UNIVERSITY OF
BATH

CM30225: Parallel Programming

Coursework 2: Distributed Memory Programming

MA2451

The University of Bath
Department of Computer Science

5th January 2022

UNIVERSITY OF BATH

Introduction

Distributed memory architecture is an approach to parallel computing that utilizes a number of separated compute nodes, each with its own unique memory space. Problems can be partitioned and dispatched to the nodes in chunks, then reformed - with the intention of sharing the load and speeding up processing. MPI (Message Passing Interface) is the standard implementation and, as the name suggests, passes messages between nodes. This stands as kin to shared memory architecture used in part one of the coursework which consists of a single node with a single memory space to draw from.

The problem involved creating a script in C to perform differential iterative relaxation on a matrix of doubles until the difference between one iteration and the next settled to a precision value. Matrix boundary values remained fixed. The approach to how parallelism should be implemented was entirely dependent on me.

Approach To Parallelism

My approach to shared memory programming was flawed. The overuse of mutexes would not serve a distributed system. So, I opted to partition and scatter the matrix based on the number of processes while keeping track of the amount of cells that have reached precision and reducing this flag to provide a global indication that a solution has been reached, ideally preventing the overuse of message passing between nodes - the most noteworthy bottleneck for a distributed memory program.

Advice from my peers pointed me in the direction of using a flat array to represent the 2D matrix for improved performance, especially with matrix dimensions above 10,000. For instance, a 100x100 matrix would be represented as a 10,000 element array - the first 100 elements representing the top row. This does, however, create the inconvenience of needing offsets for indexing to be scattered with the rows.

Our process for relaxing is as follows:

- Allocate memory for, and fill our relax matrix with its starting values in the parent process.
- MPI_Scatter needs to know the number of elements sent to the child processes from the parent[1], so calculate the number of elements that will be in the buffer sent out to each. Also true for MPI_Gather.
- Scatter the matrix, using it as our send buffer argument, and define a receive buffer whose values will be read, relaxed and written to a separate buffer.
- Each process relaxes the values in the MPI_Scatter receive buffer. Determine if the values are within precision and update a flag for each process to confirm this. Write the relaxed value to a buffer which will serve as our send buffer argument for MPI_Gather.
- Gather the matrix chunks and reduce the precision flag for the child processes, summing them to a flag for the parent. If the flag = 0 all cells are in precision and break. If not, another iteration is required. Scatter the matrix, relax and gather again.

Heeding the warnings from the shared memory feedback, I avoided the use of globals and defined the relax matrix within main(). This is of particular importance in the distributed implementation because the matrix is not global to all processes, especially when multiple compute nodes are being used.

Relaxing any cell involves using the 4 adjacent cells.

1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.000	0.500	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	1.000	1.000	1.000	1.000	1.000	1.000

There is no need to send the relax matrix itself to each process, which would serve only as a detrimental overhead, but values outside the row being relaxed still need to be read - namely those above and below the target cell. In the case where 1 row is to be relaxed by 1 process, this can be easily solved by sending the rows above and below alongside the target row as a single 3 * MatrixSize buffer.

1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.000	0.500	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	1.000	1.000	1.000	1.000	1.000	1.000

In practice, the number of processes should be less than the number of rows. If this isn't the case, the benefits of scattering the problem are being wasted. With this in mind, we can calculate the total allocation (sendcount) for each process as:

```
process_allocation = matrix_size * ((matrix_size - 2) / process_total);
send_counts[process] = process_allocation + read_only_rows;
```

Where process_total is the MPI Comm Size (total number of processes) and read_only_rows constitute 2 * the matrix size to represent the additional above/below rows that need to be sent with the buffer.

However, we need to take into account the instances where the number of rows in the matrix working area is not divisible by the number of processes. In this case, which is henceforth known as an 'odd row' case, some processes will need to handle an extra row. We can neatly add one extra row to a number of processes equivalent to the remainder of the matrix working area over process total.

```
odd_row = (matrix_size - 2) % process_total;
if (process_id < odd_row) {
            send_counts[process_id] = process_allocation + read_only_rows + matrix_size;
}
```

All processes with an ID lower than the result of the modulo operation are assigned an extra row.

Because of the new possible discrepancy in buffer sizes, we must use MPI_Scatterv and MPI_Gatherv which are little more than flexible versions of their fixed size buffer counterparts.

The receive buffers that will be gathered into the parent matrix once a relax iteration has finished do not require these additional rows, so we can define the size of this buffer for a given process as the size of the send buffer minus two rows, and the respective receive-offset for indexing should point to the row after that pointed to by the send-offset.

