Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Mathematics

Large scale collision detection with variable time step using domain decomposition and MPI

Michael Christopher Rooney
11510123

MSc in High Performance Computing
2017-2018

Declaration

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism ‘Ready Steady Write’, located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Acknowledgments and notes

Thanks to my father John and my girlfriend Tiina for their constant support.

I would also like to thank Dr. Mike Peardon and Mr. Dermot Frost for providing feedback and advice throughout this project.

All performance figures were obtained from running code on TCD’s Lonsdale cluster.

All code written for this project can be found at:
https://github.com/MikeyRooney/CollisionProject

Previously the code was split into two GitHub repositories. These archived repositories, along with their commit history, can be found at:
https://github.com/MikeyRooney/Visualiser
https://github.com/MikeyRooney/Collision-project

Table of contents

1: Introduction and background

1.1: Introduction and goals

The goal of this project is to investigate the performance benefits that domain decomposition brings to collision detection, and how a simulation using domain decomposition can be efficiently distributed across many cores using MPI. As discussed in section 1.3, the naive approach to collision detection is too slow for large scale use, which is where the need for methods such as domain decomposition arises.

Domain decomposition is not a new idea. It has been used before to speedup collision detection and N-body problems. However while investigating the state of the art, the author was unable to find much information about how far it can be exploited nor how it can be parallelised using MPI.

Therefore this project seeks to answer the following questions:

1. How much can domain decomposition be exploited to achieve a speedup in a serial program?
2. Is it worthwhile to use MPI to parallelise a domain decomposition system? Or can the same effect be achieved more easily by simply diving the simulation area in the serial version into a very large number of sectors?

While this project focuses mostly on CPU bound workloads, there are some scenarios where the amount of memory required becomes the bottleneck. This is especially noticeable with certain possible optimisations discussed later. Therefore the use of MPI to distribute the memory load will also be briefly discussed.

For this project a simple simulation model is used. It will consist of spheres impacting other spheres with no loss of energy to deformation, heat, sound etc,. The reason such a simple simulation model was used is so that effort is spent on developing and investigating the domain decomposition system, rather than developing a complex simulation model. However no part of this project will depend on this simple mode. If so desired a more complicated model could be built that makes use of the techniques discussed here.

Finally for validation and seeing the results of the simulation a simple OpenGL visualiser will be created.

1.2: Fixed time step vs. variable time step

In collision detection two approaches may be used.

In the fixed time step approach the simulation is moved forward by a fixed time step each iteration. One disadvantage of this approach is that objects may end up partially inside each other after being updated. Another disadvantage is that if the time step is too large objects may pass through each other if their velocities are large. These issues necessitate the need to check for their occurrence and possibly “rewind” the simulation to ensure accurate results. Finally another disadvantage is that if the time step is too small many iterations may occur where no collisions take place.

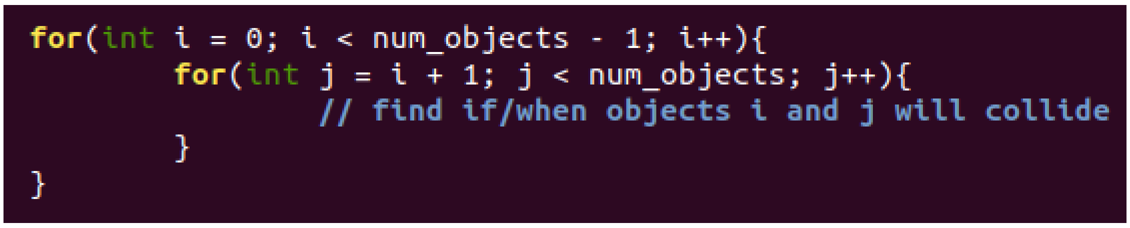
In the variable time step approach, used in this project, the simulation is not moved forward by a fixed amount of time each iteration. Instead each object is checked against every other object, and if they will collide the time of the collision is noted. Then for each object the time when it collides with the simulation boundaries is found. The soonest of these times is noted, and the simulation is advanced by this time. By doing this it can be ensured that objects will never end up partially inside one another or pass through one another.

A fixed time step simulation is useful when the results must be shown in real time and when completely accurate results are not needed, such as in a video game. A variable time step simulation is more suited when completely accurate results are needed and there is no demand for the results to be obtained in real time, such as in scientific simulation. Therefore this project will use a variable time step

In this project the concept of an *event* is used. An event is anything that may be the reason the variable time step simulation is updated. At the very least two event types are needed: object on object collisions and object on simulation boundary collisions. Domain decomposition introduces two additional event types, which are discussed later.

1.3: Naive collision detection and naive parallelisation

The simplest way of implementing variable time step collision detection is with a nested for loop as shown in figure 1. This approach checks every object in the simulation against every other object. However this scales at a *$O(n^2)$* rate and thus quickly becomes unusable as *n* grows[1].


```
for(int i = 0; i < num_objects - 1; i++){
        for(int j = i + 1; j < num_objects; j++){
                // find if/when objects i and j will collide
        }
}
```
Figure 1: naive collision detection approach.

Such an approach is trivial to parallelise using OpenMP and a dynamic or guided schedule. However with *p* threads this will give at best *$O(n^2/p)$* scaling, which is still unusable for large values of *n*.

Therefore some other approach is needed which greatly reduces the number of checks that must be performed, while also being suited for large scale parallelisation. The author believes domain decompositions satisfies both of these conditions.

2: Domain decomposition

In this section I will discuss how domain decomposition can be used to decrease the time taken to determine the next event. I will also discuss the drawbacks of using domain decomposition, as well as point out some situations where it is not suitable. Finally I will discuss some existing ways it is used.

2.1: Overview of domain decomposition and its benefits

Decomposing the simulation area into multiple distinct *sectors* can result in a large speedup. This is due to the fact that in a given sector each sphere will typically need to only be checked against other spheres in the same sector. In effect spheres in different sectors are being abstracted into one large object, the sector itself, and collisions can be checked against the sector's boundaries rather the against spheres within.

Consider a simulation with *n* objects. As mentioned previously the naive approach has $O(n^2)$ scaling as pairwise checks between all *n* objects must performed. If the simulation is divided into four sectors (as shown in figure 2), each containing the same amount of spheres, then the scaling for each *individual sector* will be at best $O((n/4)^2) = O(n^2/16)$.

In a serial system each sector will be processed one after another. In this example as there is four sectors, each with at best $O(n^2/16)$ scaling, the *total scaling* should be at best $O((n^2/16) * 4) = O(n^2/4)$. However in a parallel version where each sector is handled independently by a different core then the scaling will stay at best $O(n^2/16)$.

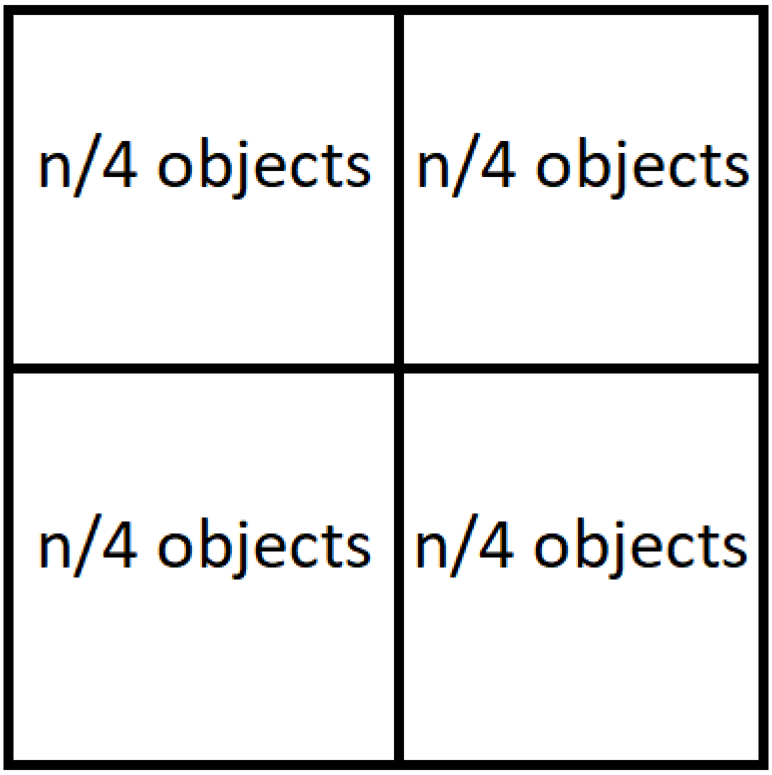


Figure 2: Simulation divided into four sectors.

Generalising the prior two paragraphs the following is obtained: if the simulation area is divided into *s* sectors, then a serial implementation will have at best $O(n^2/s)$ scaling while a parallel implementation will have at best $O(n^2/s^2)$ scaling.

This scaling can be improved by selectively performing collision checking in sectors which have had events occur within them in the prior iteration, as will be discussed later in section 2.4.

This introduces two new event types however. The first is where an object passes from one sector to another. The second is when objects partially cross into adjacent sectors and collide with objects in the adjacent sector. Both of these new events are of major importance to the performance of a simulation using domain decomposition, as discussed below.

2.2: Sector transfers

When a sphere in one sector crosses into another sector a sector transfer occurs. The simulation then moves forward to the time of the transfer and resumes from there. This limits how much of a speedup the versions using domain decomposition can gain compared to the naive version. Given the same simulation area and sphere data, the number of sector transfers will grow as the number of sectors the simulation is divided into increases.

Eventually a point will be reached where the simulation spends more time moving spheres between sectors rather than doing work related to the results of the simulation. As such this project will investigate the speedup obtained as a given simulation area is divided into more and more sectors.

2.3: Partial crossings

Consider figure 3 below. If a sphere within one sector is partially crossing into another sector then it must be ensured that it collides correctly with spheres in the sector crossed into. In this project such events are called "*partial crossings*".

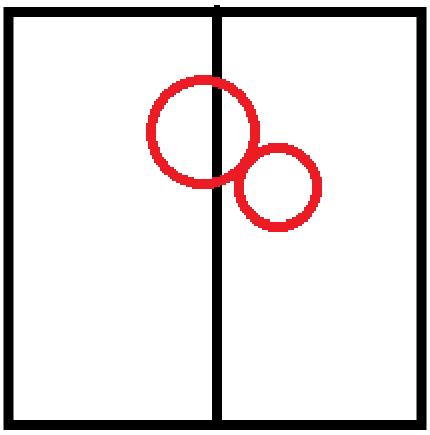


Figure 3: A simple partial crossing example

The naive approach to solve the partial crossing problem is to check each sphere against not just the spheres in its own sector, but also those in adjacent sectors. This approach can still offer some speedup when there a large number of sectors in a 2D grid (therefore many are not adjacent), but this is not an optimal approach. In addition it completely removes any performance increase that domain decomposition brings when a 2x2 division is used. As all sectors are adjacent they will all will check for partial crossings against every other sector. Also if a large 3D grid is used then sectors which are not at the edge of the grid will have up to 27 neighbours, which would be unacceptably slow as all of these neighbours must be checked for partial crossings. Thus a better approach is needed.

In order to ensure that the partial crossing problem does not reduce performance, it must be ensured that a sphere is checked against spheres in adjacent sectors only when absolutely necessary. To ensure this the following steps were implemented:

1. Each sector tracks the largest radius from the spheres it contains.
2. Only sectors adjacent in the direction a sphere is travelling can possibly be checked. If a sphere has a negative velocity on the x-axis then it will not be checked against a sector that is adjacent in a positive direction on the x-axis
3. Partial crossing checking occurs last so the time of the current soonest known event is available. The position of each sphere in the sector at the time of this event is calculated and stored in a temporary location.
4. A sphere is checked against spheres within an adjacent sector if and only if the following holds: at the time of the soonest known next event the sphere's distance from any adjacent sector it is travelling towards is within its own radius + the largest radius from the adjacent sector.

