CSE 633

Parallel Algorithms

Major Project Report

Sphere Packing in a Cuboid

Guided By:

Vipin Chaudhary

Prepared By:

Meghna Verma

Introduction:

This project is based on a paper "Parallel greedy algorithms for packing unequal spheres into a cuboidal strip or a cuboid" by Timo Kubach, Andreas Bortfeldt, Thomas Tilli and Hermann Gehring.

The purpose of the project is to study 3-D sphere packing problem portion of the paper the way it is done in the same. The problem presented in the paper was to check if a finite number of unequal sized spheres will fit in a given sized cuboid and pack them in the best density possible so that the wasted space is minimised. The same is done in this project by implementing the given logic that is used in the paper using C. It is also parallalised using OpenMP. The approach used in the paper is greedy method as whenever a sphere is placed, it is placed at the best possible location (using hole degree calculation) and then in the next step, again the same is repeated without changing the previously packed spheres (unlike the conventional sphere packing problem which tries to find the absolute best possible placement).

The Basics about the problem –

The difference in this project from the traditional sphere packing problem is as follows:

The traditional sphere packing problem has a given set of (mostly equal sized) spheres where we have to place them in such a way so that the wasted space is minimised and the density is maximum that it can be. As we have all read in middle school, in such a case, the maximum density in case of equal sized sphere is 74% or 0.74. This problem isn't as interesting as the original sphere packing problem in the way that in this case the cuboid is pre-defined and it might actually be too big for the spheres and there will be empty space left in that case which will reduce the placement density if we compare it with the traditional way. But the conventional sphere packing problem is all the more complicated than this one because it finds the absolute best way to place the spheres as opposed to this one which follows a greedy approach. But in this problem as well, the solution is done so that the sphere is placed at the best location possible at every step hence the spheres that are placed already, if enclosed, will have a really good packing density.

This problem is an NP Hard problem which means that the solution to it cannot be found in polynomial time but it can be verified in polynomial time. For the particular problem of sphere packing with unequal spheres, only a few solutions have been known till now. The previous solutions to this problem have been: simulation of spheres falling in gravity method and a dynamic programming problem which is actually only a special case for the purpose of radio surgical treatment. After these initial solutions, a few more solutions have been suggested. The one that is adapted in this algorithm is by Huang et al which was given in 2005. They did it for 2D and the writers of the referred paper adapted the solution for 3 dimensions.

Key Terms for building the algorithm:

There are some basics that need to be defined in order to specify the solution of this problem:

Let a sphere be S with center coordinates (x_i,y_i,z_i) then,

- **Placement P:** A placement $P = (i,x_i,y_i,z_i)$ means that the i^{th} sphere is placed with its center at (x_i,y_i,z_i).

- **A feasible placement:** It is a placement if the sphere lies completely inside the cuboid and doesn't collide with any other spheres that are already placed.

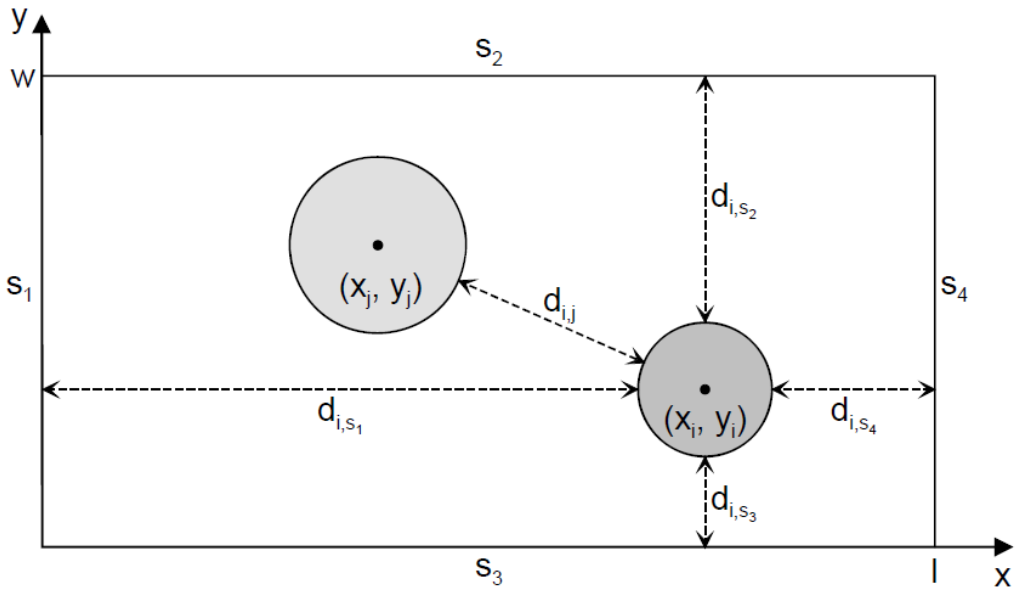
- **A corner placement:** A placement such that the sphere touches at least two items ie another sphere or an edge of the cuboid.

