Approach

I structured the distributed version of the program around the feedback and learnings from the shared counterpart. Particularly, avoiding the large copying and moving around of large matrices, using a more robust goal state checking mechanism and removing the possibilities of potential bugs.

For allocating workloads for each processor, I generate an array of problems from the matrix, this takes every single point that will be calculated (the values in the matrix that aren't part of the border) and generates values relative to the point, in groups of five, using this order: left value, top value, right value, bottom value, centre value. This way we have all the values we need calculate the average for a single point as well as the previous value before the calculation so that we can compare how much it has changed. This array of problems, denoted as p_array within the program is split up as evenly as possible between the processers. I say as evenly as possible because we need to account for the fact that it is possible that an array can't be divided equally by the number of processors. I use MPI_Scatterv for the splitting up of the problem array, it provides additional options compared to normal scatter that allows the programmer to specify how the array is split in between all the processors. This program initially calculates all these arguments before it starts processing. I initially calculate the proportion that each process will at least have by calculating the area of the matrix that will be processed and dividing it by the number of processers detected by MPI. Then I calculate the remainder if there is one. Next, for each processor, I populate the set of arrays that are arguments using the proportion and remainder values. Once I allocate the initial proportions, I then add an additional set of values for each processor until I run out of remainder values. I also use this chance to calculate the argument arrays for MPI_Gatherv as it works in an opposite way to scatter. The main processing loop consists of scattering the problem array. Then placing all the calculated values into another array. These arrays are then gathered back to the root, where the matrix is populated with the updated values and then recreated into a problem array. I also use MPI_Allgather to communicate to each processor the status of each processor. Once each processor understands that all the other processors are done with their processing, they can all exit the loop. The status is determined by the goal state checking system where if all values the process calculates meets the goal of being less than the precision, then we can say that processing for that particular processor has finished. In coursework one, I used a count which had to be locked when it was added to, which was unnecessary, this time I just used a simple Boolean check that will tell the processor that it is not done yet while it is calculating all the points it has been assigned.

While I am technically increasing the amount of data being sent to each processor compared to just simply dividing up the matrix equally and then sending it to each processor, I wanted account for the messaging overhead. By sending each processor exactly what it needs to calculate as well as the relevant information it needs straightaway, the processors themselves don't need to communicate with the others to find the information it would need, to calculate the edge values. For example if process 1 and 2 both get 10 points each to calculate, process 1 would require some values from process 2 to calculate some of its points, therefore they would need to communicate with each other. With my approach process 1 and 2 get information along with their 10 points to help with the averaging and comparing.

Correctness Testing

I specifically developed some helper functions that could aid in the development of the coursework as well as check that things were correctly happening at certain points of the program. The function print_matrix() was used to check how the matrix was changed after all the points were gathered.

This was used in conjunction with sleep(), to allow for extra time to clearly see the changes the program was doing. Using this method, I could easily identify when points were not being calculated and could adjust the program to fix these issues. I also used this function to compare the results of the sequential version and the parallel version to see if the matrices match.

Sequential:

```
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.920561 0.851690 0.773153 0.666520 0.476404 0.000000
1.000000 0.851690 0.723357 0.595025 0.449005 0.264788 0.000000
1.000000 0.773153 0.595025 0.444896 0.304824 0.165599 0.000000
1.000000 0.666520 0.449005 0.304824 0.198805 0.092785 0.000000
1.000000 0.476404 0.264788 0.165599 0.092785 0.046393 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```

Parallel (3 processors):

```
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.922514 0.849373 0.774581 0.675889 0.485400 0.000000
1.000000 0.849373 0.713326 0.590094 0.456201 0.274054 0.000000
1.000000 0.774581 0.590094 0.441601 0.309793 0.167020 0.000000
1.000000 0.675889 0.456201 0.309793 0.199075 0.100570 0.000000
1.000000 0.485400 0.274054 0.167020 0.100570 0.048287 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```

As you can see that they are **not** an exact match. I used the same dimension and precision for both conditions. I also tested these conditions on my CW1 and it does match, therefore it is possible that my CW2 version is slightly off or alternatively both my sequential and CW1 version is not exact. I decided to proceed anyway due to time constraints and the fact that the values are not that different. Ultimately, it was producing the same type of matrix. I also tested the program using a differing number of processors, and it was producing the same results each time, which means the program was at least reliable when it comes to repeatability.