Stated more simply: partial crossings are checked if and only if a sphere will be within a certain minimum distance of an adjacent sector that it is travelling towards before the soonest known event occurs. This minimum distance is based on the sphere's radius, and the radius of the largest sphere in the adjacent sector.

As will be discussed later in section 5 these steps generally work very well in ensuring that partial crossings are not checked for unless needed.

2.4: Using sector invalidation for further speedups

When the simulation is divided into sectors there is an additional optimisation that can be added easily. After the first iteration the soonest event time for each sector can be tracked and invalidated only if that sector is involved in the next event. Then each sector that has a valid soonest event time does not need to be checked in the next iteration. Instead its event time from the prior iteration can be lowered by the overall soonest event time from the prior iteration.

In this approach if the event is a sphere colliding on the grid boundary or a sphere on sphere collision within a sector then only the sector this event occurs in needs to be invalidated for the next iteration. If the next event is a partial crossing or a sector transfer then both sectors involved must be invalidated for the next iteration.

Consider an example where the simulation area is divided into two sectors. On the first iteration it is found that sector one has two spheres that collide after 10 seconds, and sector two has two spheres collide after 6 seconds. The simulation can be updated by 6 seconds and the collision applied. In the next iteration no work needs to be done for sector 1 as we know its next event occurs in 4 seconds after its previously found event time is lowered by the overall soonest event time.

With sector invalidation the theoretical max speedup in the serial version becomes much larger. As mentioned in section 2.1 the maximum scaling for a serial version with s sectors is at best $O(n^2/s)$ as all s sectors must be checked, and each individual sector has at best $O(n^2/s^2)$ scaling. With sector invalidation the scaling improves significantly. When only a single sector is invalid in a given iteration the scaling for that iteration will be at most $O(n^2/s^2)$. If two sectors are invalid for the given iteration then the scaling for that iteration will be at most $O(2n^2/s^2)$. It is important to note that the first iteration will still have at best $O(n^2/s)$ as each sector must be checked then.

If most of the time only a single sector is invalidated after each iteration, and assuming there are many iterations past the first one then the maximum scaling for the serial version with domain decomposition will approach $O(n^2/s^2)$. The scaling for the MPI version with this optimisation is discussed in section 3.

2.5: Drawbacks of domain decomposition

The aforementioned maximum scaling figures describe the best case scenario where spheres are evenly distributed throughout the simulation area and the simulation area is densely populated. However there are some scenarios where domain decomposition will offer a smaller speedup (or possibly a slowdown).

One such scenario is if the simulation area is divided into too many sectors. More time will be spent on the administrative work of transferring spheres between sectors than processing collisions. The frequency of needing to check for partial crossings will also increase dramatically. The effects of this are discussed in section 5.1.

Another scenario is if the workload is extremely unbalanced. If the majority of the objects end up in one sector then the scaling becomes $O(n^2)$ again. Solutions for this issue are discussed in section 9.3.

Finally domain decomposition can be slower if the simulation area is sparsely populated. This occurs because spheres are unlikely to hit each other, and will instead transfer between sectors most of the time. This is discussed further in section 5.2.

2.6: State of the art and existing similar work

The domain decomposition method used in this project is not new, and there are many variants of it. Some example of domain decomposition methods that are in use are octrees, k-d trees and binary space partitioning (BSP) trees[2]. These approaches are use the same fundamental idea that tests are only performed between objects that exist in the same spatial division.

One application of domain decomposition is the Barnes-Hut algorithm for N-body simulations[3][4]. In short, this approach splits the simulation area into a number of cells. For objects that are sufficiently distant the average mass and center of gravity of the containing cell is used, reducing the number of particles that need to be processed. This can improve the scaling from *$O(n^2)$* to *$O(n\log(n))$*, a significant improvement.

The Barnes-Hut algorithm is similar to the approach used in this project in that distant particles are abstracted away and handled in a different manner. For this project all spheres in a different sector are for the most part treated as one large object, and collisions against the spheres within a different sector are only checked when when a partial crossing may occur.

Another similar approach is the use of *bounding boxes*[7]. These are simple shapes (such as cubes or spheres) that entirely surround a more complicated model. When checking for collisions between two complicated models their bounding boxes can then be used for a fast reject by checking if they intersect, which is fast to check as they are simple shapes. If the bounding boxes belonging to the two models do not intersect then the models are not colliding. This is similar to how the domain decomposition approach in this project treats spheres in another sector as one large square or cube.

While researching the state of the art I did not find much discussion about how domain decomposition scales as the number of sectors increases. Nor did I find much discussion of parallelising a domain decomposition system with MPI.

The method of detecting collisions used in this project was adapted from one published online previously[8]. The adaptations consist of using trigonometry and relative velocity in finding the time of collision rather than using vectors alone.

3: Parallelising the workload of a simulation using domain decomposition

3.1: Overview of the MPI approach used

A cartesian grid topography was used. Depending on the parameters provided to the program, the grid can be 1D, 2D or 3D. Once the grid communicator is created, the process with rank 0 is assigned to sector 0, and so on. Each process is responsible for one sector. But other processes can help another process find events for a sector, as described below.

Periodic boundary conditions were not used as the simulation was assumed to have hard boundaries. However no part of the code depends on periodic boundary conditions being disabled, and it would not be difficult to support them.

At a minimum each MPI process must know about the spheres in any sectors adjacent to the sector it controls. This is needed so that partial crossings can be checked.

There are two modes of operation for the MPI version, *all help* and *neighbours help*. These are discussed below.

3.2: Reducing memory usage with shared memory

It was noted that memory usage could be reduced by using shared memory for neighbours that are on the same physical node. At startup each MPI process checks the hostname of the node it lives on. It then allocates its own spheres buffer using file-backed shared memory and broadcasts the file name used. Other processes will detect which sectors are neighbours on the same physical node, and will use this shared memory rather than allocate a buffer for their own copy of the neighbouring sector's sphere data. Neighbour sectors that are handled by processes on the same node are called *local neighbours*.

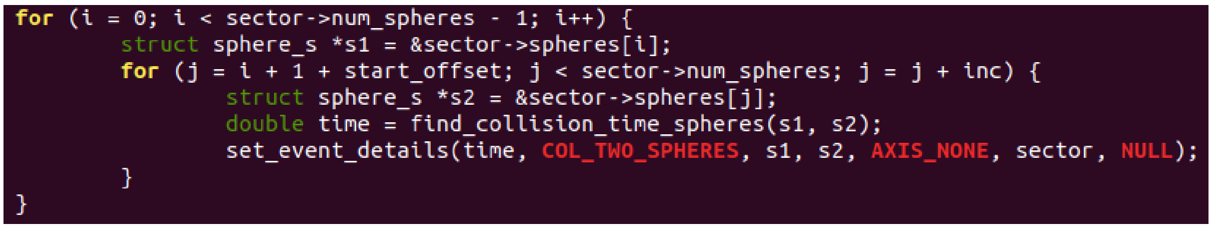
3.3: All help MPI version

In the *all help* mode every MPI process keeps a copy of all spheres in all sectors. In the first iteration each process finds the soonest event time for its own sector and broadcasts it. After the first iteration each process will handle part of the workload for the invalid sector. If two sectors are invalidated then they will be processed one after another.

There was a small challenge in distributing the workload for a single sector evenly between all processes. As can be seen in figure 1, the inner loop uses *i + 1* as a start index. Therefore if the outer loop is divided evenly between processes there will be an unbalanced workload. Consider two processes in all help mode: the first takes the first half of values for *i* and the second takes the second half of values for *i*. The second process will have much less work to do because of the inner loop using *i + 1* as its start.

To solve this issue the following approach is used. Each process uses the same outer for loop which takes the entire range of spheres. In the inner for loop each process starts at *i + 1 + rank*, and increments *j* by the number of processes. This does not perfectly distribute the workload, but comes extremely close with large values of *n*. With 64 cores processing 78’125 spheres in a *single sector* in *all help* mode, the difference in workload between the processes with the smallest and largest workload was only 0.16%.

Consider figure 4 below, which shows this implemented in code. For *all help* mode *start_offset* will be set to the process’s rank, and *inc* will be set to the number of processes.



```
for (i = 0; i < sector->num_spheres - 1; i++) {
        struct sphere_s *s1 = &sector->spheres[i];
        for (j = i + 1 + start_offset; j < sector->num_spheres; j = j + inc) {
                struct sphere_s *s2 = &sector->spheres[j];
                double time = find_collision_time_spheres(s1, s2);
                set_event_details(time, COL_TWO_SPHERES, s1, s2, AXIS_NONE, sector, NULL);
        }
}
```

Figure 4: How multiple processes can process one sector.

As mentioned in sections 2.1 and 2.4 the maximum possible scaling in the serial version with domain decomposition is $O(n^2/s^2)$, where *s* is the number of sectors. Therefore in an MPI version with *s* cores and *s* sectors the maximum possible scaling is $O(n^2/s^3)$, assuming the first iteration takes up a very small amount of the total running time.

This mode of operation is by far the fastest, but it is quite memory intensive as every single physical node needs a fully copy of all spheres.

3.4: Neighbours help

All help mode is not ideal sometimes as every physical node needs a copy of all data in the system. In simulations with objects more complicated than simple spheres, and/or when there is an extremely large number of objects there may not be enough memory on each node to store the data for each object. However, due to the need to check for partial crossings each MPI process must have access to the data that belongs to its sector’s neighbours. This allows a *neighbours help* mode to be created, where processes with neighbouring sectors can share the workload without any additional memory requirement.

The overall operation of *neighbours help* mode is similar to *all help* mode. It differs in that sectors will not track all data in the system, and a smaller number of sectors will be helping the invalid sectors. After the first iteration the neighbours of the invalid sector, and the invalid sector itself, divide the work among themselves. If there are two invalid sectors then they are processed one after another.

Rather than have complicated code which decides which processes need to broadcast the time they found for the helped sector, every process will broadcast a time.If a sector is not a neighbour of the invalid sector it will set simply broadcast DBL_MAX as its time found for the helped sector.

As described above in section 3.3, the naive way of distributing work among multiple processes results in an unbalanced workload. In *neighbours help* mode this is slightly more complicated as the number of neighbours helping varies from sector to sector. To fix this for the *neighbours help* mode the number of neighbours and a sorted list of the neighbouring sector's ids is kept for each sector. When a neighbour wants to help a sector it finds the position of its id in this list and adds one to it (the sector being helped will use a value of zero here). This will be the variable *start_offset* in figure 4. Then the *inc* value in figure 4 is set to the number of neighbours the helped sector has, plus one as the sector is also helping itself.

The performance of this approach can vary depending on which sectors were invalided in the prior iteration and where they are in the grid. Consider a large 2D grid. If a sector in the corner is invalidated it will have three neighbours help it, but if a sector in the center of the sector grid is invalidated it will have eight neighbours help it. With a large 3D grid the maximum number of helping neighbours is 27, which occurs only if the invalidated sector is surrounded by other sectors. In each case the sector is also helping itself, so the total number of cores processing events for that sector will be the number of neighbours plus one.

Therefore with a 2D grid the maximum scaling will be $O(n^2/(9*s^2))$, while in a 3D grid it will be $O(n^2/(28*s^2))$.