The method for relaxing a cell was lifted from my shared memory implementation and modified:

```
double sum = (read_from[i - matrix_size] +
              read_from[i + matrix_size] +
              read_from[i - 1] +
              read_from[i + 1]) / 4; // Sum adjacent cells. Divide by 4

double current = read_from[i];       // Store old value from send buffer
write_to[i - matrix_size] = sum;     // Write new value to receive buffer

return (precision < fabs(current - sum));  // Is result in precision? Return true (1) if not
```

Rather than using a universal counter for tracking the precision (which would be impossible without costly message passing), each child process keeps its own precision flag. Should a new value not be within precision, the above function will return true and increment the value of the flag, meaning that a flag value of 0 constitutes a completely relaxed chunk for that process.

```
MPI_Allreduce(&precision_flag_child, &precision_flag_parent,
            1, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD);
```

These flags are then reduced, the sum of all of them stored as a parent flag which we use to determine if all chunks were in precision on this iteration.

Using Azure

The code was compiled in a shell script using mpicc with the recommended warnings (-Wall, -Wextra and -Wconversion) enabled. This file was run from a SLURM script using mpirun. The node and processes-per counts were changed manually with each subsequent test.

The node count in the SLURM script was varied from 1 to 4. The number of processes per node was varied from 4 to 44 in increments of 4, with multiples of 12 omitted due to time constraints.

As mentioned in the shared memory feedback, two-digit N matrices are too small to be meaningful. I started from 100x100 as I was interested to find out how the program would handle a matrix size that is smaller than the number of processes, for example in the case of 4 nodes with 44 processors each (176 processes) 10 matrix dimensions were chosen:
100x100, *250x250*, *500x500*, *750x750*, *1000x1000*, *2500x2500*, *5000x5000*, *7500x7500*, *10000x10000* and *25000x25000* - all tested with a fixed precision value of 0.01 for the sake of comparison.

UNIVERSITY OF BATH

Correctness Testing

I decided to re-use the results of the sequential 10x10 test as a baseline, compared against the distributed architecture program running on 1 node with 4 cores. Feedback suggested this matrix was too small to give a convincing assessment, so I generated two other test cases:

100x100 using 2 nodes, 4 cores each
1000x1000 using 4 nodes, 4 cores each

The printed matrices from the slurm output scripts were put up against the same sized matrices printed by the sequential program. The file size for the 1000x1000 tests makes them impossible to include, though the 100x100 tests for both the sequential and distributed implementations are included in my submission zip.

// Sequential Result in C

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.988164	0.979039	0.973383	0.971474	0.973125	0.977746	0.984456	0.992222	1.000000
1.000000	0.979039	0.962895	0.952900	0.949541	0.952476	0.960658	0.972526	0.986254	1.000000
1.000000	0.973383	0.952900	0.940236	0.935997	0.939741	0.950129	0.965182	0.982582	1.000000
1.000000	0.971474	0.949541	0.935997	0.931482	0.935510	0.946642	0.962755	0.981371	1.000000
1.000000	0.973125	0.952476	0.939741	0.935510	0.939319	0.949805	0.964969	0.982481	1.000000
1.000000	0.977746	0.960658	0.950129	0.946642	0.949805	0.958488	0.971033	0.985515	1.000000
1.000000	0.984456	0.972526	0.965182	0.962755	0.964969	0.971033	0.979789	0.989894	1.000000
1.000000	0.992222	0.986254	0.982582	0.981371	0.982481	0.985515	0.989894	0.994947	1.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

The inverse of the issue that occurred with my shared memory program was present here. Rather than the values being larger than expected, these were smaller - significantly so in the center of the matrix:

// Distributed Result - 1 Node, 4 Cores Per Node

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.981399	0.965041	0.952901	0.946442	0.946442	0.952901	0.965041	0.981399	1.000000
1.000000	0.965041	0.934300	0.911484	0.899345	0.899345	0.911484	0.934300	0.965041	1.000000
1.000000	0.952901	0.911484	0.880746	0.864391	0.864391	0.880746	0.911484	0.952901	1.000000
1.000000	0.946442	0.899345	0.864391	0.845793	0.845793	0.864391	0.899345	0.946442	1.000000
1.000000	0.946442	0.899345	0.864391	0.845793	0.845793	0.864391	0.899345	0.946442	1.000000
1.000000	0.952901	0.911484	0.880746	0.864391	0.864391	0.880746	0.911484	0.952901	1.000000
1.000000	0.965041	0.934300	0.911484	0.899345	0.899345	0.911484	0.934300	0.965041	1.000000
1.000000	0.981399	0.965041	0.952901	0.946442	0.946442	0.952901	0.965041	0.981399	1.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

The same is true for both the 100x100 and 1000x1000 cases. The general pattern of numbers matches the control case, with patches of zeroes being in the right places and areas of columns with many repeating numbers matching up, but the exact values differ. Mutexes are not to blame in this case, so I believe the problem lies with my parallelism approach.

