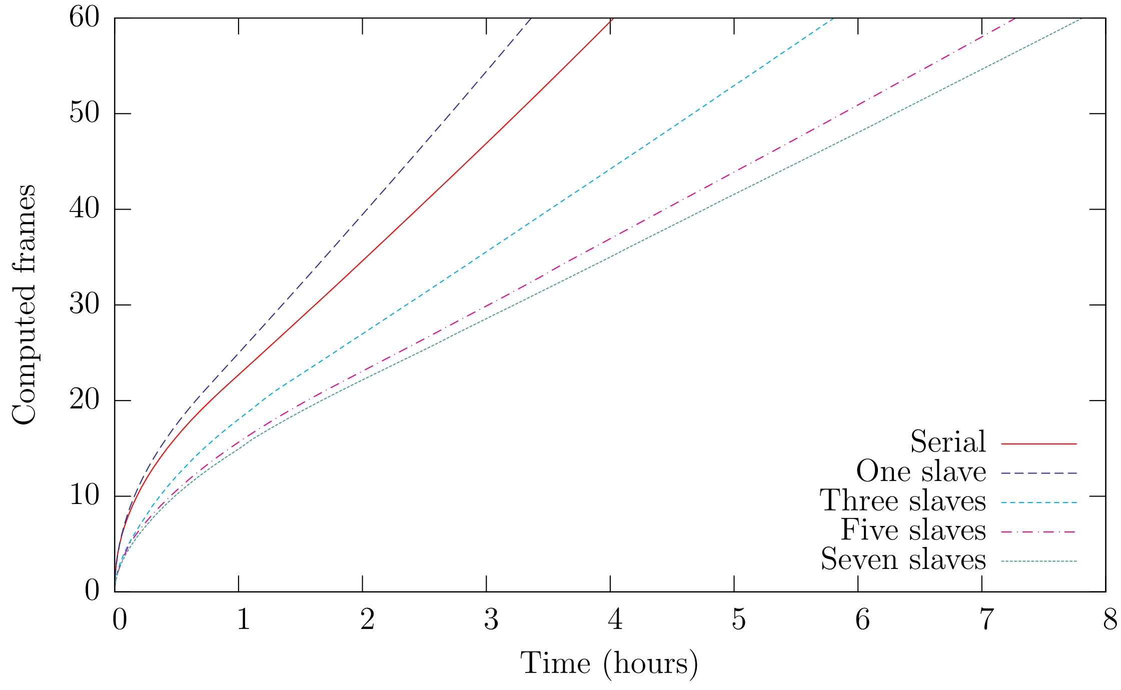


Figure 4-15: Plots showing the progression of the simulation, in terms of the number of frames computed as a function of time, for the single step relaxation method.