While this mode of operation is slower than the *all help* mode it can still provide a significant performance increase without the increased memory requirements of *all help* mode.

3.5: Simplified flow chart for MPI version

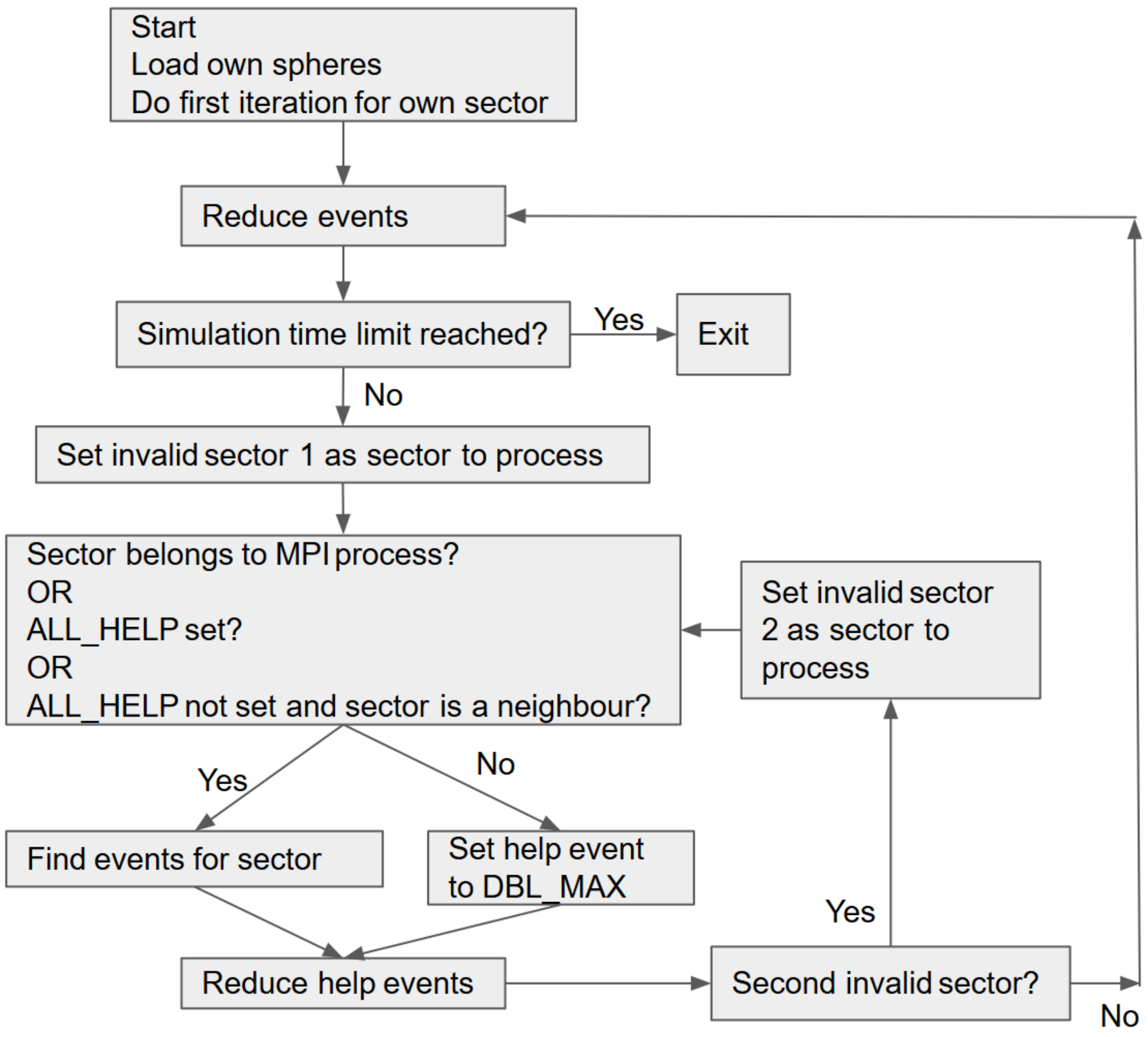

Start
Load own spheres
Do first iteration for own sector
Reduce events
Simulation time limit reached?
Yes
Exit
No
Set invalid sector 1 as sector to process
Sector belongs to MPI process?
OR
ALL_HELP set?
OR
ALL_HELP not set and sector is a neighbour?
Set invalid sector 2 as sector to process
Yes
No
Find events for sector
Set help event to DBL_MAX
Yes
Reduce help events
Second invalid sector?
No

Figure 5: A simplified flow charting showing the MPI version works.

3.6: Minimising communication costs

Regardless of which mode of operation is used, each sector needs to know the contents of at least the neighbouring sectors. Initially it was planned that a full copy of the sphere data in a sector would be broadcast after each iteration. However later this was changed so that a difference from the prior iteration is sent rather than a full state.

At the end of each iteration every process broadcasts its next event details. Each process then reduces this data to find the overall soonest next event. As a difference is broadcast rather than a full state being sent, spheres must now be moved forward by the elapsed amount of time.

Each process moves its spheres forward by this amount of time and the applies the event if it occurred within its own sector. In *all help* mode all spheres that do not belong to local neighbours must also be updated in this way. In *neighbours help* mode only spheres belonging non-local neighbours must be updated in this way. Neither mode requires spheres in local neighbours to be updated, as the local neighbours will have already updated their spheres in shared memory.

This sphere updating scales linearly with the number of spheres that must be updated. This does not impact the performance noticeably however, as will be seen when the performance is analysed in section 6.

4: Software design considerations

The following are some software design considerations which are common to the serial and MPI versions of the code.

4.1: Minimising file I/O costs

In order to minimise the I/O costs, the initial state of each sphere is written before the first iteration. Then after each iteration only the changed data of affected spheres is written to disk. This data consists of the ID, position at the time of the event and new velocity for any spheres involved in the event. For example: if there is ten spheres and in a given iteration two bounce off each other, then only the data for those two spheres is written for that iteration.

This approach requires far less disk space and storage bandwidth than writing the state of all spheres each iteration. However it does require the program that interprets the results (such as the visualiser described in section 8.3) to be able to rebuild the state of each sphere on each iteration.

In the MPI version the *MPI File* functions are used. Each iteration one process is responsible for saving the events of that iteration to the data file. For sphere on sphere collisions within a sector or a sphere collision with the grid boundary this will be the process responsible for the sector that the event occurs in. For sector transfers it is the process that owns the sector the sphere is leaving. For partial crossings the process with the lowest rank involved in the event is used. Every other process will update the file pointer for the next iteration, but otherwise won't perform any file actions.

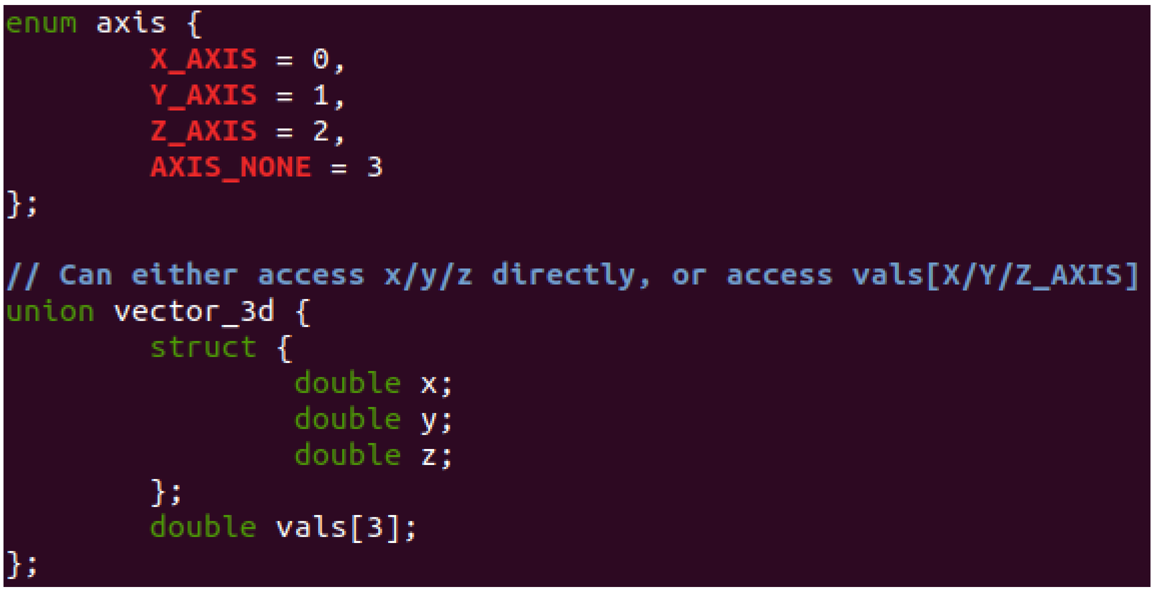
Using a difference based data file rather than a complete state based data file is quite beneficial in the MPI version. If a state based file was used then after each iteration all processes would try to write to the same file at once, which would likely have poor performance as each would be writing to different parts of the file.

See appendix B for a description of the file format used.

4.2: Reducing duplicate code

One issue that became apparent early in the project is that a large amount of duplicate or almost duplicate code was being written. Many times duplicate functions were being created with the only difference between them being the axis they handled. Reducing the amount of almost duplicated functions would make the program much easier to debug as there is no need to tediously test very slightly different versions of each function. It could also reduce execution time by a small amount as duplicate code is no longer using up space in the instruction cache.

To reduce the duplicate code issue a vector union and an axis enum were created. By using the axis enum as an index into the vector union the axis to be processed could be passed as a parameter to a function, allowing one function to handle all three axes. The definition of the vector union and axis enum are given below in figure 6.

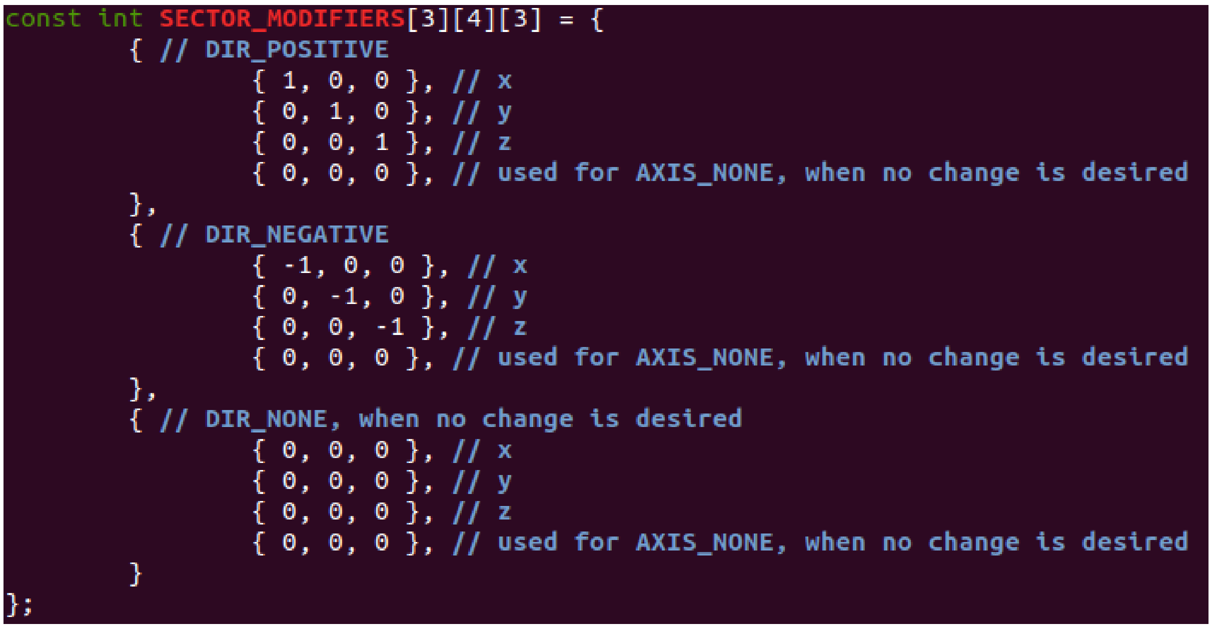


```
enum axis {
        X_AXIS = 0,
        Y_AXIS = 1,
        Z_AXIS = 2,
        AXIS_NONE = 3
};

// Can either access x/y/z directly, or access vals[X/Y/Z_AXIS]
union vector_3d {
        struct {
                double x;
                double y;
                double z;
        };
        double vals[3];
};
```

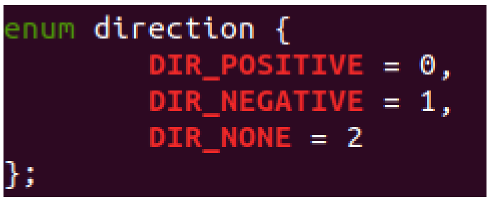
Figure 6: The definition of the axis enum and vector_3d union.