One problem I had considered, but not accounted for, was the idea that all of the processes are not in lockstep, as in not executing one-by-one or even simultaneously in any predictable order. As well as this, since the matrix is scattered with fixed read-only values from the rows above and below, it is possible that the above and below values are not up-to-date.

1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.000	0.500	0.375	0.343	0.336	0.334	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	0.250	0.063	0.160	0.040	0.010	1.000
1.000	0.000	0.000	0.000	0.000	0.000	1.000
1.000	1.000	1.000	1.000	1.000	1.000	1.000

If we consider a hypothetical scenario in which we are relaxing the 7x7 matrix above using my approach and 5 processes (one for each working row), it is possible that the row marked in blue will be relaxed first. If we were to do this sequentially, the values in the rows above would be non-zero values by the time this blue row is relaxed, meaning that the calculation for the value nearest the left (1 + 0 + 0 + 0 / 4 = 0.25) should actually differ very slightly.

The operative word here is slightly, as the values in massive matrices will usually differ from what is expected by less than the precision of 0.01 per iteration, though this does eventually add up. As of writing this report, I cannot think of a way to deal with this issue without entirely rethinking my approach or holding back processes until they run in a controllable order which defeats the purpose of parallelizing the problem in the first place and will slow the runtime down.

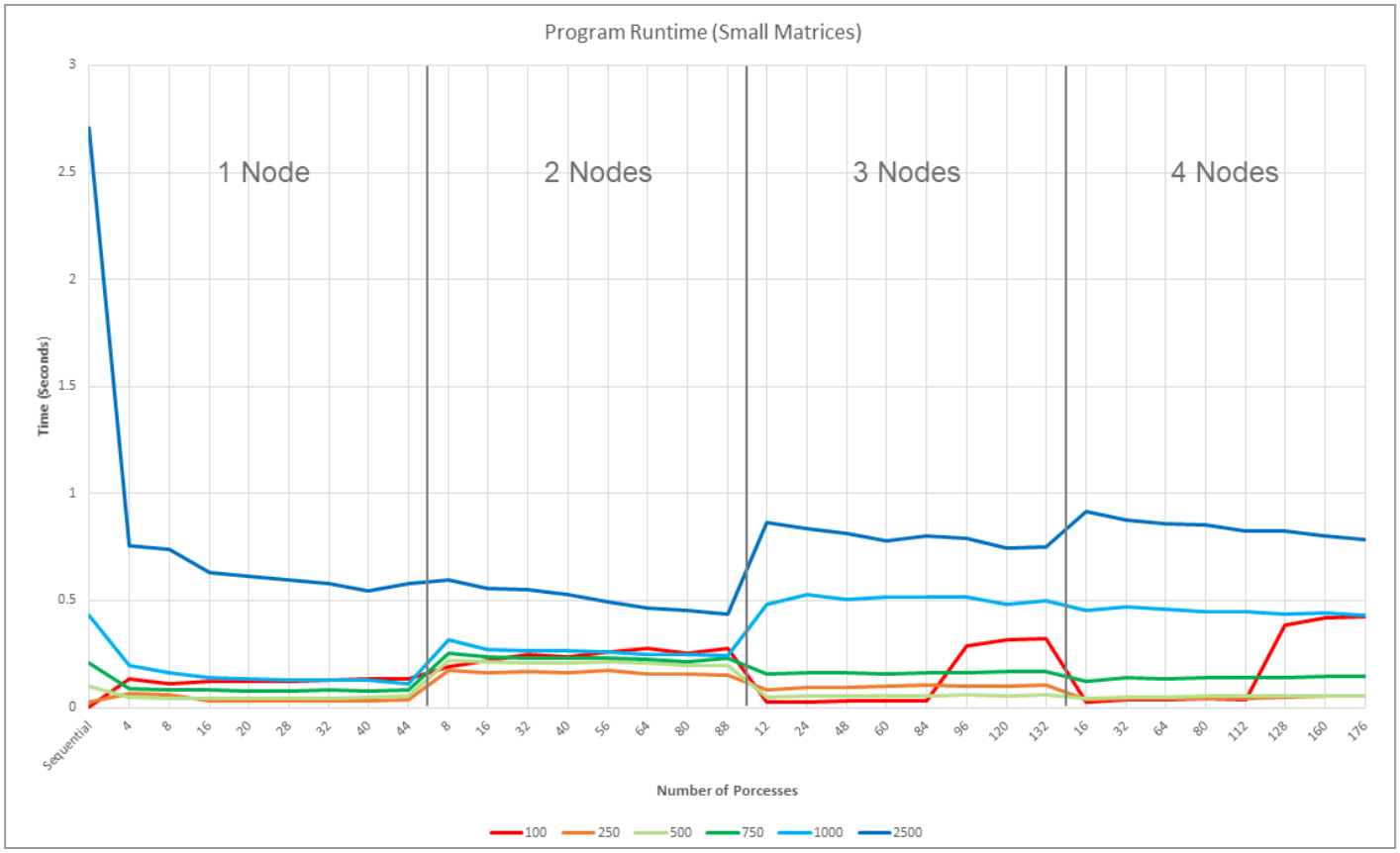
With this noted, the number of iterations to achieve relaxation was the same for all cases sequentially and in the distributed program at 37 full passes (37 reductions of child precision flags), so I decided to carry out the scalability investigation.