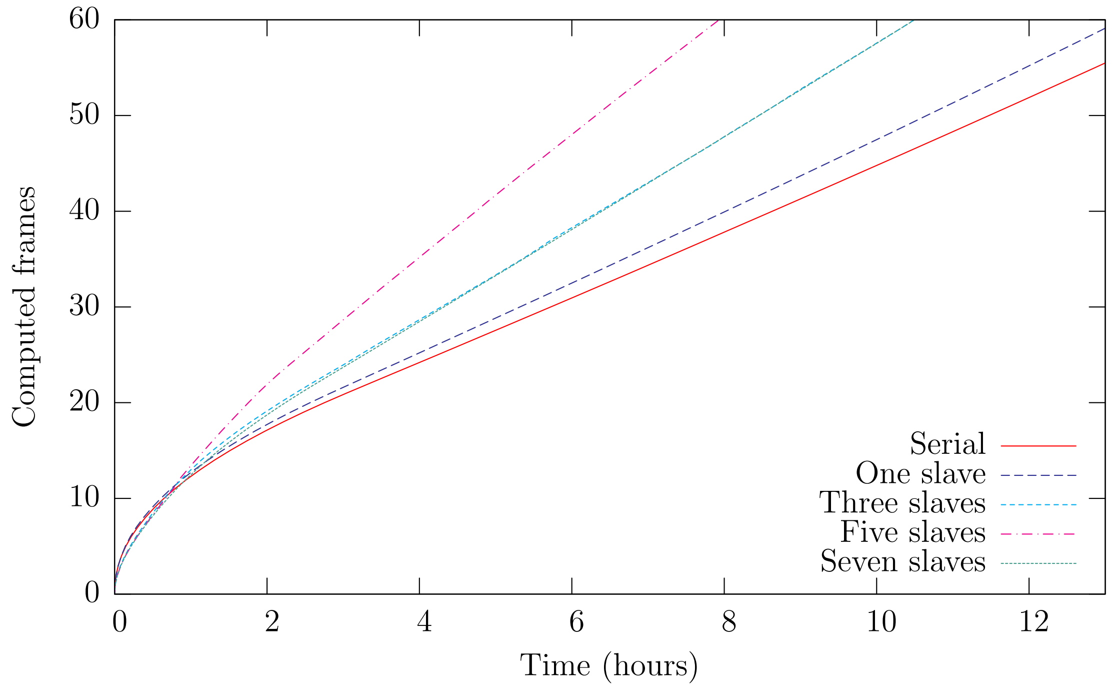


Figure 4-16: Plots showing the progression of the simulation, in terms of the number of frames computed as a function of time, for the five step relaxation method.

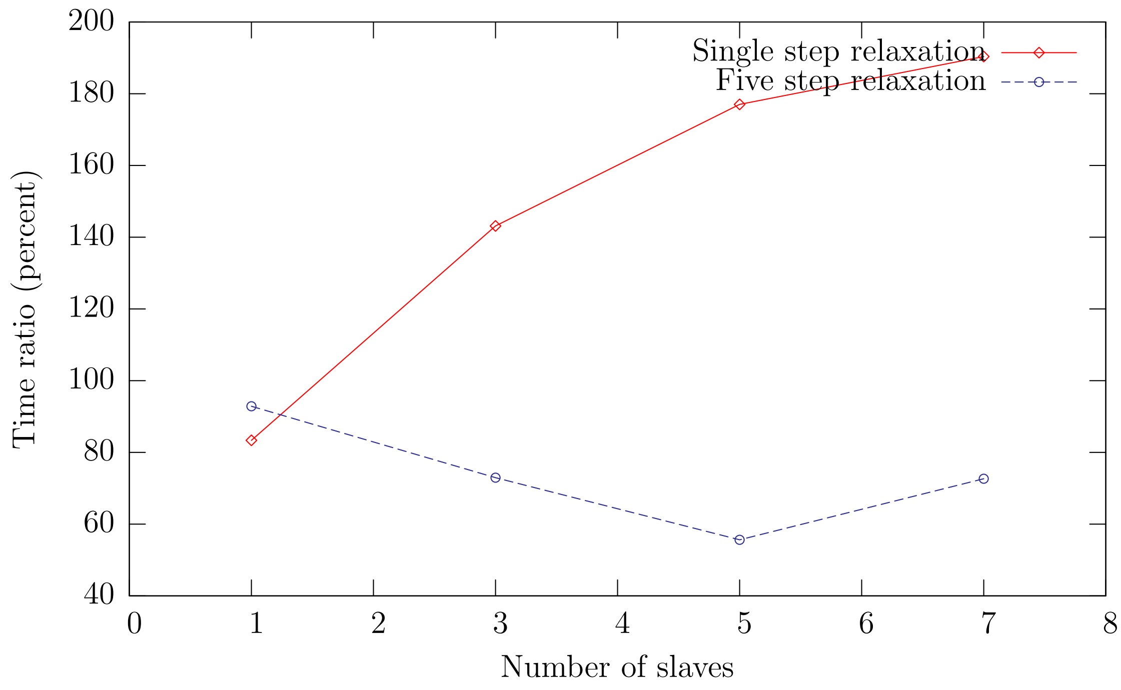


Figure 4-17: Computation times for the master/slave spot algorithm for different numbers of slave processors, shown as a ratio to those for the serial code.

serial code, since the amount of computation required for each relaxation event is much larger.