The partial crossing code in particular had many almost duplicate functions. The partial crossing code needs to iterate over all sectors adjacent to the one being processed, and there can up to 27 adjacent sectors if a 3D grid is used. This led to many different pieces of code that were almost identical except for handling different combinations of axis and directions. As example: there was one function to locate the sector adjacent positively along the x-axis and another to locate a sector adjacent positively along the y-axis. To solve this issue I created a direction enum and an array called SECTOR_MODIFIERS. The definition of these is given below.



```
const int SECTOR_MODIFIERS[3][4][3] = {
        { // DIR_POSITIVE
                { 1, 0, 0 }, // x
                { 0, 1, 0 }, // y
                { 0, 0, 1 }, // z
                { 0, 0, 0 }, // used for AXIS_NONE, when no change is desired
        },
        { // DIR_NEGATIVE
                { -1, 0, 0 }, // x
                { 0, -1, 0 }, // y
                { 0, 0, -1 }, // z
                { 0, 0, 0 }, // used for AXIS_NONE, when no change is desired
        },
        { // DIR_NONE, when no change is desired
                { 0, 0, 0 }, // x
                { 0, 0, 0 }, // y
                { 0, 0, 0 }, // z
                { 0, 0, 0 }, // used for AXIS_NONE, when no change is desired
        }
};
```

Figure 7: The definition of the SECTOR_MODIFIERS array.

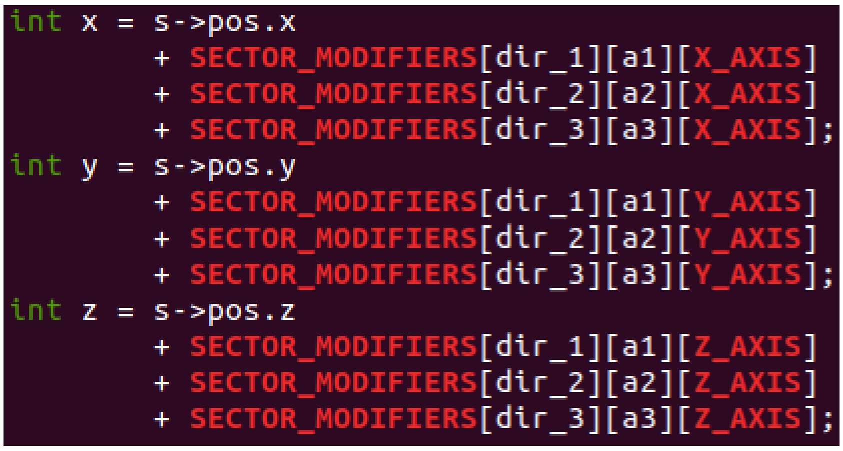


```
enum direction {
        DIR_POSITIVE = 0,
        DIR_NEGATIVE = 1,
        DIR_NONE = 2
};
```

Figure 8: The definition of the direction enum.

This SECTOR_MODIFIERS array and direction enum, along with the aforementioned axis enum, allowed adjacent sectors to be located programmatically. By specifying the direction (or directions), followed by the axis (or axes) of interest we could find the modifiers that needed to be added to the current sector's position to find the adjacent one. As can be seen in figure 7 the modifiers are non-zero only for the axes of interest. By including these zero values conditionals can be avoided as there is no longer a need to check if the axis is to remain unchanged.

A function was created to find adjacent sectors using specified axes and directions. Figure 9 shows how this function makes use of SECTOR_MODIFIERS. By passing a sector (*s* in the below figure) along with the directions on the axes of interest the sector adjacent in those directions can be found.



```
int x = s->pos.x
        + SECTOR_MODIFIERS[dir_1][a1][X_AXIS]
        + SECTOR_MODIFIERS[dir_2][a2][X_AXIS]
        + SECTOR_MODIFIERS[dir_3][a3][X_AXIS];
int y = s->pos.y
        + SECTOR_MODIFIERS[dir_1][a1][Y_AXIS]
        + SECTOR_MODIFIERS[dir_2][a2][Y_AXIS]
        + SECTOR_MODIFIERS[dir_3][a3][Y_AXIS];
int z = s->pos.z
        + SECTOR_MODIFIERS[dir_1][a1][Z_AXIS]
        + SECTOR_MODIFIERS[dir_2][a2][Z_AXIS]
        + SECTOR_MODIFIERS[dir_3][a3][Z_AXIS];
```

Figure 9: How SECTOR_MODIFIERS and the axis and direction enums are used together.

Say we wanted to find the sector that is adjacent positively on the x axis, negative on the y axis, and has the same coordinate in the z axis. To do so the following arguments would be passed:

- *a1* = X_AXIS
- *dir_1* = DIR_POSITIVE
- *a2* = Y_AXIS
- *dir_2* = DIR_NEGATIVE
- *a3* = AXIS_NONE
- *dir_3* = DIR_NONE

In this case *a3 and dir_3* are set to AXIS_NONE and DIR_NONE as we do not want to change the z coordinate.

These enums and data structures were extremely useful as they allowed for loops to iterate over combinations of axes and directions. While this approach is confusing at first it greatly reduces duplicate code and allows generic code to be written, leading to easier development and debugging.

5: Performance of the serial version with domain decomposition

As the MPI version is built on top of the serial version with domain decomposition I will examine their performance separately. It is important to understand how the serial version with domain decomposition scales as this will affect the performance of the MPI version.

5.1: Performance scaling vs number of sectors using a dense data set

As the number of sectors increases the performance scaling is likely to become smaller and eventually become negative. This is because as number of sectors grows more time must be spent checking for partial crossings, and the number of spheres transferring between sectors increases. Eventually the program will be spending more time transferring spheres between sectors than doing useful work that contributes to the results.

The data set was used the file *50000.spheres*, which contains just over 50’000 spheres. See appendix E for details of this data set. With this data set the naive serial version took 31435 seconds to complete, which is eight hours 44 minutes.

The below graph shows the speedup of the serial version with domain decomposition over the naive serial version. As can be seen using even a small number of sectors can give a large speedup over the naive serial version. However as was predicted the speedup eventually drops off, and then starts becoming smaller again.

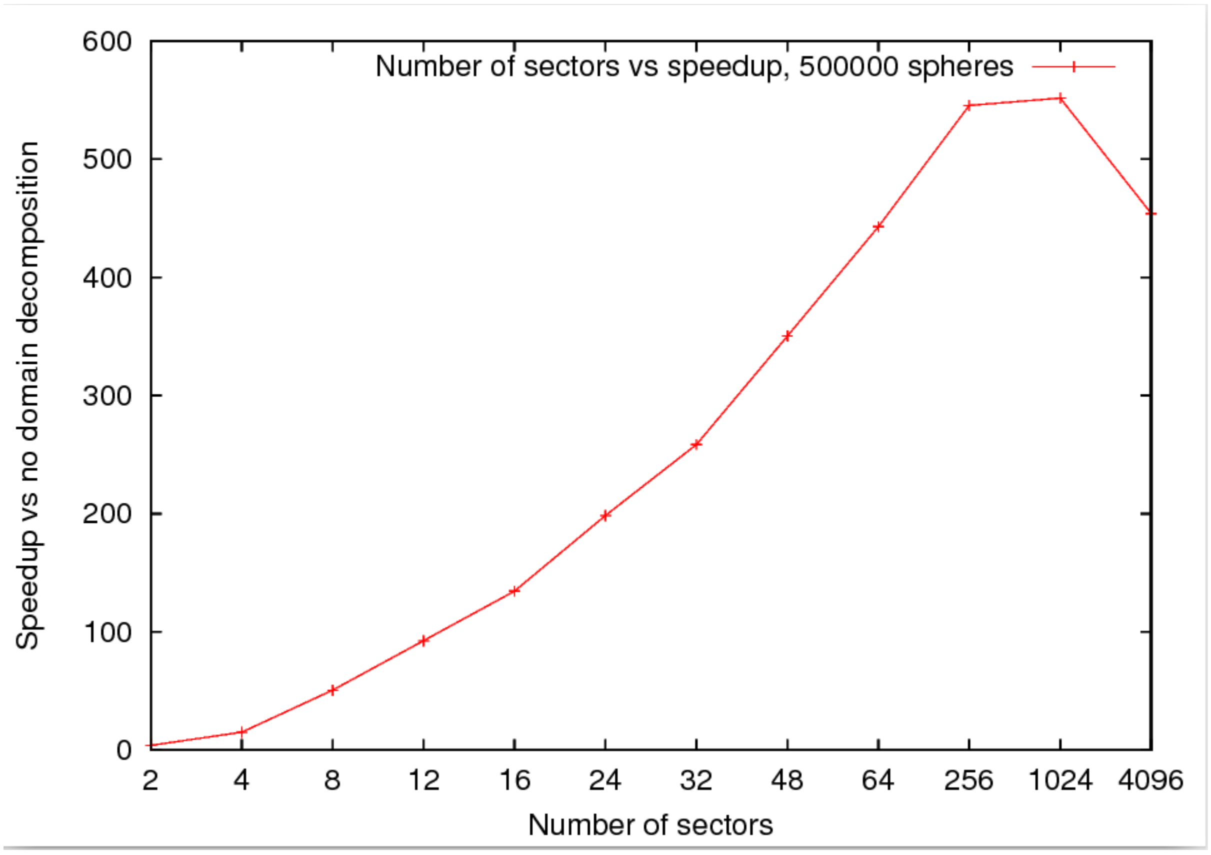


Figure 10: Speedup compared to naive serial version vs number of sectors.

Looking at the number of partial crossings and sphere transfers between sectors shows why this happens, as shown in the below table. The performance drops off right as the frequency of these two events starts to increase. For this given data set the simulation area is too finely divided when the number of sectors exceeds 256.

Number of sectors	8x8 (64)	16x16 (256)	32x32 (1024)	64x64 (4096)
Number of transfers	0	103	274	567
Number of partial crossings	8	16	23	42

With a small number of sectors the speedup achieved comes quite close to the theoretical maximum. This shows that my attempt to minimise the performance impact of partial crossings worked well. This is especially true when using 2 and 4 and sectors, as was mentioned earlier in section 2.3 the naive approach for handling partial crossings would negate any benefit domain decomposition brings in these cases. The comparison of actual speedup vs theoretical maximum speedup is given below:

Number of sectors	2x1 (2)	2x2 (4)	4x2 (8)
Actual speedup	3.9x	15.06x	50.7x
Maximum possible speedup	4x	16x	64x

5.2: Performance scaling with number of sectors and a sparse data set

The aforementioned results use an extremely dense data set. Spheres are located very close together and fill the simulation area evenly along the x and y axis. This is the best case scenario.