Scalability

Typical methods for timing the program did not work. Both the <time.h> and <sys/time.h> methods I had previously used caused errors. I am unsure if this is a problem with the mpicc compiler or an issue on my part. Regardless, using Microsoft's documentation for MPI[2] I found the MPI_Wtime() function which gives us the time in seconds since some arbitrary time in the past. The time does not need to sync with any child processes, so I left the calculation of timing within the parent process.

```
double starttime, endtime;
starttime = MPI_Wtime();
// MPI Program
endtime = MPI_Wtime();
if (process_rank == 0) {
    printf("Runtime = %f\n", endtime - starttime);
}
```

A consistent bug I encountered hampered my testing process somewhat. An error stating “text file busy”, supposedly a linux error caused by running an executable that is already being used, became prevalent when testing large numbers of cores on multiple nodes. I am not certain of the cause, possibly how I am allocating processes, but several high core count timings are 1-run only rather than the average of 3 runs.

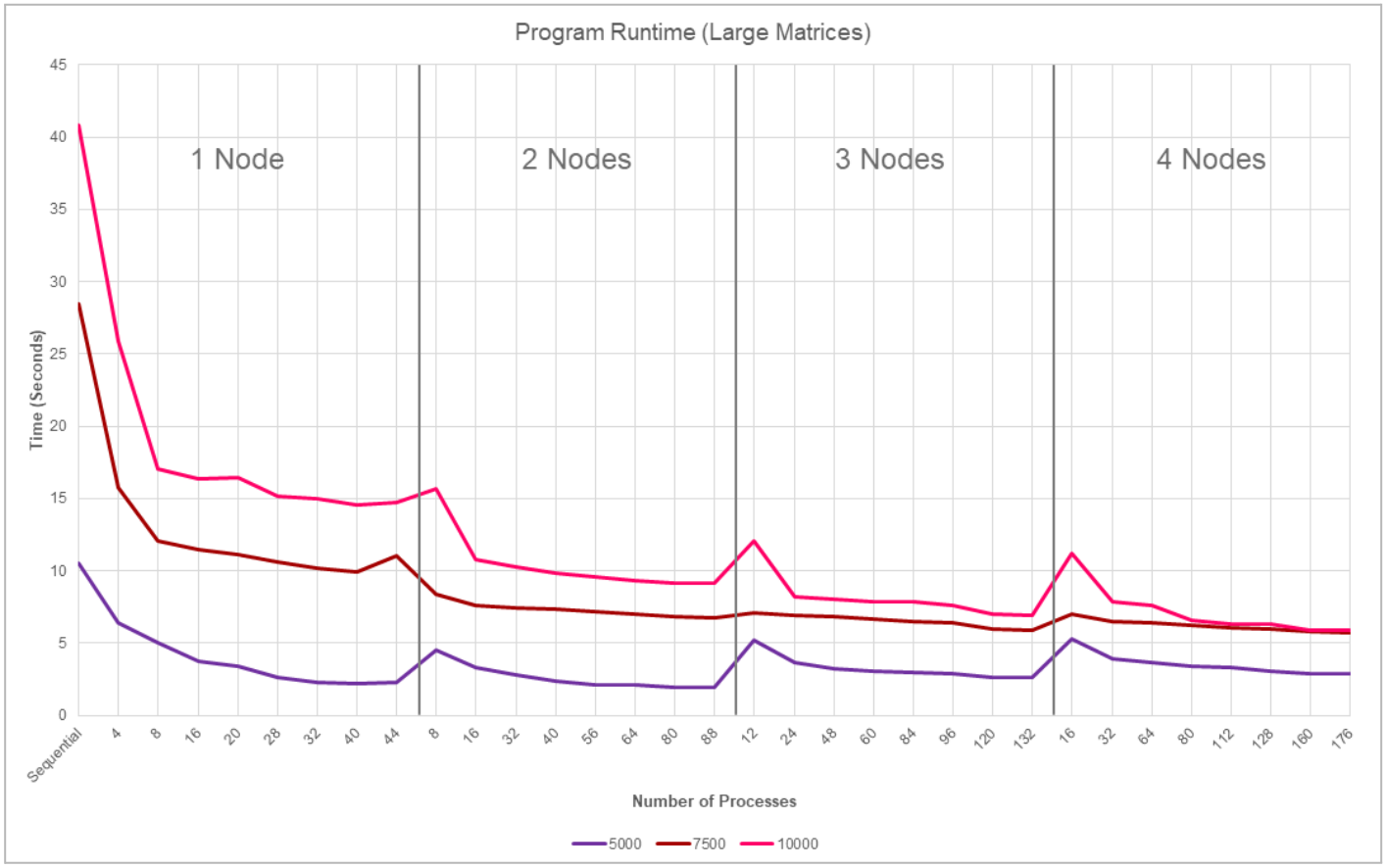


Most small matrix sizes, 100 to 1000, follow a similar pattern to that of the shared memory implementation when using one compute node. The runtime decreases quickly until around 16 processors per node are in use then begins to plateau. The 100x100 matrix sees only an increase in time taken.

When another node is added, the same pattern of gradual decay as more processes are added is still present but the runtimes see a sharp jump upward. I attribute this to the MPI message passing overhead coming into full effect now that there are separated memory spaces, which is only compounded by the small matrix sizes. From 3 nodes onward, behavior varies dramatically. 250x250, 500x500 and 750x750 follow the pattern I would expect, with the new nodes improving the runtime with diminishing returns.