The results show the problem of creating a parallel algorithm using a master/slave architecture. In this algorithm, too much stress is placed on the master node, and very poor scalability with the number of nodes is achieved, as the slaves often stand idle waiting for the master node to pass them work. The distributed approach, discussed in the following subsection, therefore seems much more advantageous.

However, the above algorithm could potentially work very well on a shared memory machine. In that case, the master node would only need to handle the queue and transfer minimal job information to the slaves, since they would be able to access the particle positions directly. This could be implemented in MPI, but it could also be ideally suited to running in Cilk [20].

| Slaves | One step time (s) | Ratio | Five step time (s) | Ratio |
|---|---|---|---|---|
| (Serial) | 289 | 100% | 1020 | 100% |
| 1 | 241 | 83.3% | 948 | 92.9% |
| 3 | 414 | 143.1% | 744 | 72.9% |
| 5 | 512 | 177.0% | 568 | 55.6% |
| 7 | 551 | 190.4% | 741 | 72.6% |

Table 4.2: Times for the computation of a single frame using the master/slave algorithm, for the single step relaxation and the five step relaxation. The ratios show the amount of time taken for a run compared to the time of execution of the serial version.

### 4.4.4 A distributed parallel algorithm

#### Overview

The parallelization method described in the previous section had the significant drawback that large amounts of data needed to be transmitted to and from the master node. For the case of a single-step elastic relaxation, the master node cannot copy out the jobs to the slave nodes rapidly enough. A second parallelization scheme was therefore considered, in which the container is divided up between the slaves, with each slave holding the particles in that section of the container. A master node holds the position of the spots and computes their motion. When a spot moves, the master node tells the corresponding slave node to carry out a spot displacement of the particles within it. Only the position and displacement carried by the spot need to be transmitted to the slave, significantly reducing the amount of communication. Furthermore, many spot motions can be submitted to each slave simultaneously in a long worklist, minimizing the number of messages that need to be sent.

However, the drawback with this method is that sometimes a spot's region of influence may overlap with areas managed by other slaves. In that case, each slave must transmit particles to the slave carrying out the computation, and then receive back the displaced particles once the computation is carried out. Note however that since this communication happens between slaves, and not between the master and the slave, the workload is much more evenly distributed. The information about when slaves must pass their particles to neighboring slaves for computation can be passed to them by the master node as a type of job in the worklist.

### Details of the worklist

Since sending very small MPI messages is inefficient, the master node sends slaves a worklist of spot motions and other jobs that need to be carried out as an array of 1024 floating point numbers. The slaves sequentially process this information, and then wait for the master node to supply a new set of tasks. Each entry in the worklist begins with a type:

- **1** – Add a particle.
- **2** – Carry out overlapping spot motion.
- **3** – Carry out non-overlapping spot motion.
- **4** – Pass particles to a neighboring slave.
- **5** – Send all particle positions back to the master.
- **6** – Send the total number of particles to the master.
- **7** – Finish the simulation.
- **0** – End of worklist; wait for new tasks.

After the job type, subsequent numbers may follow to describe the specifics of the job. For job type 1 to add a particle, four numbers follow, giving the particle's position and a numerical label. Job types 5, 6, and 7 have no subsequent numbers. For the dump and count operations, the slaves scan all the particles they have, and then send the data to the master node for analysis.

For job type 3, to initiate a spot motion, twelve numbers follow, giving the position and displacement of the spot, plus the details of the relaxation. For job type 2, an additional six numbers are sent, describing a grid of processors which need to be contacted in order to see if they have any particles contributing to the spot motion. Each of these processors is sent a job type 4, telling it to package up all particles which could potentially be involved in a spot motion, and send them off to a neighboring processor. The slave must then wait until the processor carrying out the spot motion sends back the positions of all particles which have been displaced.

