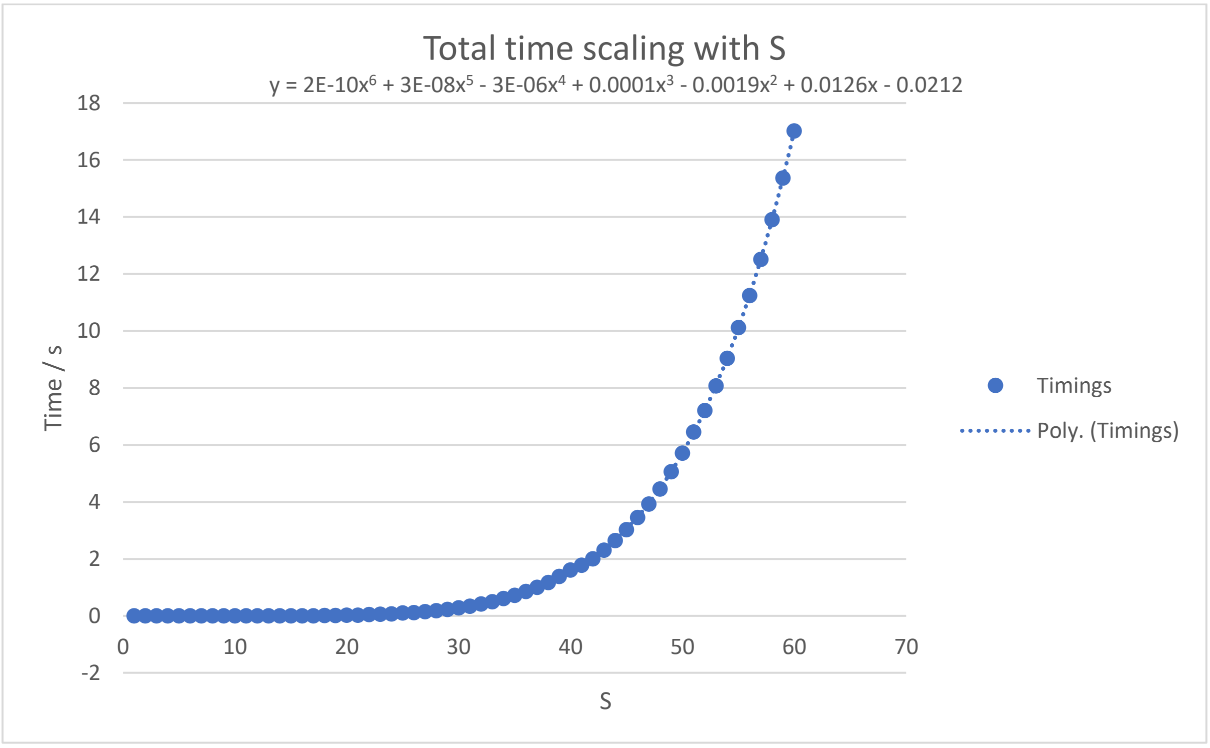
PX425 Assignment 5 Report:

Section 1 – First Steps:

Part 1



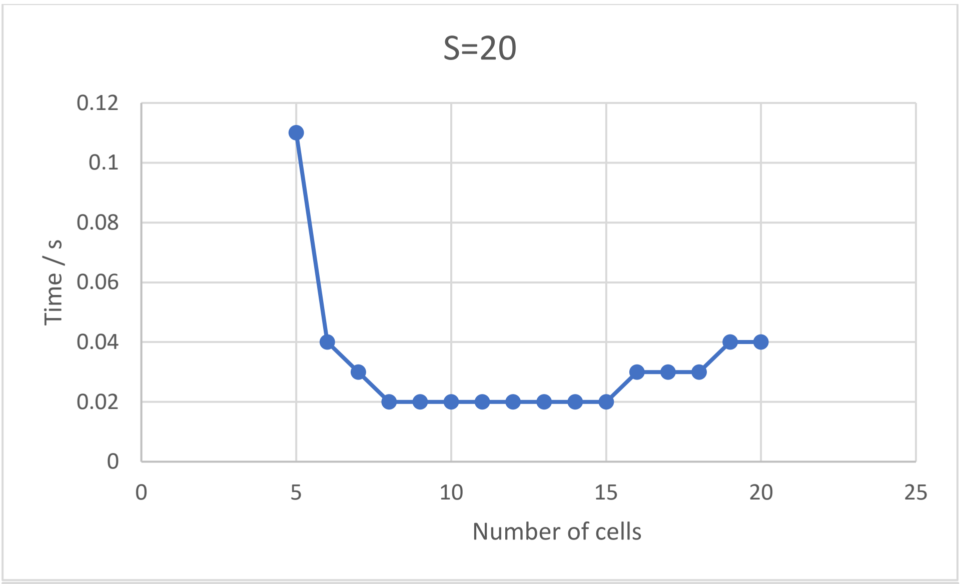
Time scaling with S on the x-axis and time in seconds on the y-axis. The plotted points (dots) are the timings, the dashed line beneath is a polynomial (order 6) fitted to the data, the equation for which is given above. This clearly shows time scales with S^6.