Mentioned in the *‘Using Azure’* section, my curiosity about a greater number of processors than matrix rows bore fruit. When the process count reaches 100, the runtime for 100x100 spikes higher than with any other configuration. Though, interestingly, the spike when using 3 cores occurs at 96 processes (32 per node), which is less than the 98 working rows in the matrix. So, perhaps this phenomenon begins as the ratio of processors to rows approaches 1:1 rather than spiking when it reaches this point, and judging by how the runtime continues to increase after spiking, adding more than the maximum 176 processes may compound the issue further - though this would never be done for any practical reason.

1000x1000 appears to me to be a “sweet spot” where the benefits of adding more cores versus the downsides of MPI overheads are in conflict, as I can see no other reason why the runtime spikes again when a third node is introduced. I would have expected the pattern for the smaller matrices and for this matrix to be swapped, with smaller matrices running slower with more nodes. 2500x2500 is a similar case, though here we clearly see the benefits of parallelism now that the problem size is growing - the runtime falling by almost an order of magnitude.

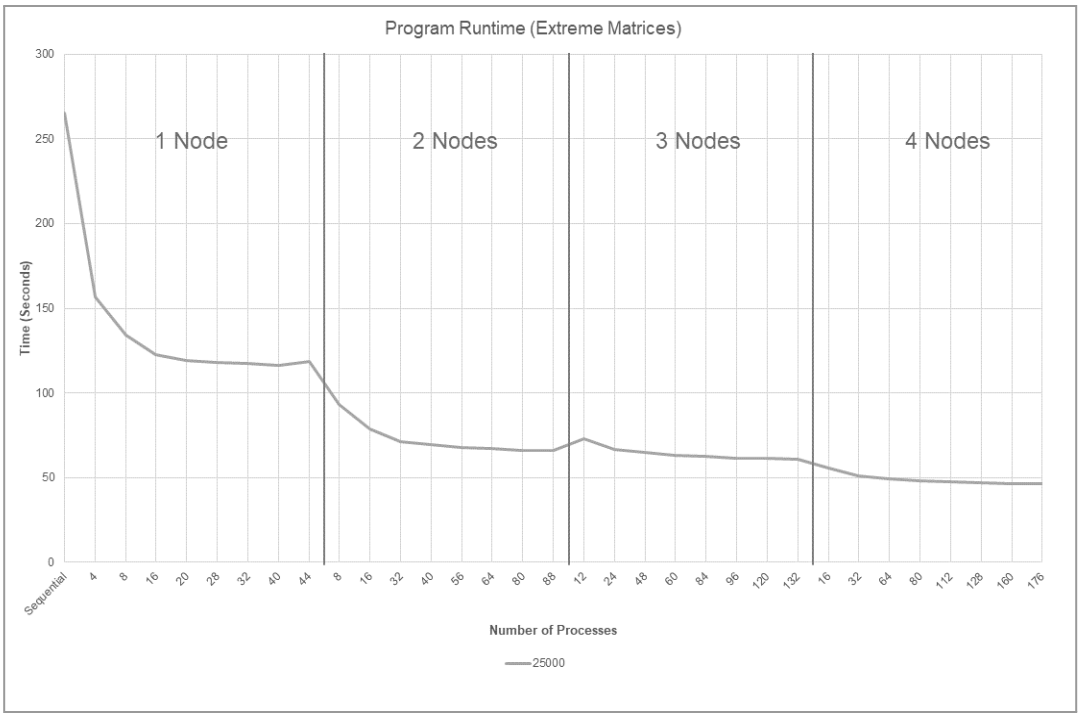


The larger matrices are the real test of the program’s effectiveness. The 5000x5000 matrix exhibits the same “sweet spot” effect as some small matrices where the runtime increases with more nodes. This issue does not appear in the 7500x7500 matrix so it can be assumed that this is roughly the problem size where my implementation becomes generally more effective with more nodes.