**Passing particles between slaves**

In this algorithm, we expect that communication between nodes will be significant bottleneck. To minimize the amount of communication, the code looks at all possible ways to create a grid of slaves using the specified number of processors, and chooses the one which minimizes the shared surface area between nodes. For the test cases considered here, this always results in a grid of the form $1 \times 1 \times n$ for $n$ slave nodes. However the cases of $2 \times 1 \times 2$, $2 \times 1 \times 4$, and $3 \times 1 \times 3$ nodes were also investigated.

Two different methods for passing particles between slaves were considered. For the first method, two MPI messages were used: each slave node first sends the number of particles it is passing to the node carrying out the computation, and it then sends a buffer containing the particle positions and their numerical identifiers. The slave deletes the positions of the particles which can be affected by the relaxation, but keeps copies of the particles which remain fixed in the outer relaxation shell.

Since the receiving node knows exactly how many particles to expect from each processor from the first message, it can efficiently store the incoming particle positions sequentially in a single block of memory, before adding the particles to the ones from its own region for the relaxation calculation. Once the relaxation is completed, the two-message format is used to communicate the particles back to the slave nodes; only the moving particles need to be transferred, since the receiving node kept a copy of all the fixed particles. Figure 4-18 shows a frequency plot of the number of particles passed and returned by this procedure.

Since sending short MPI messages is inefficient, a second communication method was also investigated, whereby particle positions were sent in a single fixed-length buffer. Particle positions are read from the buffer sequentially, and the buffer end is marked with a $-1$ entry. In the case when a processor receives a buffer completely full of particle positions, it waits for a subsequent one to be sent. A buffer length of 1024