- Let u and v be the two items that are touching the sphere i.

- Then the **hole degree** λ of the corner placement p is defined as

$$\lambda = 1 - d_{min}/r_i$$

  Here, r_i is the radius of circle i while d_{min} is the minimal distance from circle i to other circles of P and the edges with the exception of items u and v.



The hole degree of a corner placement shows how close to other circles of a plan a given circle is accommodated. The higher the mean hole degree of the placements of a packing plan, the higher the density. So, the following rule is followed in finding the best possible placement of a sphere.

Maximum hole degree rule: It says that given a set of possible corner placements, the placement with maximum hole degree should be selected as the next one.

The Algorithm:

The algorithm used in the paper has a frame procedure that creates initial packing plans to place first two spheres in the best possible way and then there is core procedure that places the remaining spheres one by one in the best location possible.

The Frame Procedure in the algorithm:

```
Procedure B1.6_3DKP(in: instance data I, parameter τ, out: packing plan P_best)
P_best := ø;
for (every initial plan P_init) do
        P_res := B1.6_3DKP_C(I, τ, P_init);
        if (d_p(P_res) > d_p(P_best)) then
                P_best := P_res;
                // stop immediately if a global optimal solution is reached
                if (P_best includes all given spheres) then return P_best; endif;
        endif;
endfor;
return P_best;
end.
```

The frame procedure tries to place first sphere at the corner of the cuboid and places another sphere just touching it.

The Core Procedure in the algorithm:

```
Procedure B1.6_3DKP_C(in: instance data I, parameter τ, inout: packing plan P)
determine list L of possible corner placements p = (i, x_i, y_i, z_i)
regarding (incomplete) packing plan P and calculate the hole degrees λ(p);
while (there are corner placements in L) do
        select placement p* with the maximum hole degree λ_max from L;
        if (λ_max > τ) then
                P := P ∪ {p*}; // implement placement p*
                update list L;
        else
                best_density := 0;
                for (each corner placement p in L) do
                        let P' be a copy of P and L' be a copy of L;
                        P' := P' ∪ {p}; // implement placement p tentatively
                        update list L';
                        P'' := B1.6_3DKP_C2(I, L', P'); // complete plan P'
                        if (P'' includes all given spheres) then P := P''; return P; endif;
                        if (d_p(P'') > best_density or
                           d_p(P'') = best_density and λ(p) > λ(p*)) then
                                p* = p; best_density := d_p(P'');
                        endif;
                endfor;
                P := P ∪ {p*}; // implement placement p* finally
                update list L;
        endif;
endwhile;
return P;
end.
```

The Core Procedure finds the best placement of a sphere by calculating the minimum distance (ie maximum hole degree) for every corner placement and if the sphere can be placed within the cuboid following the rules, the placement plan is updated.

My Modifications in the Algorithm:

In frame procedure: After studying sphere packing problems, I found out that if I arrange the spheres in the decreasing order and place the first two after that in the corner touching each other side by side, it would be the best possible placement from the point of view of density and I wouldn't have to scan through all the possible placements to find the best one. (But, the disadvantage of doing so was that this is the only portion of the algorithm that the writers had parallelised because it didn't have dependencies and this process, although makes algorithm a little better, it reduced my scope for parallelisation.)