The two other functions I used were print_array() and print_int_array() to check the composition of arrays in the program as they are used heavily in the program. I used print_array() for the double arrays and to analyse the states of the p_array and what values were given to each process after MPI_Scatterv(). I used print_int_array to analyse the various arrays that were used as arguments for Scatterv and Gatherv, to make sure that the correct amounts of data was being split between each process.

Once I determined that the program was functioning correctly, I then run some test jobs to see how well the program scaled. I found various issues with memory usage and low performance when using more processors which was perplexing. I found that the program will run out of memory when increasing the processor count and would be fine with a lower count even with the same problem size. I had made sure that I was freeing old arrays and was not copying matrices or anything. The issue would occur instantly, so the program was not eventually running out of memory while it was running. I found that even though you need to call functions like MPI_Scatterv for all processors, certain arguments are only significant at specific processors like root. What I was actually doing was generating the matrix and problem array at each processor, at matrix sizes of around 20000 by 20000 this would mean a problem array size of around 16GB. 16GB * 44 processors is 704GB. Each node has access to 352GB. Which means I was exceeding the available memory per node when using more processors. I found that we only need to generate these arrays for the root process and that the other processors could just be passed null pointers.

Scalability Investigation

I focused my scalability investigation around using the same parameters as my shared memory program as well as extra scenarios with much larger matrices.

Speedup

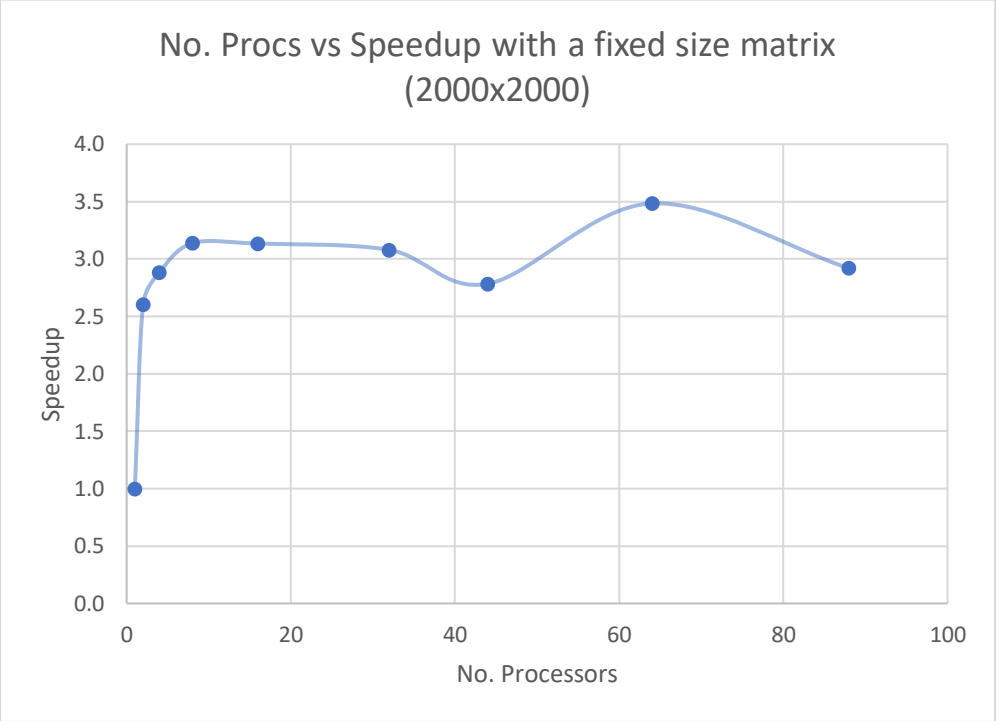

Figure 1- Procs vs Speedup

This graph (fig. 1) is a lot more interesting compared to my shared version. Here we can see a case of superliner speedup when we use two processors. Speedup is defined to be less than the number of processors. However, we have a speedup of 2.5 for 2 processors. This could be because of a fundamental change in the algorithm, making the parallel version too fundamentally different to the sequential version, so they can't really be compared anymore. I would need to revisit the sequential version and think about if it's possible to make it more efficient.