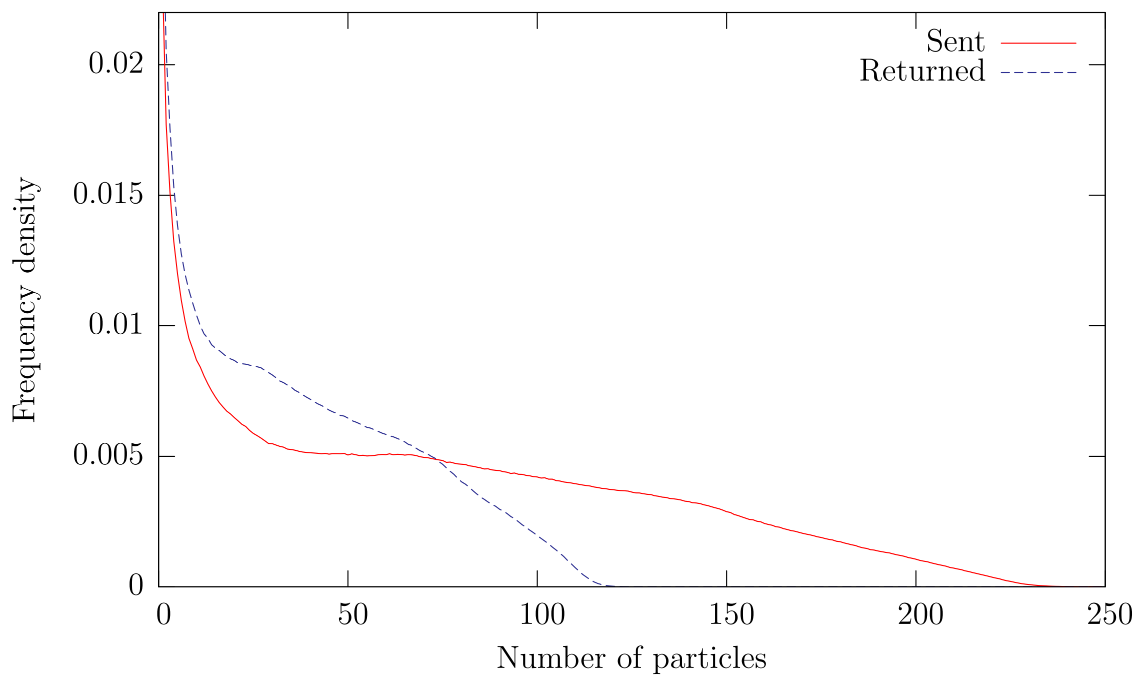


Figure 4-18: Frequency plots of the number of particles transmitted between slaves nodes in the distributed parallel algorithm. The slave nodes send all particles involved in the relaxation, but receive back only the displaced ones, and not the ones which remain fixed during the calculation.

(which can hold 256 particles) was used, which with reference to figure 4-18 should be ample for most inter-processor communications. In addition, a buffer length of 512 was investigated, which should allow most receiving operations to be done in one message, and most send operations to be done in two.

### 4.4.5 Timing results

Figure 4-19 and table 4.3 show how the computation speed scales as a function of the number of slave nodes. We see much better results than for the master/slave algorithm, and even with a single step relaxation we see a marked speedup over the serial code, with the simulation running almost twice as rapidly when five or seven slave nodes are used.

For a larger number of processors with a single step relaxation, we begin to see a small slowdown. This may be an indication that inter-processor communication eventually becomes the most significant factor in computation. While the actual time for the communication may play a role, perhaps a more serious problem may be due to job distribution. Suppose we are running on two slave nodes, and that the master node finds that three spots move on the first slave, then one spot moves on the overlap between slaves, and then three spots move on the second slave. If this occurs, then the second slave will have to wait until the overlapped job is completed before it can execute its three jobs, meaning that the total time for the computation will be similar to executing in serial.

Normally we expect that jobs will be more evenly distributed between nodes, so that computation can be executed in parallel. However, it may be that for larger number of processors, bottlenecking of jobs becomes more frequent. We will be guaranteed that one slave will always be running, but other slaves can potentially have multiple dependencies on others, particularly as the number of slaves and the number of overlapping jobs increases.

In certain situations, a reordering of jobs could significantly improve the flow of the algorithm. In the example above, if the three jobs on the second node were independent of the overlapping job, they could be executed while the first node was processing its own local jobs. This could perhaps be implemented using a queueing system similar to that used in the master/slave algorithm.

Figure 4-20 shows the results of the simulations using a fixed message length, compared to the serial code and those for the variable message length, We considered the case of a $1 \times 1 \times 7$ node configuration using the single step relaxation process. From the figure, we see that the 1024 length buffer results in a slower computation, with a time ratio of 126.7% compared to the variable length case. However, the simulation using a 512 length buffer is faster, with a time ratio of 88.6% compared to the variable length case, and time ratio of 55.7% compared to the serial case.

### 4.4.6 Conclusion

For both of the algorithms considered, significant speedups over the serial version of the algorithm were possible for certain situations. However, the time of the sim-