In Core Procedure: To find the best possible location, I scanned the corner placements with the calculation of density at the same time rather than first finding the possible corner placements and then finding the densities as done in the paper. I did this to facilitate parallelisation in the way that there is more computation in the portion to run in parallel. Also, the procedure that I followed was to place a sphere above each sphere and then rotate it all around touching that sphere itself and find the position with the minimum (and feasible) mean distance to find the best possible location.

The Final Algorithm Used:

Main:

Enter size of cuboid (l, w, h)
Enter number of spheres less than 500 (n)
Radii from 1 to 10 in length are assigned randomly (r[ ])
Radii are sorted
FrameProcedure (r[0], r[1])
    If (FrameProcedure returns True)
        Do
        For all radii in the array
            CoreProcedure (r[i])
            If CoreProcedure returns False, break
        Do end
    If end
If CoreProcedure is False,
    Print (“Failure”)
Else
    Print (“Success and Time taken”)

Frame Procedure (r[0], r[1]) :

Place first sphere at the left bottom corner of the cuboid
Place second sphere touching first sphere and the bottom edge of the cuboid
If both fit inside the cuboid
    Return True
Else
    Return False

Core Procedure (r[i]) :

For every placed sphere in the cuboid, do
    Place the i^{th} sphere just on the right of the placed sphere
    While (angle<pi) Do
        Rotate the tentative placement by 0.1 radian
        Check if the location fits inside the cube and doesn't collide with placed, then,
            For all the placed spheres
                Find minimal distance (max lambda)
                If the d_{min} is now less than previous shared value,
                    Change the tentative location to new value
                If End
            For End
    While End
End For
If a location to place a sphere is found
    Return True
Else
    Return False

Explanation of this function-

This procedure finds the best possible placement of the sphere by rotating it around an initial placement so that the best density is achieved.

For example, the third sphere will first be placed just on the right of first sphere but it will then collide with the second sphere, so it will be rotated till it doesn't collide with the other placed sphere and then d_{min} will be found for each possible placement (as the angles are varying) and the tentative placement will be updated. Then in the next iteration of for loop, it will be placed just on the right of the second placed sphere and in this case, if the d_{min} found is lesser than that found in placing it near the first sphere, the location will be updated. And again, it will be rotated touching the second sphere by 0.1 rad each time to

find the location that best suits the d_{min} rule. If a location is found then the global data structure containing the placed spheres will be updated and true will be returned by the core procedure. If no possible location is found, the initial values of the tentative locations have been set to -1 which won't change so we will know and false will be returned by the function.

Data Structures and Important Variables used in the Program:

Global:

To store the placed spheres: placedx[500] , placedy[500], placedz[500]

To store coordinates and radii of the spheres: x[500] , y[500], z[500], r[500]

No of spheres: n

Time calculated: ttime

Dimensions of the cuboid: l, w, h

In Core Procedure:

To store intermediate possible placements: tempx, tempy, tempz, tentx1, tenty1, tentz1

To store actual tentative placement: tentx, tenty, tentz

To store d_{min}: mindist (the actual variable that is used to compare it), dist(to store temporarily)

In Main:

To find time taken: start, end

Other than these, there are loop variables i, j, k, m etc that are used but the above variables are important in the understanding of the program.

Parallelisation:

The program is parallalised using OpenMP which uses Fork and Join method for the process. It is a good choice for parallelising the program because it has loops with a lot of work in them which the multicore processor can divide in tasks and also the fact that OpenMP can do it in a way that best suits the work that is present to do.

But the problem in this program is first that was mentioned before that the writers of the paper only parallelised the initial placements portion that was actually unnecessary and the other problem being the fact that there are a lot of dependencies because of shared global arrays and there will be race condition for accessing those portion in the cache line.

I have parallelised a portion of the Core Procedure that tries to find the best possible location by going recurrently in a For loop trying to place a sphere in appropriate place. I chose this portion because the computations in this were a lot and it was the heaviest portion of the program as far as time taken is concerned.

But the speed up after using OpenMP as well wasn't much and it isn't an ideal program to actually parallelise. But the computation time was decreased so we can say that even if it isn't ideal, it can be done to run the program faster.