This is shown to be true when we consider the 10000x10000 matrix which continuously decreases (with the exception of the characteristic spikes as we go from the maximum cores per node to the minimum).

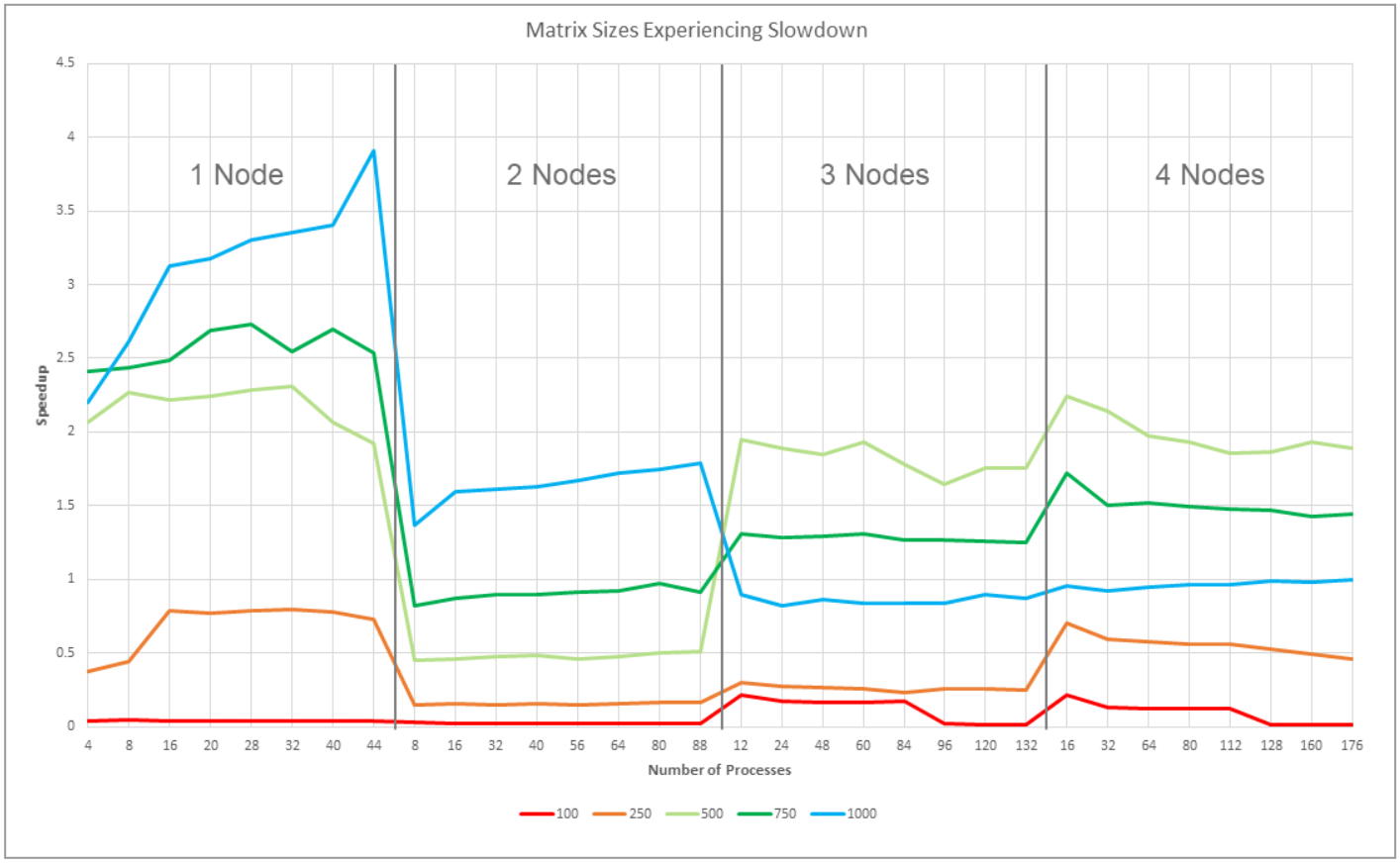
Matrix sizes across the board above 1000x1000 are faster (using one node) than my shared memory implementation, likely due to the use of scattering in place of mutexes. This only improves further with more nodes.