The maximum speedup has been seemingly reached at the 8 core mark and then we see a slowdown at 44 cores, this could possibly be because of a bottleneck in memory access, having more processors causes a higher chance of processors waiting for their turn to access memory. This is backed up by the fact that speedup increases again when using 64 (32x2) cores. Now that the program is split over two nodes, we are less likely to hit a memory bottleneck, since each node has their own share of memory. This also rules out communication overhead, since we have more processors and therefore more communication, but the speedup has still increased. I look into this further by fixing the number of cores at 24 and changing the number of nodes as seen in figure 2. Karp-Flatt also shows an increase in the sequential proportion when we up the processor count, which should be obvious considering how the speedup changes.

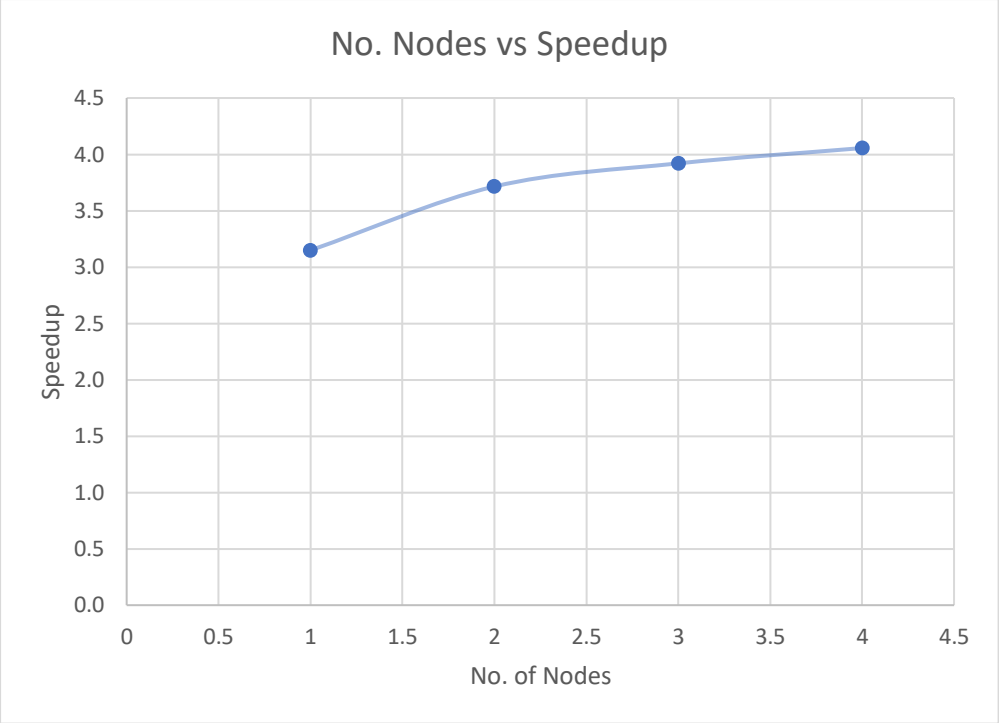

Figure 2- Nodes vs Speedup

Here we can see that speedup increases even though I'm using the same number of cores. The change in speedup does decrease however, as we add more nodes, possibly with communication overhead between nodes. We will need to investigate with more nodes in order to be more conclusive however.