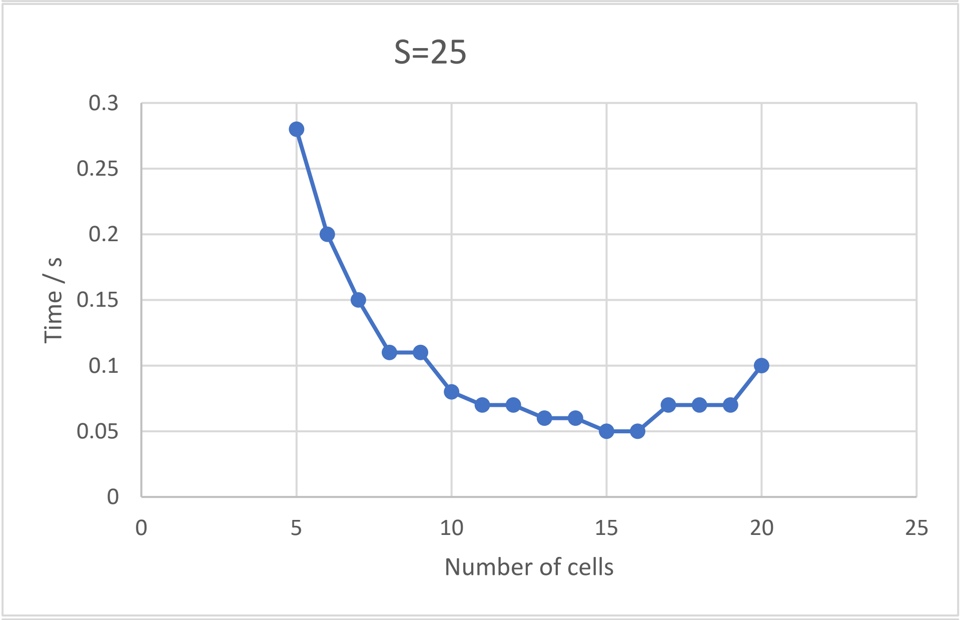
This is a plot of the code compiled without any optimisation. On O3, the timings are much quicker of course. (e.g. at S=40, the mean time, over 5 runs, on O0 is 6.5s, whereas it is 1.6s on O3).