In the paper, they only did it for 50 spheres and I did for 500 in increments of 100 starting from 100 spheres and saw the behaviour of the program change.

Experiments:

I ran the programs on an Intel Core i3 processor with 4GB RAM and a speed of 2.40GHz.

The OS on which it was run was Ubuntu 12.04.

I ran each of the serial program, OpenMP with 4 threads and OpenMP with 8 threads for the following cases:

1. 100 spheres with cuboid size as
    a. 1000,1000,1000
    b. 500, 500, 500,
    c. 200, 200, 200

2. 200 spheres with cuboid size as
    a. 1000, 1000, 1000
    b. 500, 500, 500
    c. 200, 200, 200

3. 300 spheres with cuboid size as
    a. 1000,1000,1000
    b. 500, 500, 500,
    c. 300, 300, 300

4. 400 spheres with cuboid size as
    a. 1000,1000,1000
    b. 500, 500, 500,
    c. 300, 300, 300

5. 500 spheres with cuboid size as
    a. 1000,1000,1000
    b. 500, 500, 500,
    c. 300, 300, 300

I have included a spreadsheet with all the 316 experiments with their details and the times that they took to run along with the averages that are needed to find the behaviours of the programs. Each a. in the 1-5 above was run 10 times and b, c portions were run 5 times each to find the average for each case.

For running the serial file, I compiled it with –lm because it had math.h included in it. The whole of the command used to run the serial command on terminal is as follows-

```
gcc Spheres.c -o a.o –lm

./a.o
```

For running the parallel file, the command for the terminal was as follows-

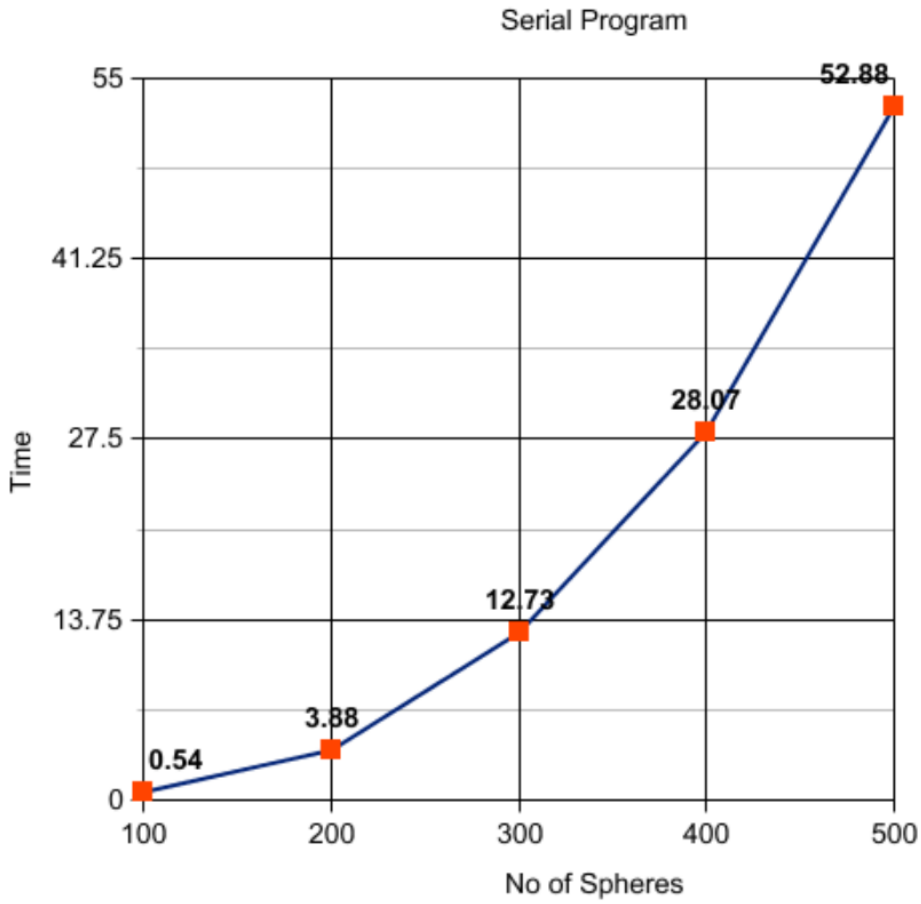
```
gcc SpheresOpenMP.c -o s.o -lm -fopenmp

./s.o
```