With a less dense data set performance will likely be worse. As the data set becomes less dense the chance of spheres colliding with each other becomes lower, so when domain decomposition is used most of the execution time may be spent transfering spheres between sectors.

The following table shows the speedup obtained using the *50000_sparse.spheres* data set.

Number of sectors	**Speedup**	**Number of sphere transfers**
2x1 (2)	0.21x	63
2x2 (4)	0.49x	116
4x2 (8)	1.13x	192
4x3 (12)	3.41x	139
4x4 (16)	2.91x	267
8x8 (64)	8.85x	576

As can be seen with a sparse data set the scaling is much worse than with a dense data set. Sometimes the program actually takes longer when domain decomposition is used. This shows that this domain decomposition approach is most suitable for dense data sets.

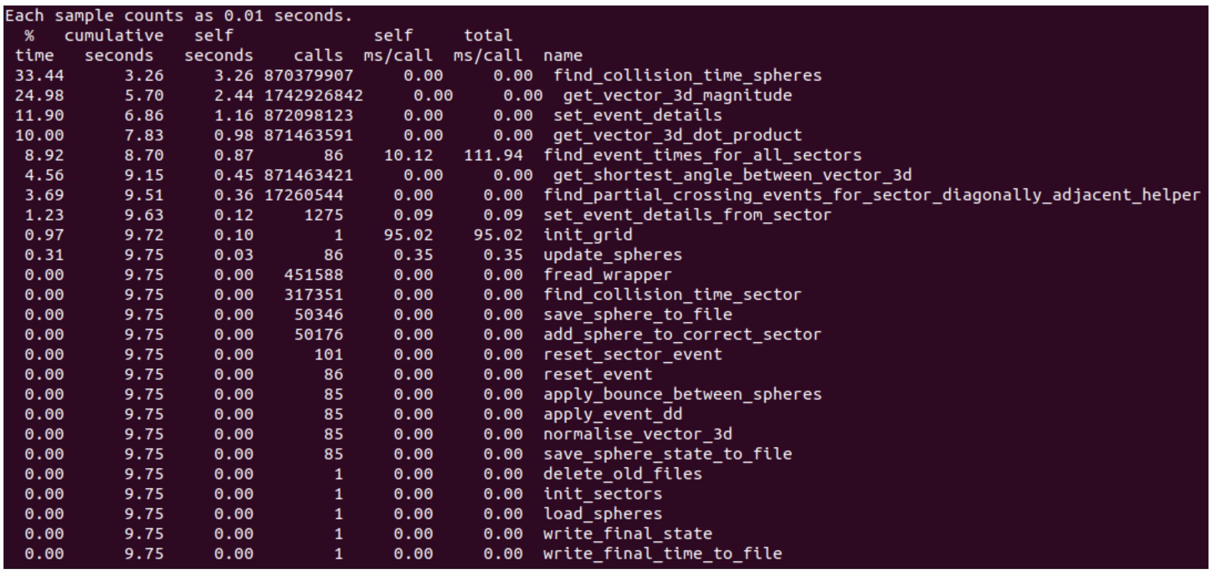
It is interesting to note that with a sparse data set larger number of sectors can provide larger speedups despite having more sphere transfers. While sphere transfers are more expensive, their cost can sometimes be outweighed by the time saved by using domain decomposition.

5.3: Profiler results

The following paragraphs refer to the *50000.spheres* data set, as used in section 5.1 above.

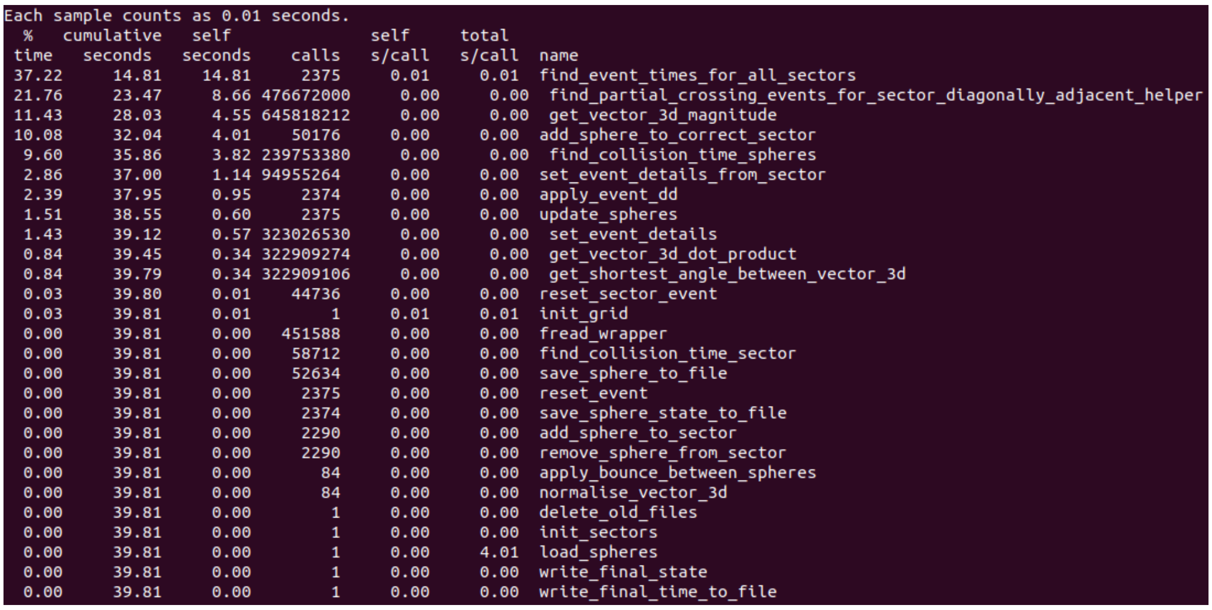
Gprof was used to profile the serial version with domain decomposition to determine where the program was spending most of its time executing. When the program is compiled with the *-pg* flag each function stores some data about where it was called from. Gprof can then use this information to estimate how much time was spent in each function[5].

Figure 11 below gives the profiler results using a 4x4 grid of sectors. As was mentioned earlier it is crucial to performance that the partial crossing code is only used when needed. This output looks promising. The functions that the program spent the most time in are all related to finding collision events between spheres, such as the several *vector_3d* functions which are essential to finding collision events. Very little time is being spent in the input/output functions or other administrative functions.


```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.44      3.26     3.26 870379907     0.00     0.00  find_collision_time_spheres
 24.98      5.70     2.44 1742926842     0.00     0.00  get_vector_3d_magnitude
 11.90      6.86     1.16 872098123     0.00     0.00  set_event_details
 10.00      7.83     0.98 871463591     0.00     0.00  get_vector_3d_dot_product
  8.92      8.70     0.87       86    10.12   111.94  find_event_times_for_all_sectors
  4.56      9.15     0.45 871463421     0.00     0.00  get_shortest_angle_between_vector_3d
  3.69      9.51     0.36 17260544     0.00     0.00  find_partial_crossing_events_for_sector_diagonally_adjacent_helper
  1.23      9.63     0.12     1275     0.09     0.09  set_event_details_from_sector
  0.97      9.72     0.10        1    95.02    95.02  init_grid
  0.31      9.75     0.03       86     0.35     0.35  update_spheres
  0.00      9.75     0.00   451588     0.00     0.00  fread_wrapper
  0.00      9.75     0.00   317351     0.00     0.00  find_collision_time_sector
  0.00      9.75     0.00    50346     0.00     0.00  save_sphere_to_file
  0.00      9.75     0.00    50176     0.00     0.00  add_sphere_to_correct_sector
  0.00      9.75     0.00      101     0.00     0.00  reset_sector_event
  0.00      9.75     0.00       86     0.00     0.00  reset_event
  0.00      9.75     0.00       85     0.00     0.00  apply_bounce_between_spheres
  0.00      9.75     0.00       85     0.00     0.00  apply_event_dd
  0.00      9.75     0.00       85     0.00     0.00  normalise_vector_3d
  0.00      9.75     0.00       85     0.00     0.00  save_sphere_state_to_file
  0.00      9.75     0.00        1     0.00     0.00  delete_old_files
  0.00      9.75     0.00        1     0.00     0.00  init_sectors
  0.00      9.75     0.00        1     0.00     0.00  load_spheres
  0.00      9.75     0.00        1     0.00     0.00  write_final_state
  0.00      9.75     0.00        1     0.00     0.00  write_final_time_to_file
```
Figure 11: The gprof results with a 4x4 sector grid.

It can be seen above that my attempt to minimise the time spent checking for partial crossing was successful. So little time was spent in partial crossing functions that most of them did not register in the profiler. The only partial crossing check which was used enough to register in the profiler was *find_partial_crossing_events_for_sector_diagonally_adjacent_helper* where the program spent approaximately 3.7% of its time. This is perfectly acceptable as it is a very small amount of the overall execution time.