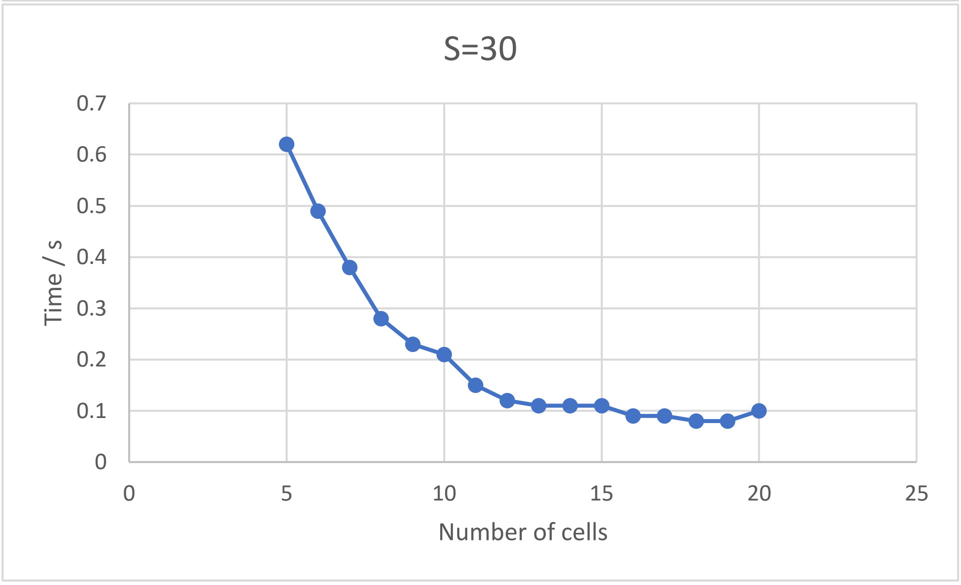
Part 2

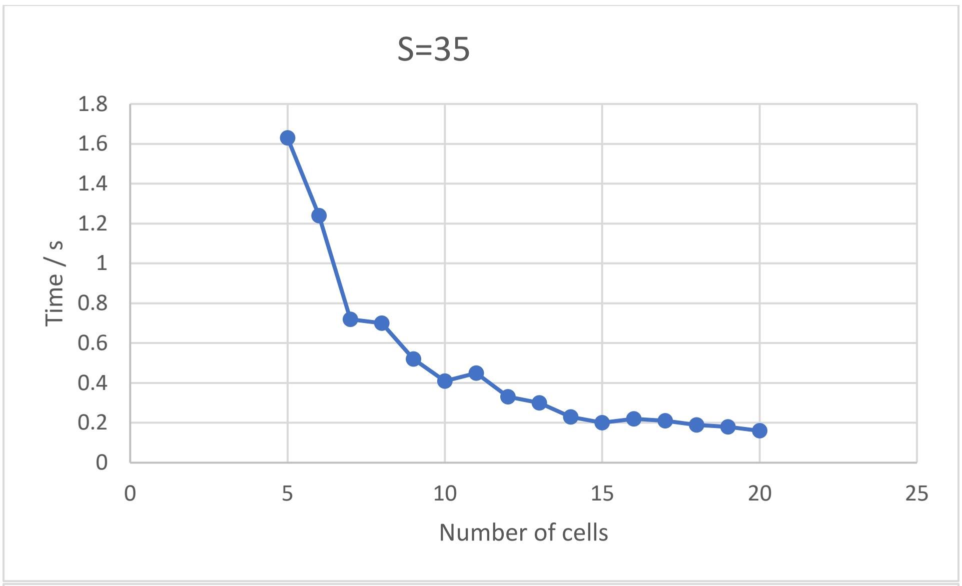
Again, this has been done on O0 for more usable timings.

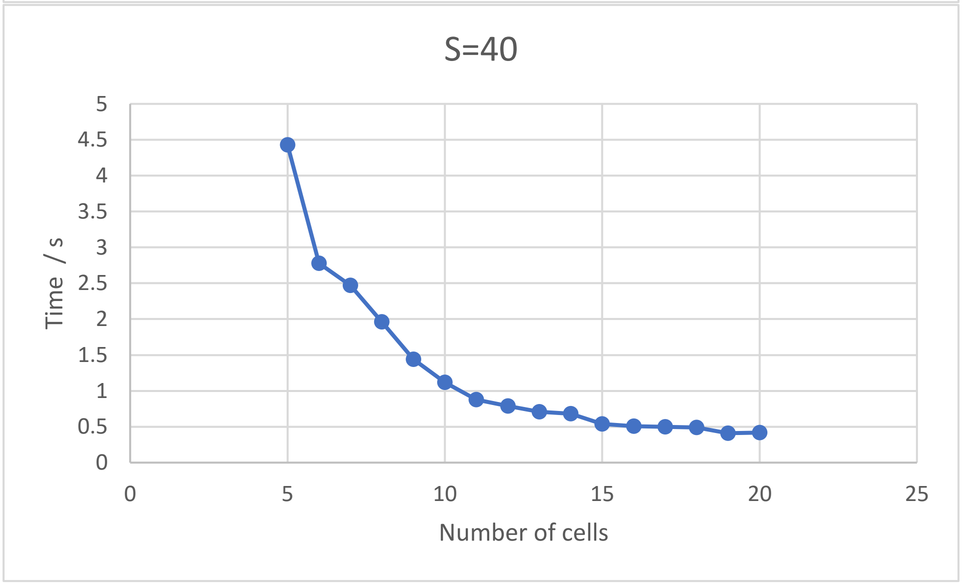
Below are the timings for varying numbers of cells, between 5 and 20, for values of S from 20 to 40.