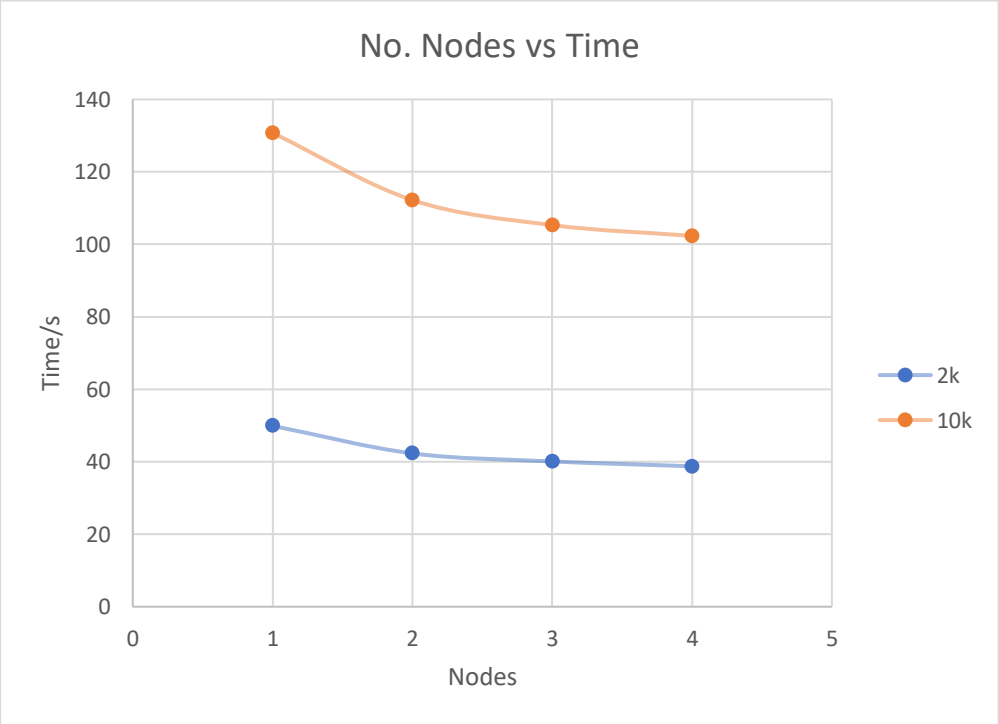

Figure 3-Nodes vs Time

The decrease in completion time is also more pronounced with larger matrices. Larger matrices use more memory, reinforcing the idea of the program having a memory bottleneck.