Figure 12 below gives the profiler results using a 200x200 grid of sectors. In this case the function *find_partial_crossing_events_for_sector_diagonally_adjacent_helper* now takes up approximately 21.8% of the program's execution time. As the simulation area is so finely divided there is simply no way to avoid the need to check for partial crossings. This again shows how the serial version with domain decomposition fails to scale past a certain number of sectors.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 37.22     14.81    14.81     2375     0.01     0.01  find_event_times_for_all_sectors
 21.76     23.47     8.66 476672000     0.00     0.00  find_partial_crossing_events_for_sector_diagonally_adjacent_helper
 11.43     28.03     4.55 645818212     0.00     0.00  get_vector_3d_magnitude
 10.08     32.04     4.01    50176     0.00     0.00  add_sphere_to_correct_sector
  9.60     35.86     3.82 239753380     0.00     0.00  find_collision_time_spheres
  2.86     37.00     1.14 94955264     0.00     0.00  set_event_details_from_sector
  2.39     37.95     0.95     2374     0.00     0.00  apply_event_dd
  1.51     38.55     0.60     2375     0.00     0.00  update_spheres
  1.43     39.12     0.57 323026530     0.00     0.00  set_event_details
  0.84     39.45     0.34 322909274     0.00     0.00  get_vector_3d_dot_product
  0.84     39.79     0.34 322909106     0.00     0.00  get_shortest_angle_between_vector_3d
  0.03     39.80     0.01    44736     0.00     0.00  reset_sector_event
  0.03     39.81     0.01        1     0.01     0.01  init_grid
  0.00     39.81     0.00   451588     0.00     0.00  fread_wrapper
  0.00     39.81     0.00    58712     0.00     0.00  find_collision_time_sector
  0.00     39.81     0.00    52634     0.00     0.00  save_sphere_to_file
  0.00     39.81     0.00     2375     0.00     0.00  reset_event
  0.00     39.81     0.00     2374     0.00     0.00  save_sphere_state_to_file
  0.00     39.81     0.00     2290     0.00     0.00  add_sphere_to_sector
  0.00     39.81     0.00     2290     0.00     0.00  remove_sphere_from_sector
  0.00     39.81     0.00       84     0.00     0.00  apply_bounce_between_spheres
  0.00     39.81     0.00       84     0.00     0.00  normalise_vector_3d
  0.00     39.81     0.00        1     0.00     0.00  delete_old_files
  0.00     39.81     0.00        1     0.00     0.00  init_sectors
  0.00     39.81     0.00        1     0.00     4.01  load_spheres
  0.00     39.81     0.00        1     0.00     0.00  write_final_state
  0.00     39.81     0.00        1     0.00     0.00  write_final_time_to_file
```

Figure 12: The gprof results with a 200x200 sector grid.

6: Performance of the MPI version with domain decomposition

6.1: Performance with a dense data set

The following refers to the data set *50000.spheres*.

Figure 13 below shows the speedup given by the MPI version compared to the serial version with domain decomposition with the same number of sectors, where the MPI version has one core handle one sector. Example: the MPI version with 64 cores and 64 sectors is compared against the serial version with 64 sectors. The theoretical maximum speedup is the same as the number of cores, so with 64 cores the max speedup should be 64x. The red line is the speedup in *all help* mode. The blue line is the speedup in *neighbour help* mode. The green line represents the maximum possible speedup in *all help* mode.

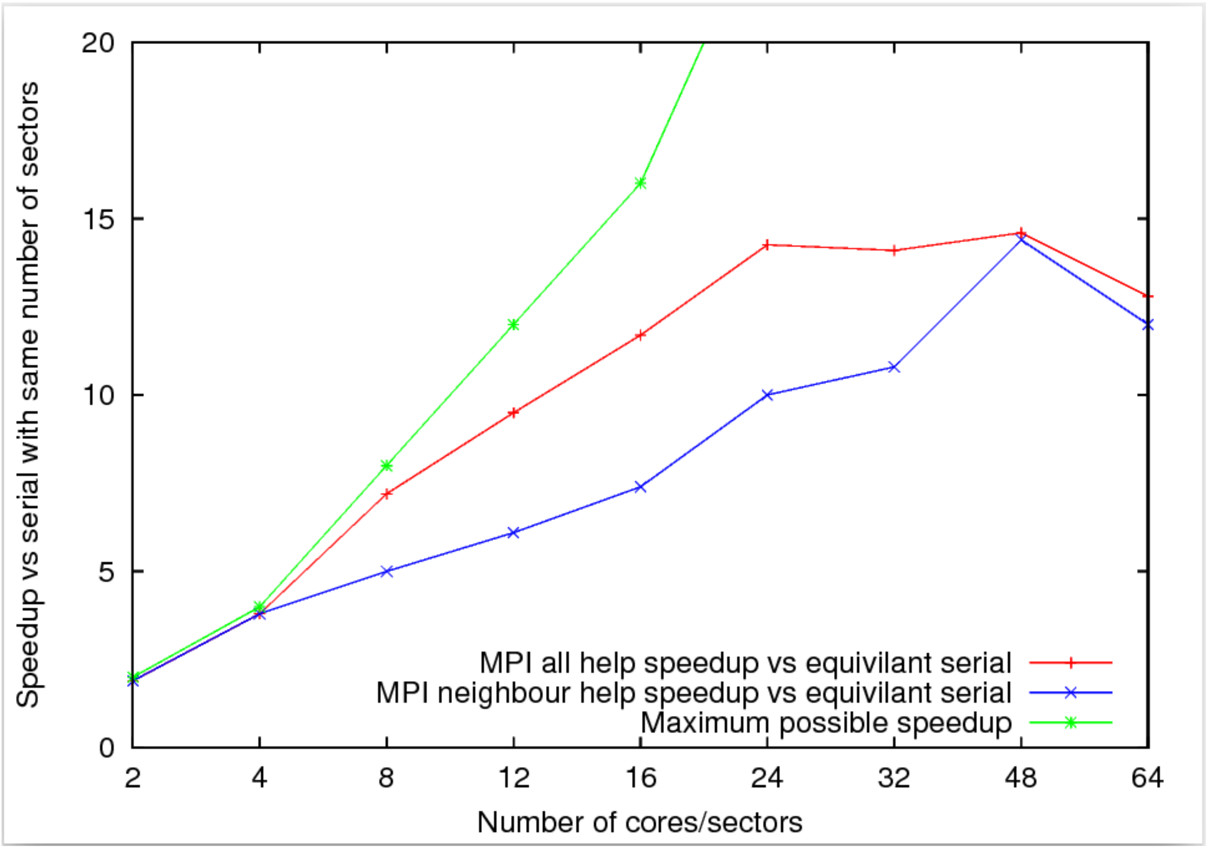


Figure 13: Speedup of the MPI version compared to the serial version with the same number of sectors.

The *all help* results will be discussed first. As can be seen the MPI version scales very well with a low number of cores. For two, four and eight cores the speedup comes very close to the maximum. However performance past here is disappointing, especially when run with 64 cores which is slower than the run with 32 cores.

Now to discuss the *neighbour help* results. For two and four cores the *neighbour help* results match the *all help* results. This was expected as in these two cases all sectors are neighbours, so regardless of which mode is used all sectors will be processed by all cores. As the number of cores and sectors grows so does the speedup provided in *neighbour help* mode. This is because as the number of sectors increases, so does the number of neighbours for sectors not at the edge of the grid, and the chance of a sector being at the edge of the grid (and therefore having less neighbours) becomes smaller. This speedup will get infinitely closer to expected maximum as more and more sectors are located away from the edge of the grid.

Interestingly with more than 32 cores the *neighbours help* speedup exceeds the maximum speedup I predicted in section 3.4. I think this may be because there is less cache pressure on each core, compared to the serial version which puts a lot of pressure on a single core's cache.

I believe the poor scaling in *all help* mode was due to the relatively small data set. Consider the MPI version with 64 cores and 64 sectors. With the data set used in obtaining figure 13, there is just over 50'000 spheres. This means there is only approximately 781 *(50'000/64)* spheres per sector. When *all help* mode is used past the first iteration this is then divided between 64 cores. This is simply too little work for each core. As the workload is finished so quickly, communication and file I/O ends up being a large amount percentage of the total running time which limits the performance scaling.

To test this theory I used more data sets featuring more spheres. This below table shows the scaling for the MPI version using 16, 32 and 64 cores and sectors compared to the serial version with the same number of sectors. As can be seen scaling improves as the MPI version is given larger data sets, and eventually the scaling is very good once the data set is large enough.

Number of spheres	**Speedup with 16 cores (max 16x)**	**Speedup with 32 cores (max 32x)**	**Speedup with 64 cores (max 64x)**
50'000	11.69x	14.09x	12.76x
500'000	12.56x	22.28x	29.86x
1'000'000	15.06x	24.21x	33.27x
2'500'000	15.43x	24.57x	42.58x
5'000'000	N/A	N/A	50.88x
10'000'000	N/A	N/A	56.62x

Note that 16 and 32 core runs of the 5'000'000 and 10'000'000 spheres data sets were not performed due to time constraints. 10'000'000 spheres took 67 hours to process using the serial version with 64 sectors, so the time taken with 16 or 32 sectors would likely exceed the time limit on Lonsdale. For this same reason I was unable to test with even larger data sets.

6.2: Performance with a sparse data set

As was mentioned in section 5.2, the serial version with domain decomposition has poor scaling with sector count when the data set is sparse. I decided to investigate how the MPI version scales with a sparse data set. As in section 5.2 the *50000_sparse.spheres* data set was used. The below table gives these results.

Number of cores and sectors	2x1 (2)	2x2 (4)	4x2 (8)	4x3 (12)	4x4 (16)
Speedup	1.96x	3.77x	7.28x	10.3x	12.07x

These speedups are very similar to the speedups shown in figure 13, which showed the speedup of the MPI version using a dense data set. From this I conclude that the MPI version does not require the data set to be dense in order to achieve a good speedup.

7: Analysis of results and conclusion

7.1: The limits of domain decomposition and its usefulness

In section 1.1 I stated that one of the questions this project sought to answer was: “How much can domain decomposition be exploited to achieve a speedup in a serial program?”.

After analysing the results it is apparent that domain decomposition cannot be exploited indefinitely to achieve further and further speedups. As the number of sectors grows eventually sector transfers and partial crossings, which are expensive to handle, make up a vast majority of the events that must be processed.

However I conclude that depending on the data set domain decomposition can be extremely worthwhile to use. As figure 10 shows, domain decomposition still provides significant speedups despite eventually plateauing. However as discussed in section 5.2 domain decomposition is best used with dense data sets, but can still be useful with sparse data sets.

7.2: Is an MPI version worthwhile?

In section 1.1 I stated that one of the questions this project sought to answer was: “Is it worthwhile to use MPI to parallelise a domain decomposition system? Or can the same effect be achieved more easily by simply diving the simulation area in the serial version into a very large number of sectors?”.

Consider figure 10 again. It can be seen that as the number of sectors grows the speedup for the serial version levels off and eventually decreases. With 64 sectors the domain decomposition serial version was 442x faster than the naive serial version. When this is scaled to 64 cores with MPI the result is over 5’000x faster than the naive serial version. This is far faster than than the speedup achieved by simply increasing the number of sectors.

In addition, as was mentioned the scaling of the MPI version improves as the data set’s size is increased, so the total speedup over the naive version with a large data set would be even larger. The data in the prior paragraph refers to the *5000.spheres* data set, with which the MPI version had poor scaling when 64 cores were used. If a larger data set is used the total speed up over the naive serial version becomes much larger. Assuming the same 442x speedup using 64 sectors, and the 56.6x scaling seen with 10’000’000 spheres when 64 cores are used, the total speedup over the naive serial version should approach 25’000x.

Also as was discussed in section 6.2, the MPI version provides significant speedups for both dense and sparse data sets. Therefore I conclude that the MPI version is extremely worthwhile to use, especially for very large data sets.