S=20
Time / s
Number of cells
0.12
0.1
0.08
0.06
0.04
0.02
0
0
5
10
15
20
25


S=25
Time / s
Number of cells
0.3
0.25
0.2
0.15
0.1
0.05
0
0
5
10
15
20
25


S=30
Time / s
Number of cells
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25





T1 decreases as ncells increases, increases as S increases.

T2 decreases as ncells increases, increases as S increases more than T1 does.

T3 constant with variation in ncells, but increases with S.

For S=20, the fastest is at ncells=12 to 15, whereas for S=40, it is fastest at ncells=19 to 20.

T2 increases again after 15 for S=20, but decreases continuously for S=40, T1 is also larger in S=40 so T1’s decrease affects the timing more for S=40.

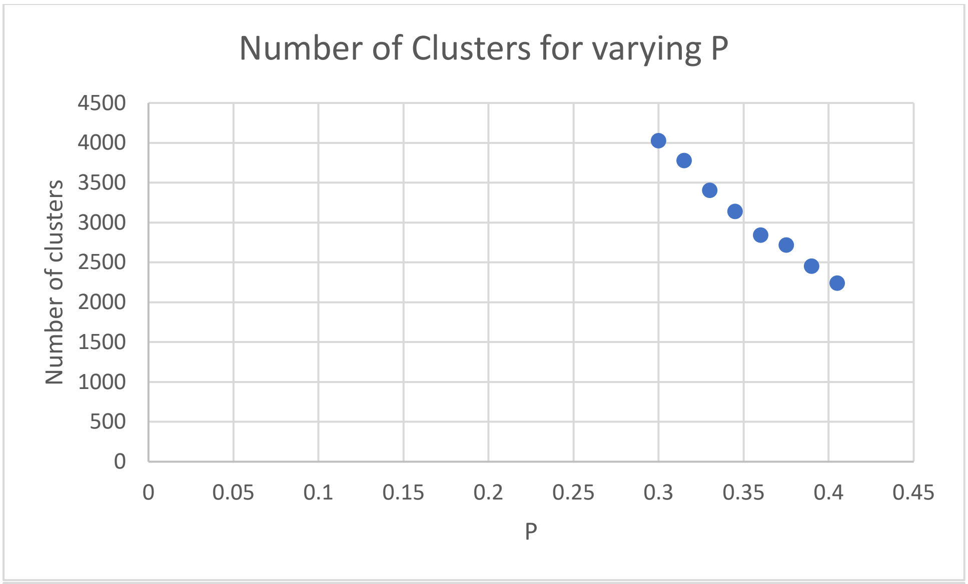
The optimal cell number seems to be around S/2.

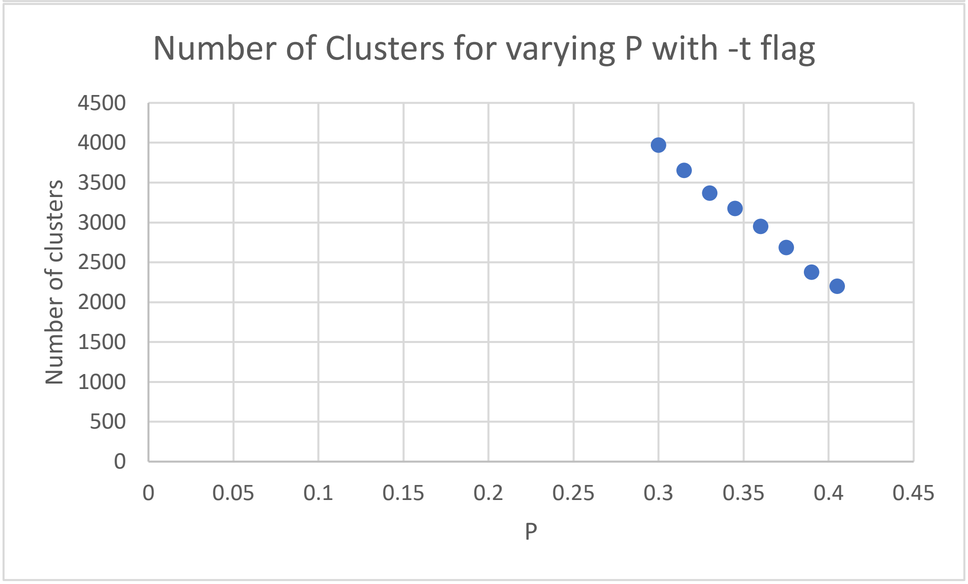
Part 3

As P increases, the number of clusters decreases.