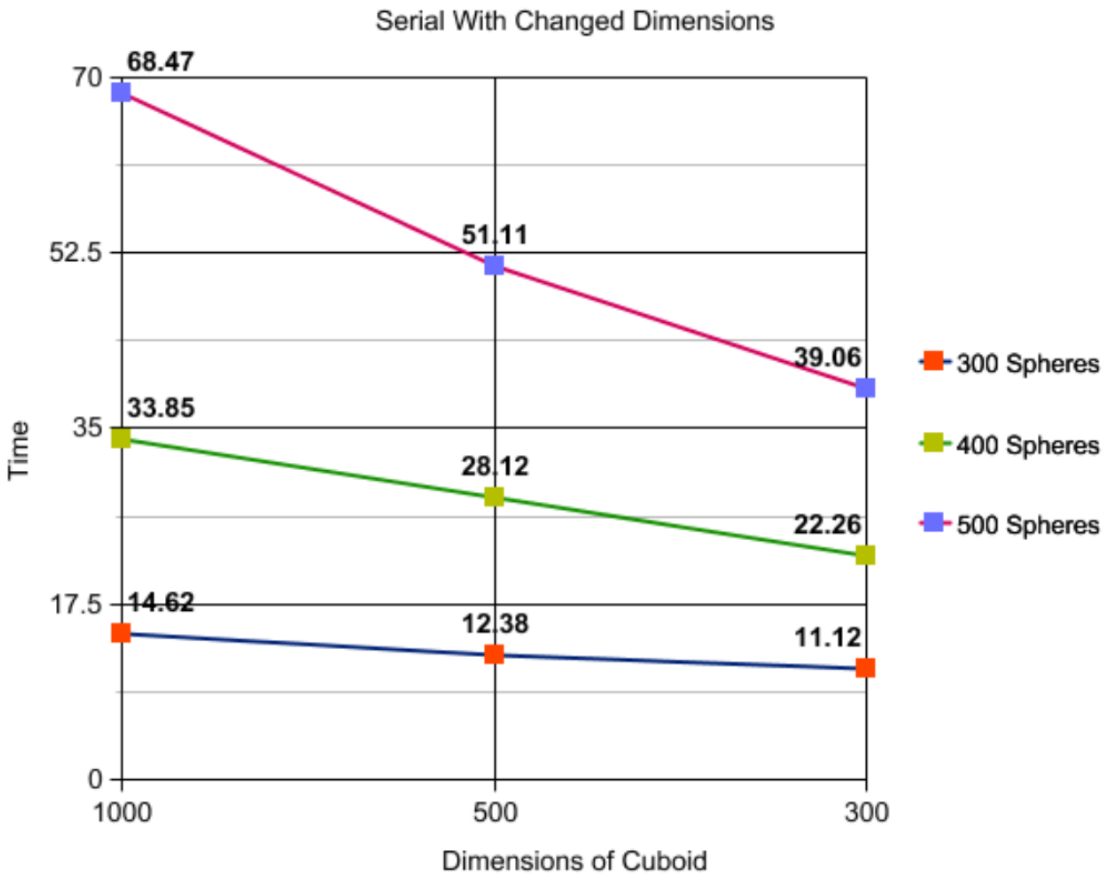
Analyses:

There were a lot of interesting things that I found varying different parameters. The following is the list of the inferences that I reached to along with the graphs showing the results that show what I found out.