The MPI version in *neighbours help* also allows the memory load to be distributed while still providing some speedup. Finally there is a further collision optimisation that reduces execution time at the cost of higher memory usage. The MPI version allows this memory load to be distributed between physical nodes, while with a serial version there would be a significant memory load on just one node. This optimisation is described later in section 9.1.

From all of these points the author concludes that an MPI version provides significant speedups past what can be achieved with domain decomposition alone in a serial program, and is extremely worthwhile.

7.3: Accuracy

Note: doubles were used for all floating point storage in this program.

One issue with this approach that I discovered while developing this project is accuracy. The domain decomposition approach slightly changes the results. This happens because floating point operations are not associative and the domain decomposition system changes how a sphere moves through the simulation area.

Consider a single sphere bouncing from one edge of the simulation area, hitting the opposite edge and then bouncing back. With a variable time step this means the sphere effectively moves just between the two edges of the simulation area in the naive serial version. However with domain decomposition the sphere is “stopped” at sector boundaries, and then moves to the simulation boundary in the next iteration. As this changes the order of the floating point operations it introduces tiny errors that grow larger each time a sphere is stopped at a sector boundary.

The sector invalidation approach also introduces a small error. If a sector’s time is still valid, then on the next iteration its time is reduced by the overall soonest time. This may slightly change the sector’s soonest event time, especially if it is valid for many iterations and is subtracted from many times.

While testing the error did not exceed 10^{-10}. However if a simulation was to be run for a very long time then this error could grow to an unacceptable level. Arbitrary precision libraries could be used to decrease the error, but these would be slower than using doubles. If this error is a concern then perhaps the approach described in this project is suitable only when the underlying system already uses arbitrary precision libraries and the domain decomposition code is adapted to use these libraries..

8: Testing and validation

8.1: Testing and validating basic collisions

The first part to be tested and validated was the serial collision detection and handling code. This was initially done by solving problems on paper and checking against the simulation results. As a simple collision model was used it was not hard to solve the problems using pen and paper. However this approach had the downside of needing to check the results using a debugger specifically each time I wanted to validate the collision code.

Later I added a small number of test functions in the code. These would create two spheres with specific velocities and positions and the simulation would be run. The time of collision between the sphere is then found and checked against a provided expected value. After the collision occurs the velocities of the spheres are then checked against known values obtained by solving the problem on paper. This allowed me continuously test the collision code while I developed it. I could simply run the program and it would print whether or not the tests passed.

As mentioned earlier the simple collision model being used assumes that no energy is lost during the collision. This allowed me to add additional test checks which ensured that the total energy and momentum in this system did not change after the collision occurred. This was a quick and easy way of ensuring that the system was behaving correctly.

8.2: Comparing against the naive serial version

While working on the serial version with domain decomposition and the MPI version I wanted to be able to ensure that their results matched those produced by the naive serial version. At first I had each version save the final state of its simulation to a text file, which I then checked manually against the text files saved by other versions. While this worked it was extremely tedious and is not usable when the number of objects in the simulation is large.

To better support finding differences in results produced by each version, I added some optional comparison features to each version. All versions of my code can optionally save the final state of the system to a specified binary file. Then another version of the program can accept this file as an argument, and will compare the final state of its objects against those saved in the file. When doing this the maximum velocity and position errors are found and printed.

This allowed the final state of the naive serial version to be saved for later comparison in the other versions. This was invaluable for finding issues due to how easy it was to use. If any errors were present they were immediately visible in the output of the program.

8.3: Using the visualiser for debugging

While the comparison feature could tell me that an error existed, it gave no hint as to why that error existed. To allow for easier debugging a simple OpenGL visualiser was created.

This was immensely helpful. It showed exactly what was going wrong in the simulation code, and allowed ideas to be formulated about why the error was occuring.

One particular area where the visualiser was useful was in validating the partial crossing code. I created scenarios where spheres would partially cross over sector boundaries and used the visualiser to display the simulation results. Bugs in the partial crossing code were obvious as spheres would fail to collide over the sector boundaries and would instead pass through one another. By setting up the tests so that collisions occur along many axes at once I could verify the partial crossing code worked for each axis.

This can be seen in figures 14 and 15 below. In this test the simulation area is divided into a 3x3x3 grid. Spheres in the outer sectors move towards spheres in the inner sector and partial crossings take place.

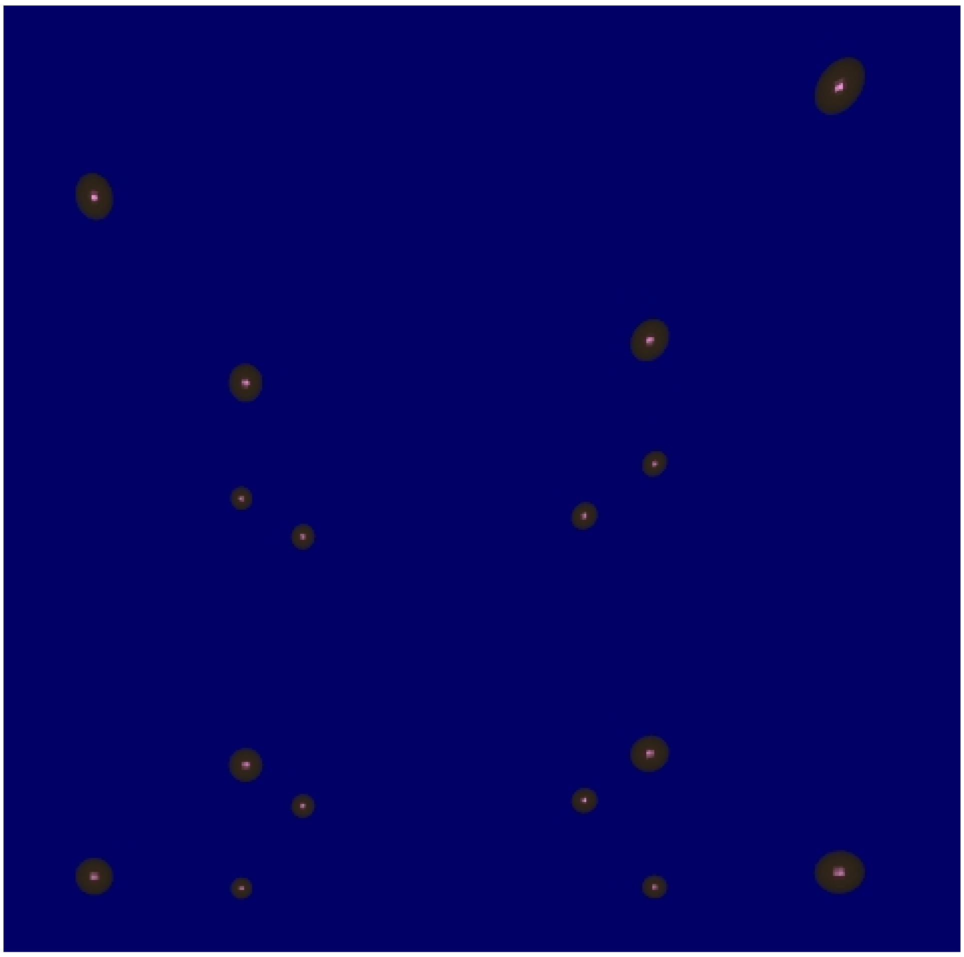


Figure 14: The spheres are about to collide over sector boundaries.

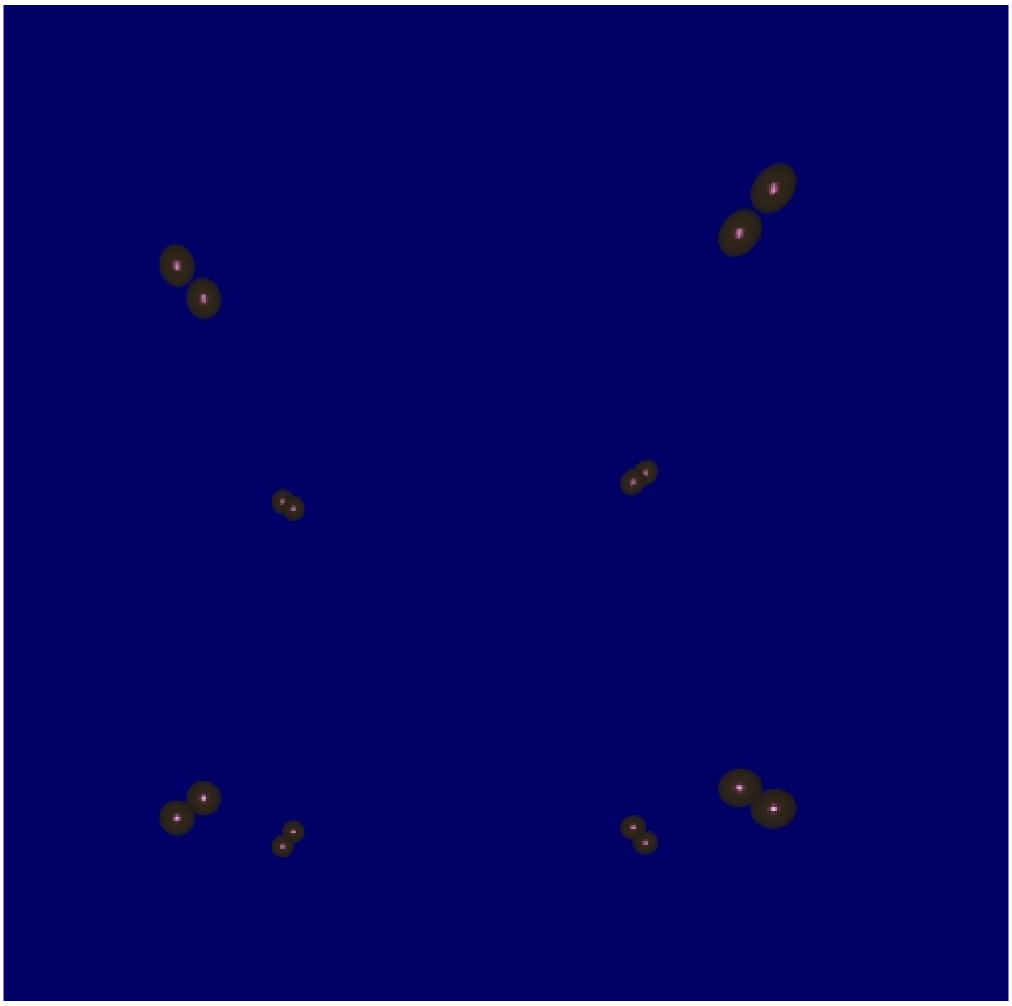


Figure 15: The spheres have collided over sector boundaries. This confirms the partial crossing code works well.

8.4: Testing with electric fence

Electric Fence[6] was used several times while developing this project. While it does not validate that the data is correct, it is a useful tool for ensuring that buffers are not being overrun and that memory is not being used after *free()* has been called. This helps to find why obscure issues occur and prevent seemingly random crashes from occuring. Thankfully Electric Fence did not ever detect any issues in this project, which allowed me to focus on searching other areas that might have been the reason for incorrect results.

8.5: Runtime sanity checks for debugging

One issue that occured throughout the project was that new features would add bugs where spheres would fail to correctly transfer to different sectors. This led to incorrect results that were strange and very hard to debug. This was because a sphere in an incorrect sector could cause issues in a much later iteration rather than immediately causing issues when it failed to transfer.

To catch these issues when they occur I added an optional sanity check function. In the serial domain decomposition version this simply checks that each sphere is within the bounds of the sector it belongs to, and prints and error and stops the program if not.

In the MPI version I sometimes had issues where processes ended up with different views of which spheres were in which sectors. As an example: a process controlling a sector might remove a sphere that is transfering from its own sector to another sector, but the process controlling the destination sector or any other neighbouring processes would not register that the sphere transferred. To catch this issue each MPI process checks that spheres are correctly inside not just its own sector, but also those that neighbour it.

This sanity check function was invaluable in detecting these errors and aided in finding many bugs.

8.6: Known issues

The code does not check that each sphere occupies a unique area initially. It is up to the user to ensure that spheres do not overlap in their initial state. If any spheres overlap initially then the behaviour of the program is undefined.

The code does not support the scenario where a sphere is large enough to overlap into sectors that are not immediate neighbours. Example: a sphere that is so large it exists in the sector responsible for it and extends into this sector’s neighbour, and this neighbour’s neighbour. Overlapping into neighbours is fine as the partial crossing code handles this. This can happen if an very large sphere is used, or if the simulation area is too finely divided using domain decomposition.

The MPI version will fail if the nodes in the cluster do not have unique hostnames, or if the nodes have blank hostnames. Hostnames are used to determine which processes live on the same physical node so that shared memory can be used. Without unique hostnames the MPI processes will incorrectly believe that some other processes live on the same node and will try to use shared memory to access their data. The Lonsdale cluster has unique hostnames for each node so this issue was never encountered while testing.

9: Further work

9.1: Individual sphere invalidation