Starts becoming likely to span at about P=0.375 for S=40, with and without the -t flag.

Below are plots of the number of clusters with different values of P, with and without the -t flag.





Parallelism strategy:

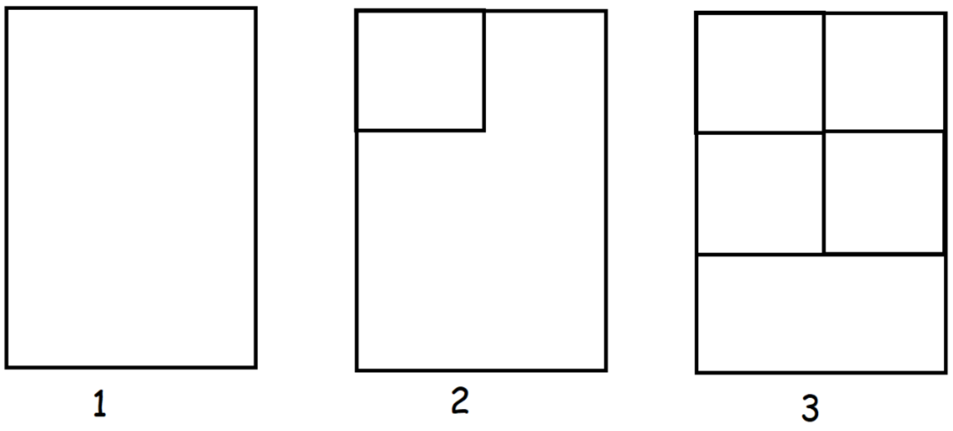
In serial, the code starts at [0,0,0] in the cube and then checks its neighbours, then proceeds towards the point [n,n,n] where n is the number of cells.

There are a few ways to parallelise this. One method would be to fork threads on each [x,x,x] and give different threads different neighbours to check and then join back into one thread. This could also be done with message passing, which would be quicker in some circumstances.

Alternatively, if we visualise the 3D array, it is essentially a cube of cells with length, height, width = n. We can split this cube up into smaller sections and give different threads different sections to work on. Once each section is completed, the boundaries between them will need to be considered, but since they are just 2D planes, they are not a significant time cost. Each boundary can be done by individual threads.

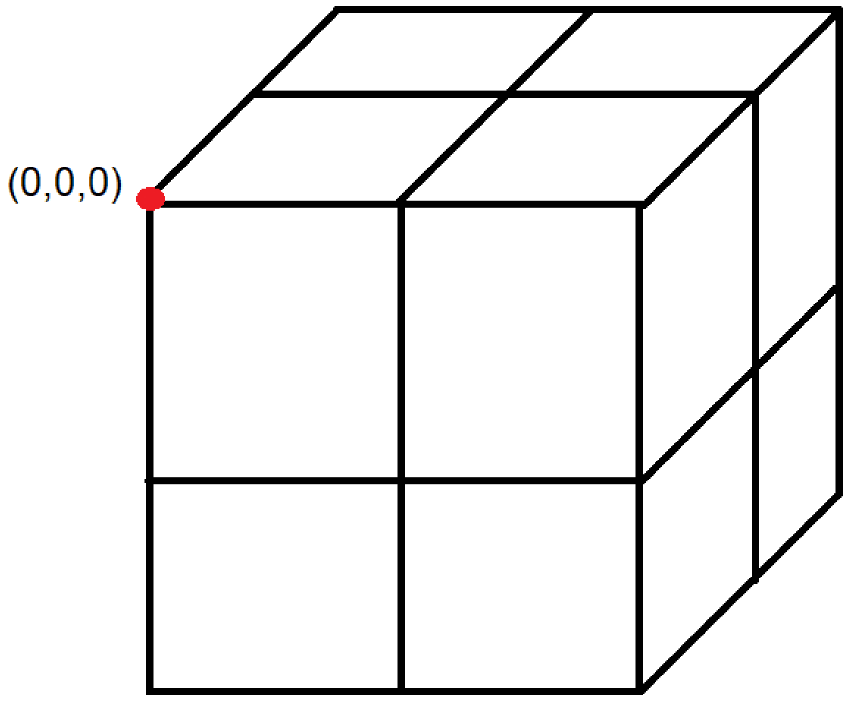
There is a problem with this, however. This occurs if we split a cube into nt pieces (nt = number of threads), and nt is not a cube number. To go through an entire cube without repeatedly checking the same cells and to not have to constantly check if we're trying to access the outside of the cube, there are only 2 starting points that can use the method implemented in the code, radiating out from the middle or from one of the corners. (NB, the middle can only be used if n is odd so from the corner is ideal).