I have decided to plot the extreme matrix size separately so that the large matrix trends are clear:



The increase in performance with more nodes does not appear to increase any more dramatically than the large matrix cases, so my conclusion that there is general improvement for these matrix sizes may be flawed. It would be interesting to test how the runtime changes with greater than 4 nodes.

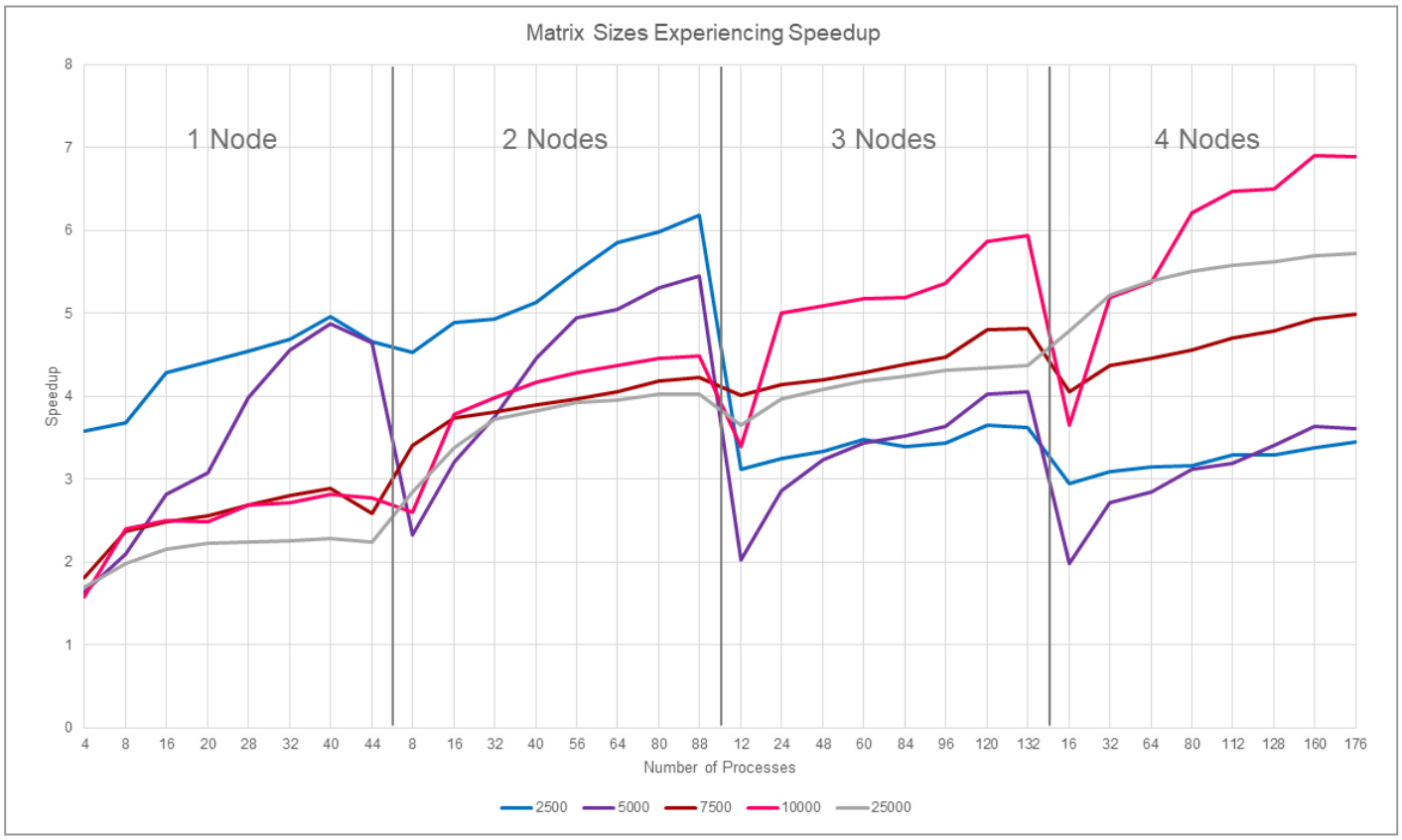
Speedup



There are some slight differences in how the sequential and distributed versions of the program work so values for speedup and efficiency should be taken with a pinch of salt, but the similarities between the 'critical section' where relaxation is performed make it, in my mind, appropriate to use the information they give us for our conclusions.

Speedup is defined as: S_p = *Sequential Speed / Parallel Speed* [3]

The speedup calculations in the graph above reinforce the conclusions from the runtime analysis. The smallest matrix sizes experience general slowdown, while mid-sized matrices experience it only due to the initial overheads associated with having more nodes. 1000x1000 is, again, an odd case where the program slows down as more nodes are added, though we can see it almost reaches 1:1 speedup with 176 processes, so perhaps with node counts beyond 4 this matrix size would see some speedup.