Rather than invalidate an entire sector or two when an event occurs, instead only the spheres involved in the event could be invalidated. This reduces the execution time as on each iteration past the first many less pairwise checks need to be done. However collision details between sphere pairs now need to be kept. This is needed as once the sphere is invalidated we need to know which other spheres had the next soonest event prior to invalidation. This optimisation would greatly increase performance at the cost of much higher memory usage.

Consider the naive serial version: once the invalidated sphere (or spheres) is known only this sphere needs to be checked against all other spheres in the simulation. This has linear growth, which is far faster than the original quadratic growth. However the memory required grows at an $O(n^2)$ rate. This would likely be unacceptable for any large simulation.

In the domain decomposition serial version, the invalidated sphere (or spheres) need to only be checked against other spheres in the same sector, and possibly checked for partial crossings. Again this is linear growth. With domain decomposition the memory required grows at an $O(n^2/s^2)$ rate assuming an even sphere distribution, where s is the number of sectors.

The MPI version is quite useful for this optimisation as sector only need to track collision details between their own spheres, meaning each process will keep this data only on its own physical node, spreading the memory load over every physical node. However as this reduces the work that needs to be done the data set would need to be even larger to achieve good speedups with high core counts. The *neighbours help* mode is particularly suited for this optimisation as it helps spread the memory load even further.

9.2: Extending the neighbours help mode

As was mentioned earlier the MPI version offers a neighbours help mode, with the intent of distributing the memory load across all nodes while still offering some parallelisation benefits. As was also mentioned earlier, if neighbours are on the same physical node they will use shared memory so there is less duplicate data using up memory.

This can be further exploited to allow more nodes in neighbour help mode to share the workload without increasing memory usage. Consider figure 16 below. Using neighbours help as is, if rank zero's sector is invalid then its neighbours, ranks one, four and five, will help it.

However as ranks four and five have their own copy of the spheres owned by rank zero, they could make this data available to all other ranks on physical node one by using shared memory, allowing them to help too. Likewise ranks two and three on physical node zero could help as well by using the shared memory already used by ranks zero and one. Doing this would increase the number of helping nodes from three to seven without increasing the memory used. Therefore this is definitely a worthwhile feature to add, and I would have been added had time permitted.

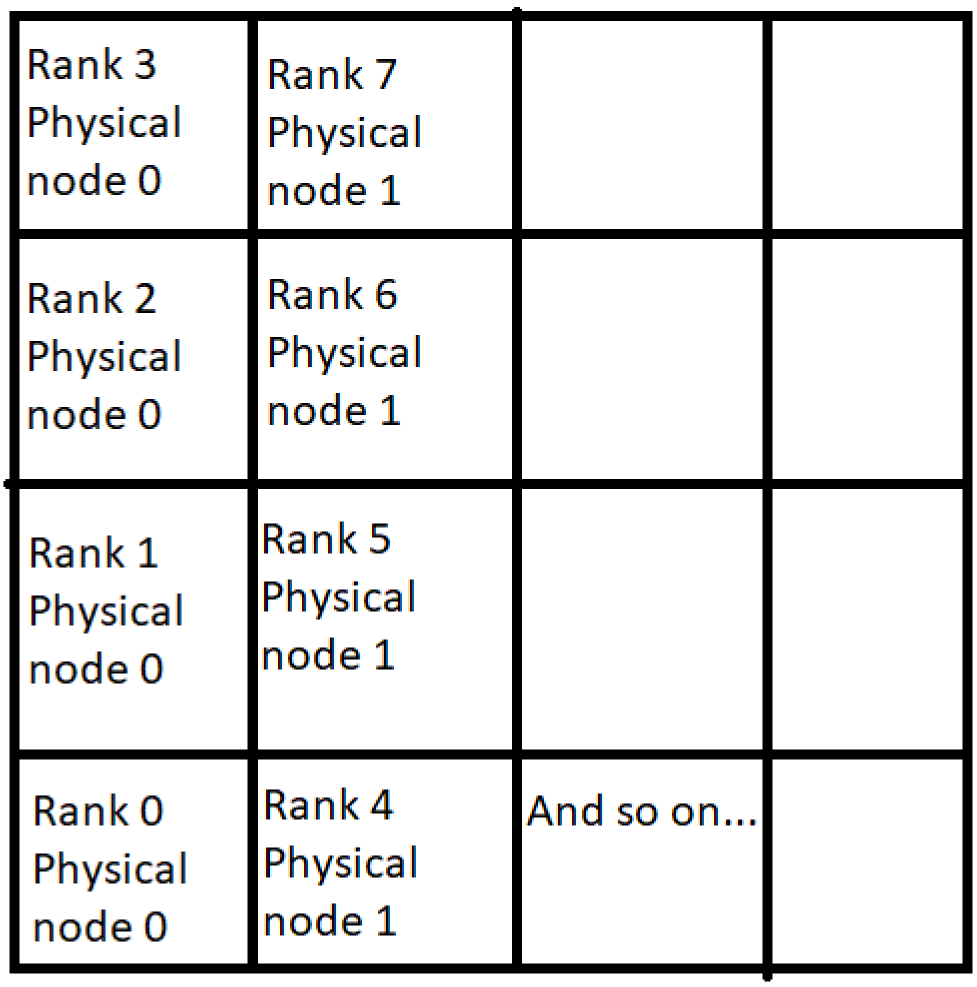


Figure 16: Showing which ranks and on what physical nodes are neighbours to rank 0.

9.3: Load balancing during simulation

I had originally planned to implement real time load balancing. This would adjust the size of the sectors if it was found that some sectors had significantly more spheres than others. As mentioned previously if the amount of spheres in each sector becomes highly imbalanced then scaling suffers greatly. Figure 17 below shows what the effects of load balancing may look like if all spheres happened to bunch up in one sector.

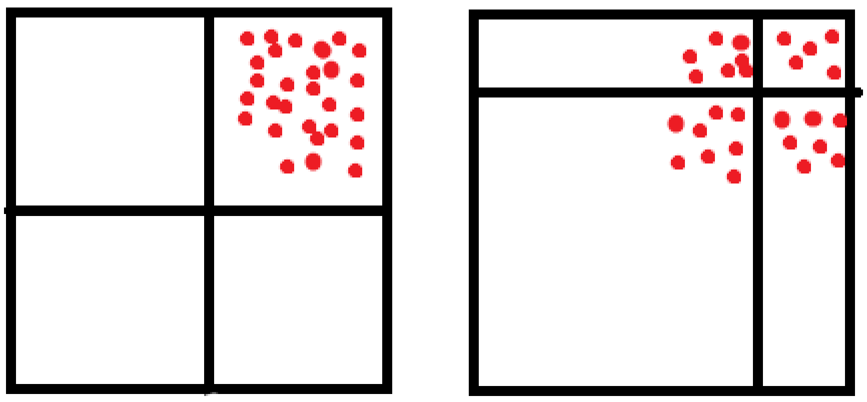


Figure 17: Before and after load balancing has taken place.

However towards the end of the project it was found that the domain decomposition implementation was working quite well, so there was little need for this feature to be added to ensure good performance. Also it was unlikely that such a large feature could be added and debugged in the time that remained for this project. Finally no acceptable solution was found for implementing load balancing in the MPI version. It was quite possible that the time taken to resize the sectors and move spheres between MPI process may be too much and actually increase the total time taken. However this feature may be desirable for certain workloads and given more time to work on this project it would definitely be implemented.

Bibliography

[1] Andrew Petersen
Broad Phase Collision Detection Using Spatial Partitioning
http://buildnewgames.com/broad-phase-collision-detection/

[2] University of Freiberg
Collision Detection based on Spatial Partitioning
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.298.5241&rep=rep1&type=pdf

[3] Josh Barnes and Piet Hut
A hierarchical *O(N log N)* force-calculation algorithm
https://www.nature.com/articles/324446a0

[4] Tom Ventimiglia and Kevin Wayne
The Barnes-Hut Algorithm
http://arborjs.org/docs/barnes-hut

[5] GNU gprof
https://sourceware.org/binutils/docs/gprof/

[6] efence(3) - Linux man page
https://linux.die.net/man/3/efence

[7] 3D collision detection
https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection

[8] Pool Hall Lessons: Fast, Accurate Collision Detection Between Circles or Spheres
Joe van den Heuvel and Miles Jackson
https://www.gamasutra.com/view/feature/131424/pool_hall_lessons_fast_accurate_.php

Appendix

Appendix A: Command line arguments

The serial and MPI versions both support the following command line arguments:

-x, -y, -z:
Optional.
Sets the number of slices in the specified axis.
Defaults to 1. If all are set to 1 domain decomposition is not used.
Must be at least 1, must equal number of processes in MPI version

-c:
Optional.
Sets the compare file. This should be a final state file from a previous run.
The max error between this run and the compare file will be printed.

-f:
Optional.
Sets the final state file. This will contain only the final velocity and position of each sphere.

-o:
Required.
Sets the output file. This contains all data needed to make use of the simulation.

-i:
Required.
Sets the initial state file.

-l:
Optional, but must be set if -e is not used.
Sets the time the simulation will run for.

-e:
Optional, but must be set if -l is not used.
Sets the limit on the number of non-transfer events.

-t:
Optional. Only used in serial version.
Runs some tests which verify the collision system works.

Appendix B: Output file format

The output file (specified with *-o*) is a binary file that contains the initial sphere data, and all changes to the sphere's that occur during the simulation. Rather than store the complete state at each iteration it instead stores data about what has changed in each iteration. This greatly reduces the file size.

The header consists of the size of the simulation area in the x, y and z axis stored as doubles, followed by the number of spheres as a 64 bit integer. After this the radius and mass of each sphere is stored. This is kept in the header as it does not change throughout the simulation and thus does not need to saved again after an iteration. This radius and mass data is not saved alongside a sphere id, but the position of the data gives the sphere id. For example: the first piece of radius and mass data belongs to sphere 0, and so on.

After each iteration there may be one sphere which has changed (as happens with a grid boundary impact or a sector transfer) or two spheres which have changed (as with a sphere on sphere collision or a partial crossing). When writing changes after an iteration, the iteration number is written (64 bit integer) followed by the time elapsed (double) in the simulation. Then the number of affected spheres is written (64 bit integer), followed by the id (64 bit integer) and post-iteration velocity and position (doubles) of each affected sphere.

Following the header a special version of the above is written. The number of affected spheres is set to the total number of spheres, the time elapsed is set to 0, and the initial state of each sphere is written. This serves as the initial state of the simulation. Following this the above is written as normal using either one or two affected spheres per iteration.

Appendix C: Final state file format

The final state file has a simple binary format. It is simply the number of spheres (64 bit integer) followed by velocity and position (doubles) of each sphere. The sphere id is not saved, instead each sphere is saved ordered by id, so the first sphere is the sphere with id zero, and so on.

Appendix D: Data set file format (“.spheres”)

The header of the data set file format consists of the simulation area’s size in each dimension, followed by the number of spheres.

After this comes a block of spheres. Each sphere has a 64 bit ID, its position in each axis, its velocity in each axis, and finally its mass and radius.

Appendix E: Data sets used for testing

All data sets were generated from the file *config_generator.c* and saved in a “*.spheres”* file. See appendix D for details of this file format.

Note: the run time for each data set is small as they are extremely computationally intense and it would take too long to simulate for longer periods of time.

50000.spheres:
50’176 spheres in a grid of size 500 in each axis.
Run for 0.00118 seconds.

Intended to test the best case scenario with a dense data set.

50000_sparse.spheres:
50’176 spheres in a grid of size 50’000 in each axis.
Run for 0.01018 seconds.
Intended to test the worst case scenario with a sparse data set.

500k.spheres:
50’1264 spheres in a grid of size 1750 in each axis.
Run for 0.0023952 seconds.
Intended to test the best case scenario with a dense data set.

1mil.spheres:
1’000’000 spheres in a grid of size 2200 in each axis.
Run for 0.0009917 seconds.
Intended to test the best case scenario with a dense data set.

2mil.spheres:
2'502'724 spheres in a grid of size 4000 in each axis.
Runs for 0.0009 seconds.
Intended to test the best case scenario with a dense data set.