If, however, we have a cuboid rather than a cube, the boundaries will not all be met at the same time, and so a check and stop must be implemented for each of the two shorter directions whilst the other continues.



The figure above shows a 2D representation of this. The boundary to the right is reached before the boundary at the bottom, meaning that there will be a significant amount of unnecessary calculation or expensive checks will have to be made.

To counteract this, when splitting up the cube, it must be split only into smaller cubes. A cube can be split up into smaller even cubes a cube number amount of times (1,8,27…).



For openMP, we only have access to 28 threads on orac, meaning that the best case scenario should be to run the code on 27 threads and split the overall cube into 27 smaller sub-cubes (3x3 cubes). For running on 1 thread, no split will be made, for between 2-8 threads, 1 split will be made (into 8 cubes. For between 9 and 26 threads, 2 splits will be made, the original cube will be split into 8 cubes, then each of those 8 will be split into a further 8 (64 in total). For 27 threads, 1 split will take place into 27 sub-cubes. The reason for the extra splits is to make sure each thread has work to do and to minimise load imbalance, although with this implementation, it is sometimes unavoidable.

Splitting into 8 cubes requires an even n, and into 64 requires n to be a multiple of 4, splitting into 27 cubes requires n to be divisible by 3. Fortunately, n is not up to the user, so this won't be an issue as it can be set depending on the value of S.

There is an issue with this method, however, which is that before joining the cubes, there will be a larger number of clusters found (possibly up to 8 times as many). This will mean that find_spanning_cluster() will take a lot longer as there are more clusters to search through.

Since there are more clusters, it is important to change the way this works. For within each sub-cube, the original method is fine, but for the overall cube, we should check if a spanning cluster is touching the boundaries of each sub-cube and only check if those span with clusters from other sub-cubes.

Once the subcubes are joined, the number of clusters is reduced to (nearly) the same amount as when run in serial. Some clusters are repeated in the count and so are checked through twice, but the time taken for this is not a large proportion. Therefore, there is not an issue with having a potential time loss due to extra clusters.

To actually merge the subcubes, the boundaries of each cube are all that needs to be considered, taking all the neighbours on one side of each cell in a boundary. So for the boundary between two cubes, say cube a and b, there is a maximum of 9 cells to check for each cell on a boundary, which are all 9 of the connected cells in the direction of boundary b from boundary a. If there are 8 cubes, a, b, c, d, e, f, g and h, each touching face must be checked with each other touching face, the corners of diagonally connected cubes will be checked as all 9 adjacent cells are being checked which would include the corners of diagonally connecting subcubes. If a and b are touching faces, a needs to be checked in the b direction, but b doesn't need to be checked in the a direction as this has already been calculated, this means instead of checking 24 faces connecting, only 12 need to be checked.

This merging cannot be parallelised.