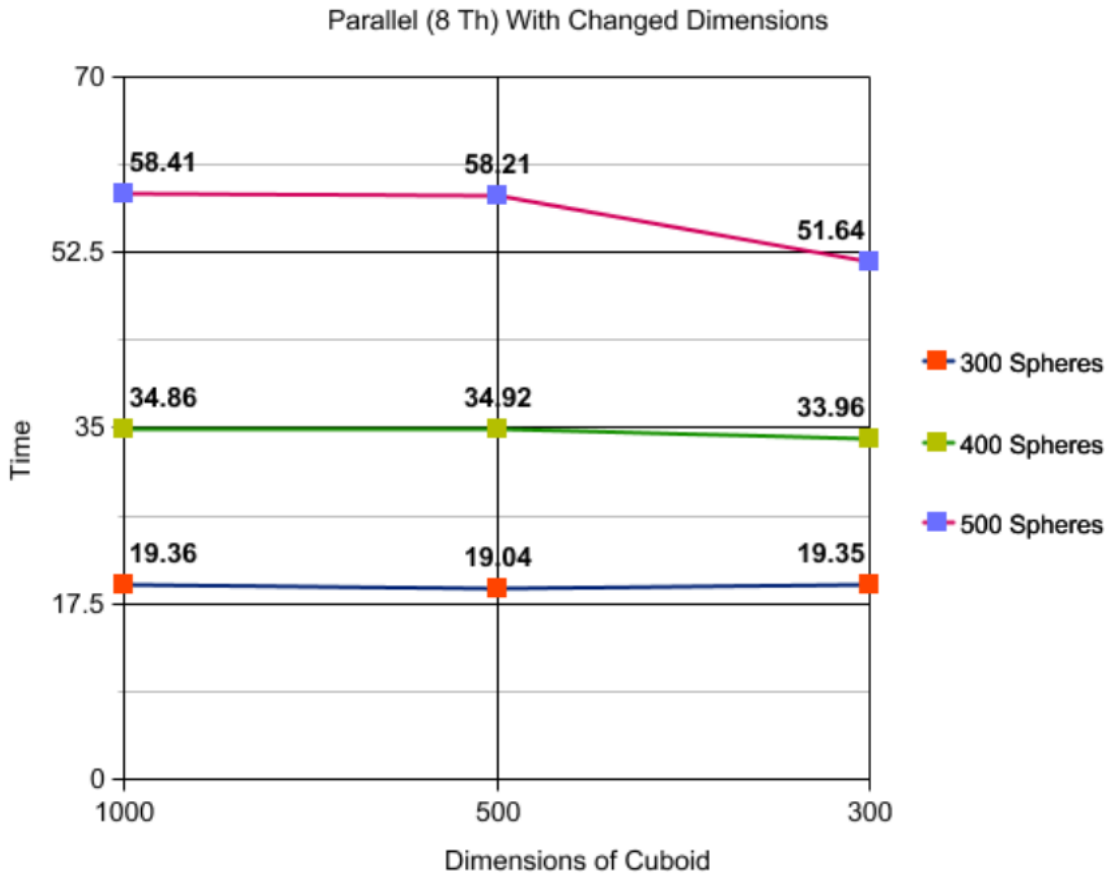
1. Varying the number of spheres for the serial program (here I have taken average of 20 experiments of each with varying but consistent sizes of the cuboid). What we see is that the time taken by the program increases almost in a parabolic way which was expected because the computations increase rapidly with increasing number of spheres.