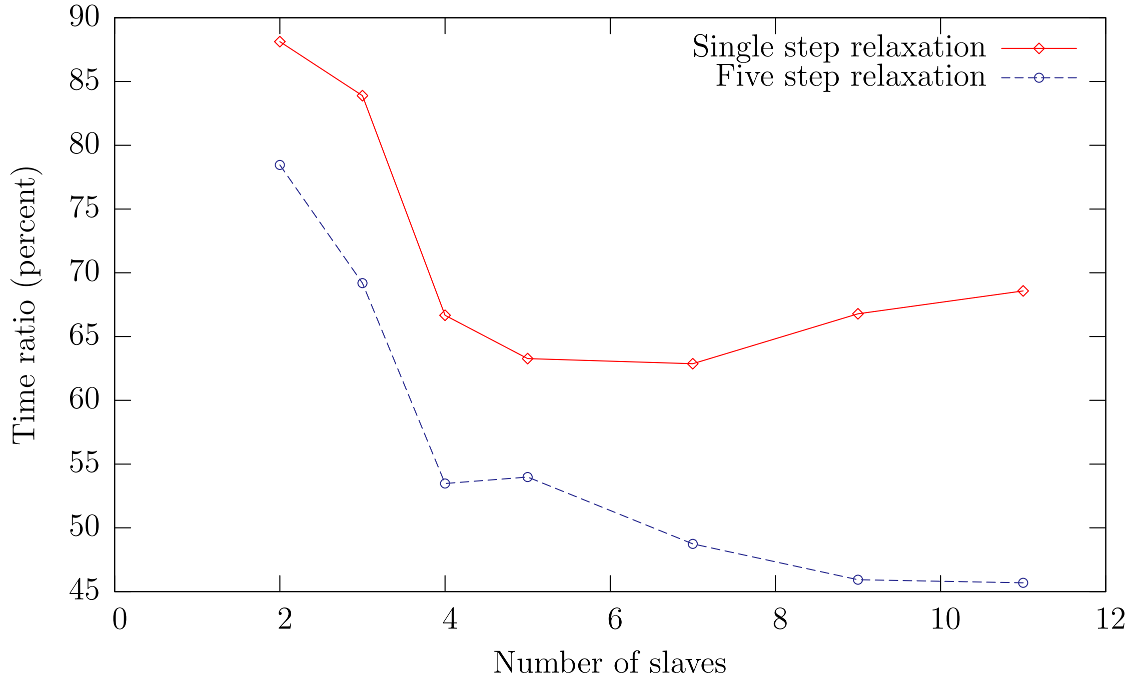


Figure 4-19: Computation times for the distributed parallel spot algorithm for different numbers of slave processors (using the $1 \times 1 \times n$ grid configuration) shown as a ratio to those for the serial code.

| Slaves | Processor grid | One step time (s) | Ratio | Five step time (s) | Ratio |
|---|---|---|---|---|---|
| (Serial) | | 289 | 100% | 1020 | 100% |
| 2 | $1 \times 1 \times 2$ | 254 | 88.1% | 801 | 78.4% |
| 3 | $1 \times 1 \times 3$ | 242 | 83.9% | 706 | 69.2% |
| 4 | $1 \times 1 \times 4$ | 193 | 66.7% | 546 | 53.5% |
| 5 | $1 \times 1 \times 5$ | 156 | 54.0% | 551 | 54.0% |
| 7 | $1 \times 1 \times 7$ | 182 | 62.9% | 497 | 48.7% |
| 9 | $1 \times 1 \times 9$ | 193 | 66.8% | 467 | 45.9% |
| 11 | $1 \times 1 \times 11$ | 198 | 68.6% | 466 | 45.7% |
| 4 | $2 \times 1 \times 2$ | 326 | 112.6% | 828 | 56.6% |
| 8 | $2 \times 1 \times 4$ | 272 | 94.0% | 664 | 65.1% |
| 9 | $3 \times 1 \times 3$ | 320 | 110.8% | 769 | 42.3% |

Table 4.3: Times for the computation of a single frame for different configurations of slave processes, for the single step relaxation, and for the five step relaxation. The ratios show the amount of time taken for a run compared to the time of execution of the serial version.