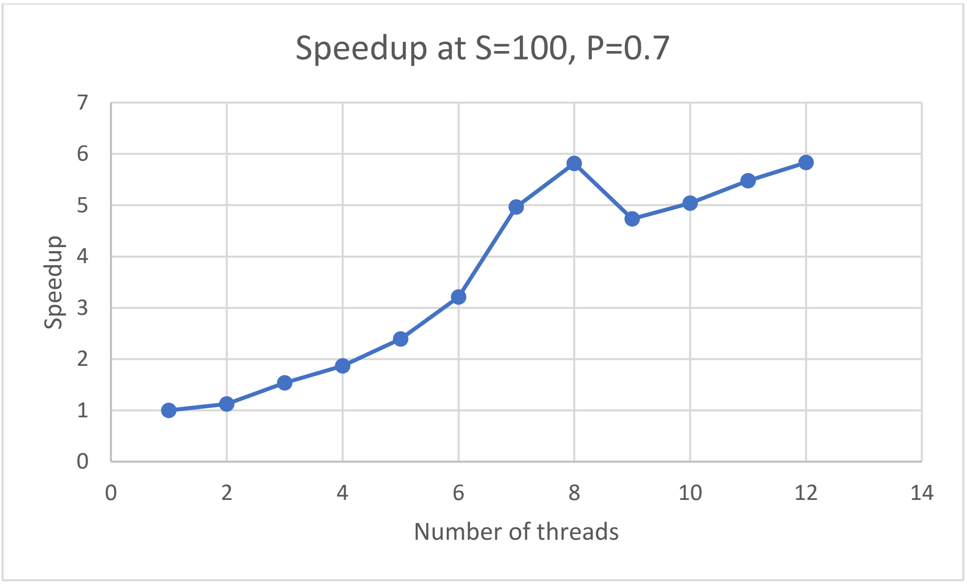
This is not as effective for lower values of S, especially those below 16.

Final Timings:

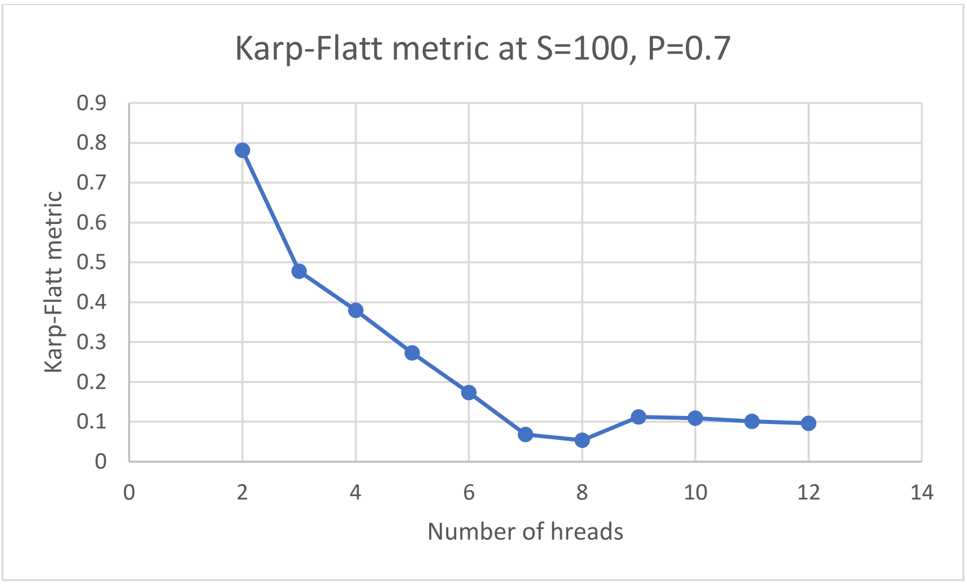
For some reason, orac would not run on more than 8 threads, so these timings are done on my own machine with 12 threads available.

In the parallel implementation, P=0.7 is about the threshold for it to span at S=100.

The speedup: $\psi\ (\mathrm{N}) = \frac{\mathrm{T(1)}}{\mathrm{T(t)}}$, where t is the number of threads, at P=0.7 and S=100:



The Karp-Flatt metric: $e = \frac{\frac{1}{\psi} - \frac{1}{t}}{1 - \frac{1}{t}}$, where t is the number of threads, at P=0.7 and s=100:



There seems to be substantial speed up when there are 8 threads, which is likely because the splitting up and then merging of subcubes is not very well optimised. When there are less than or equal to 8 threads available, the cube is split into 8 smaller sub cubes, which means that the closer to 8 threads, the faster the work is done. When going above 8 threads, the generation and merging of subcubes again is more detrimental to the time than the added parallelism is beneficial to it. Having a computer with 64 threads would be ideal for this situation but, unfortunately, I do not have access to such a machine.

Perhaps an alternative implementation of splitting the cube into a number of subcuboids equal to the number of available threads would have been a better option, to deal with the issue mentioned before about having to check for boundaries constantly with cuboids, a flood fill algorithm could be used to go through the remaining cells. Although flood fill is a slower way to go through a grid, especially in 3D, it would mean that each thread would have the same amount of work and would always be working, meaning for higher thread counts not equal to a power of 8, it would be a much more efficient method overall.