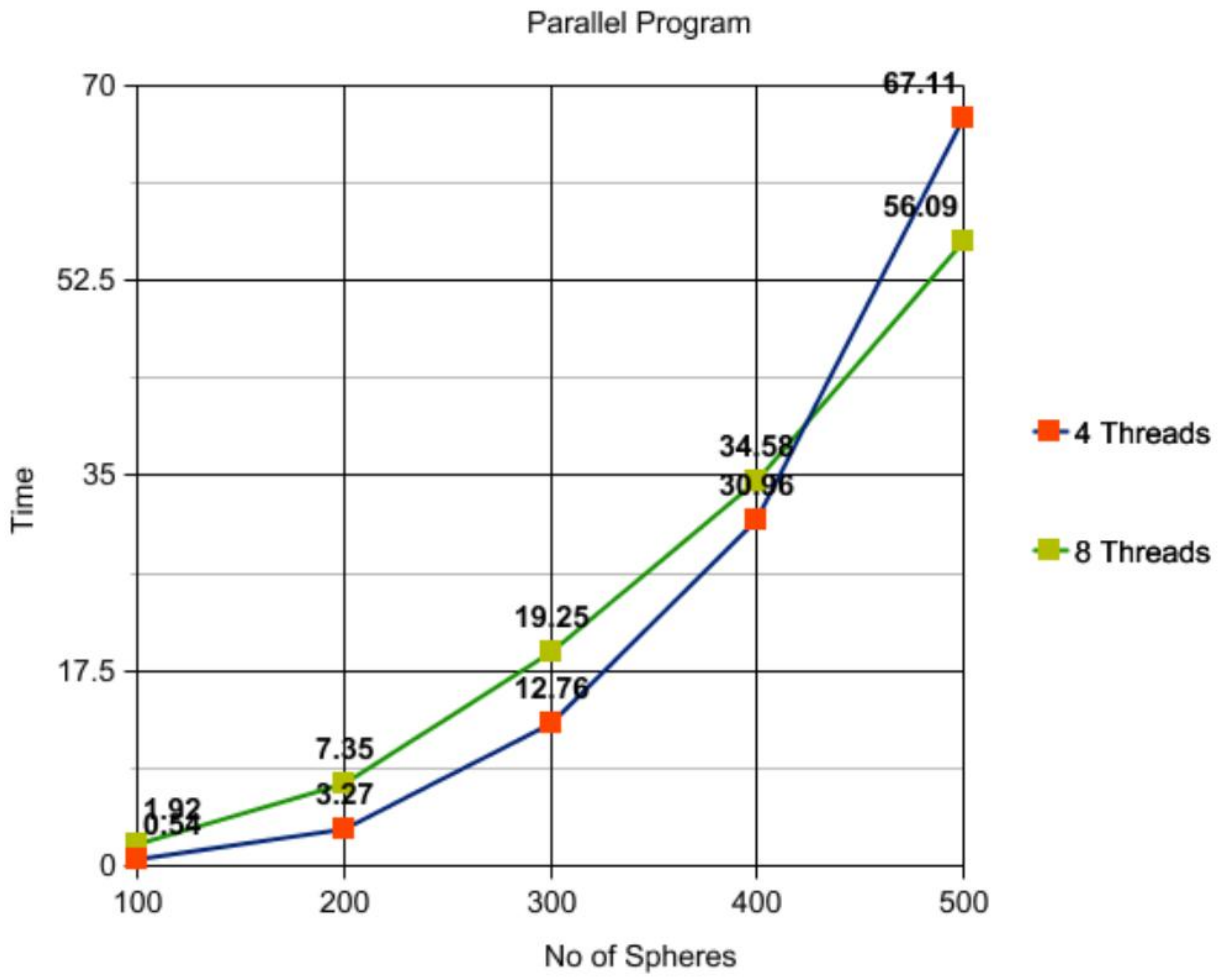
2. Varying the dimensions of the cuboid in the serial program with different number of spheres, what I found was that with more number of spheres, the time to fit the spheres decreases faster than how much it does in case of smaller number of spheres. I think this is because the area to scan becomes smaller and in case of larger number of spheres, it is an advantage.



3. Changing the dimensions in case of parallel program didn't see this kind of decrease in time. It almost remained constant with changing dimensions except in case of 500 spheres which showed improvement with decreased dimensions which was significant from going from 500 to 300 units more than from 1000 to 500.



4. Varying the number of threads for the parallel program as 4 threads and 8 threads with the varying number of spheres (and sizes of cuboids, which was done in same way for both number of threads), I found that initially the 4 threads programs in this environment does better with smaller number of spheres but as soon as the number of spheres start increasing, the performance of 8 thread program increases drastically.



These are all the results that I reached to doing this project.

Conclusion:

This project was a really interesting one and I learnt a lot in the process of completing it exploring the field of parallel computation. This problem is a real world problem that is used in a lot of applications that are actually really crucial and it isn't an easy one. Especially making this problem to run in a faster speed is a difficult problem because it has a lot of computations for each possibility along with keeping track of previously done work.

I found that my serial code takes good bits from a lot of concepts of sphere packing that makes it a good logic but parallel code isn't very efficient and it can be improved upon much more than it currently is. But for that I think a different approach than this paper will be needed because this method of sphere packing has too many dependencies for efficient parallelisation.