Efficiency

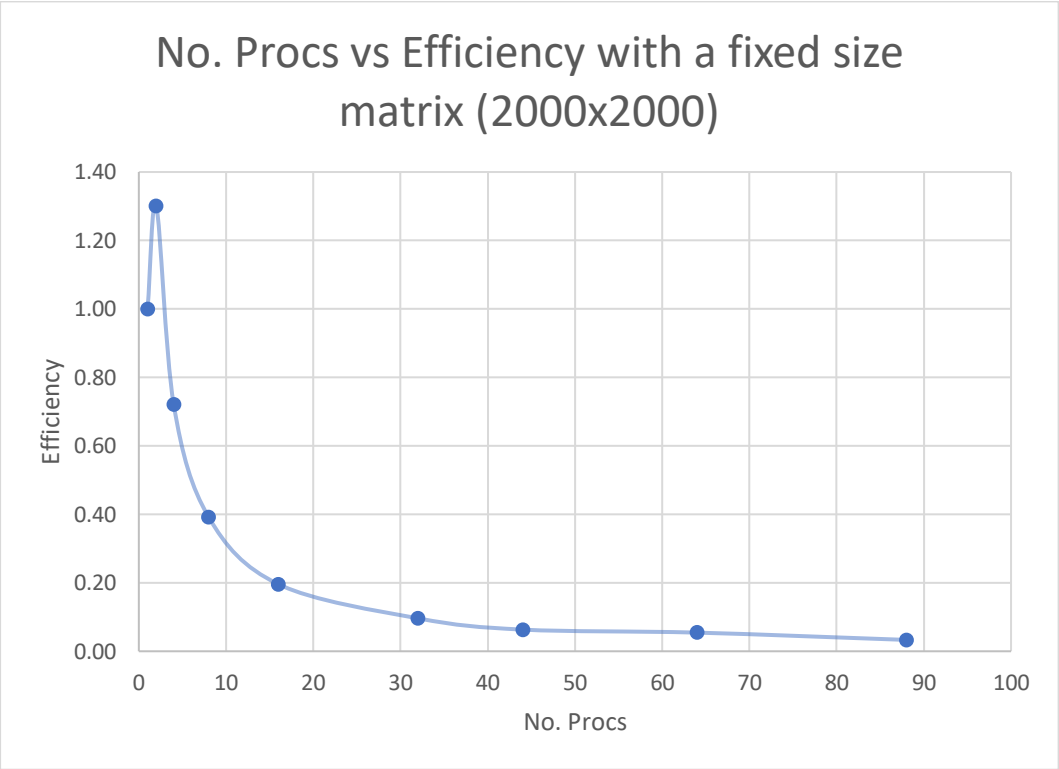

Figure 4-Procs vs Efficiency

Here we can clearly highlight the super linear speedup since our efficiency is over 1, which shouldn't be possible. Efficiency does take a massive hit with this problem size, so we should start thinking about scaling our problem size for this.

Scaling

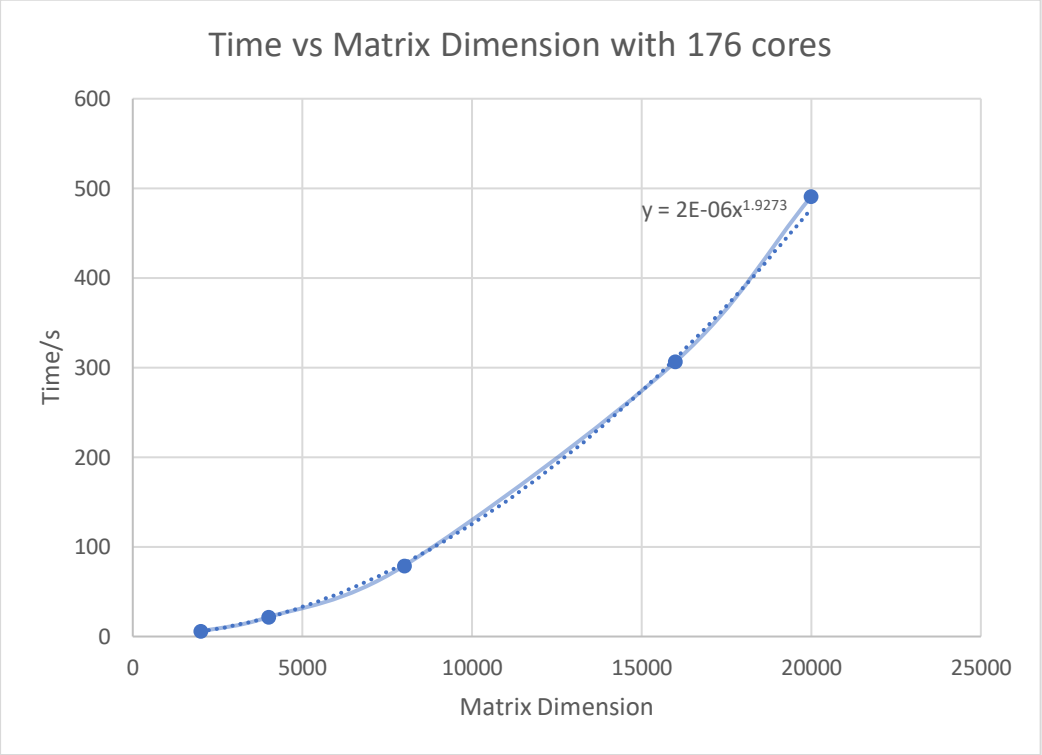

Figure 5 - Time vs Size

Figure 5 shows a similar scaling relationship to my shared version of the program, except I can run problem sizes that are much larger. The program looks like it scales well, we must remember, just like with the shared version, the problem size is not scaling linearly here since the actual size is dependent on the area rather than just one dimension of the matrix. Therefore, we can conclude that the scaling is linear as the trendline equation is close to x^2 and the problem size is close to the matrix dimension squared.

Compile Program and Run

To test the program out yourself. Compile using:

mpicc main.c -Wall -o main.o

You can run the output normally or with optional arguments like this:

mpirun -np 176 ./main.o -d 25000 -p 0.01

-d: Dimension of matrix.

-v: Starting value of matrix.

-p: Precision to work to.

-np: Number of processors.