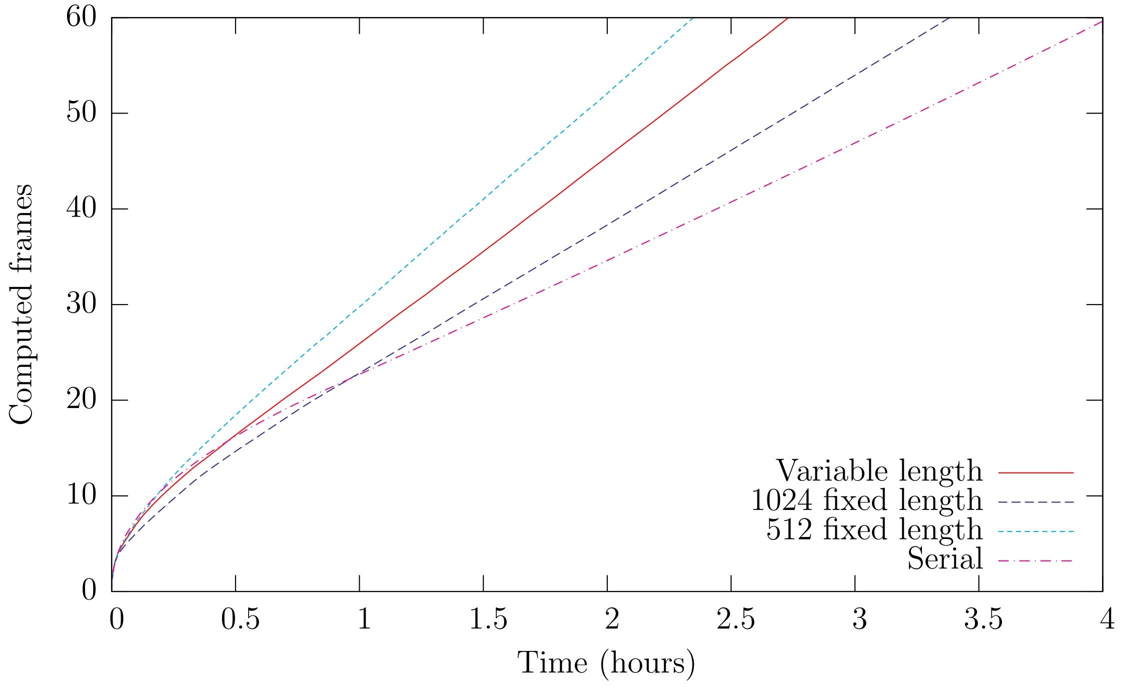


Figure 4-20: Plots showing the progression of computation for the simulations using a fixed message length, for a $1 \times 1 \times 7$ node configuration with a single step relaxation process. Plots for the serial code and the corresponding variable message length simulation are shown for comparison.

ulations is definitely far from the optimal behavior for parallel algorithms, where we would like the time taken to scale inversely with the number of processors. In both of the algorithms, inter-processor communication and job distribution caused significant losses in the effectiveness of the algorithm. Queueing and reordering jobs, and improving the efficiency of message passing were both considered, and have both resulted in speedup improvements.

However, it is hoped that with more work, significant advancements could still be made in both of these algorithms. For the master/slave method, where the efficiency of the master node is paramount, improvements in the handling of the queue could speed up the computation. For the distributed method, implementing a job reordering scheme would be beneficial. In both schemes, improving the format of the messages between nodes could be investigated further. The tests with fixed length messages of 512 numbers showed improvements over the variable length messages, and a more thorough tuning of the length of this buffer could be beneficial.

Making more advanced use of MPI could also help. In places, efforts were made to make use of synchronous sending of messages, using the commands `MPI_Irecv` and `MPI_Wait` to exchange data between two processors concurrently. However, more advanced MPI functionality was not considered, and in places send operations were used which waited for confirmation of their send when they did not need to.

Other types of algorithms could also result in better performance. A fully distributed algorithm, in which particles and spots are all shared between slaves, may exhibit better load-balancing. This possibility was not considered here, since it was unclear how to run an event-driven simulation of this type across many nodes. However, if the random waiting-time distribution for spot motion was replaced with moving at every fixed time interval, the slaves could more easily manage the spots in their region without reference to other nodes.

Another possible algorithm would involve calculating overlapping spot motions using all of the slaves involved, rather than first transferring all the particles to one of the slaves for computation. Each slave could handle the relaxation of the particles in its region, and then send messages to the neighboring slaves to handle interactions between particles on different slaves. While this could result in more messages being sent, far fewer particles overall would actually need to be transferred, since only the skin of particles exactly at interface between regions would need to be communicated.