The largest matrices exhibit some impressive speedup. 2500x2500 and 5000x5000 still seem to fall within the hypothetical overhead vs benefit sweet spot and have an overall negative and flat trajectory respectively. 7500x7500 and beyond all exhibit general speedup growth with node growth. The lowest values at 4 cores per node get higher with each added node, as do the peaks. There appears to be no indication of a plateau, so I wonder whether the speedup would continue to increase beyond what has been tested here since none of the results are even close to superlinear, and whether we would eventually come close to the Amdahl limit.

Comparing these results to the shared memory implementation:

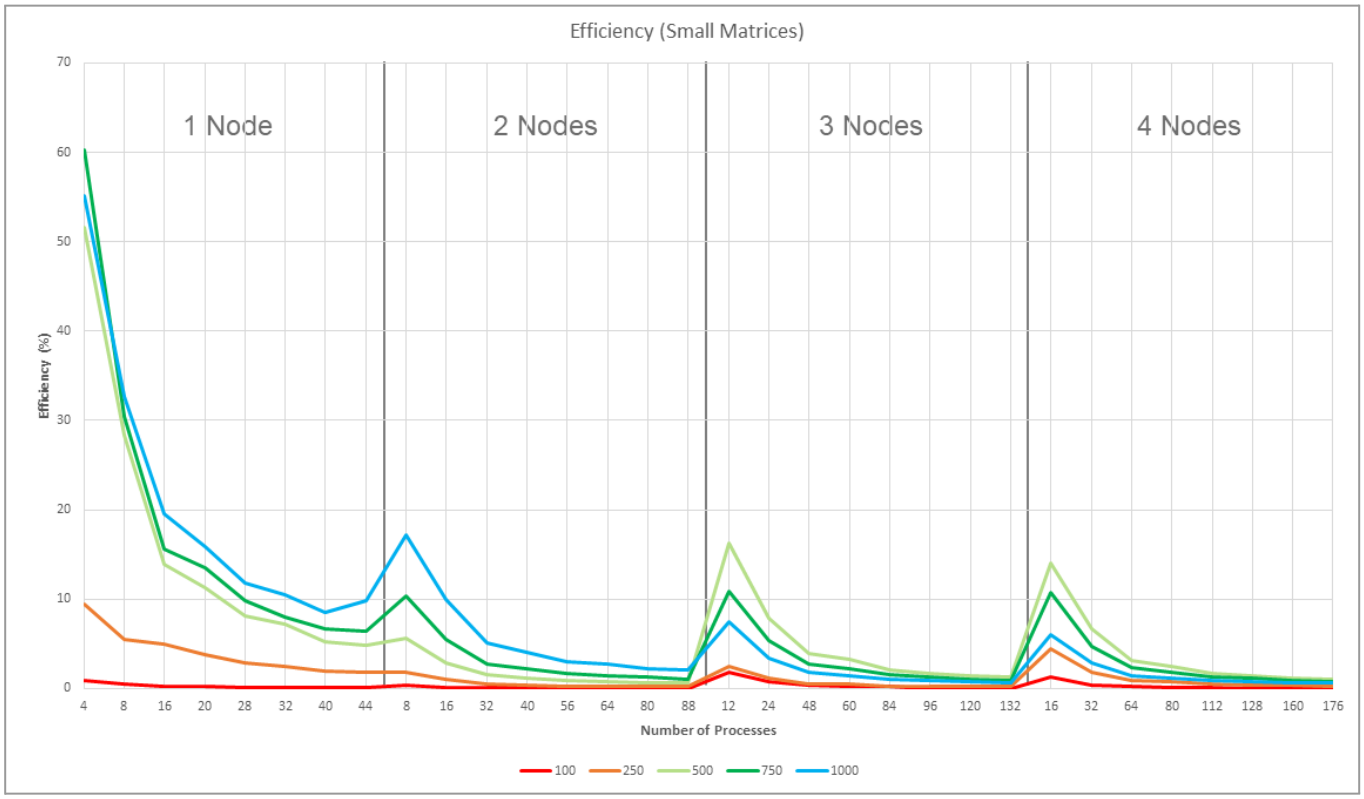
Smaller matrices perform worse, regardless of node count. The 250x250 matrix, of which only one test scored below 1 for the shared memory program, experiences slowdown throughout with the distributed memory program.

Larger matrices perform better. One of the conclusions I drew from the shared memory program is that its effectiveness when using large matrices appeared to level off and even decrease at 5000x5000. The distributed memory program's results are clear, with a continuous increase in speedup as we add more processes.

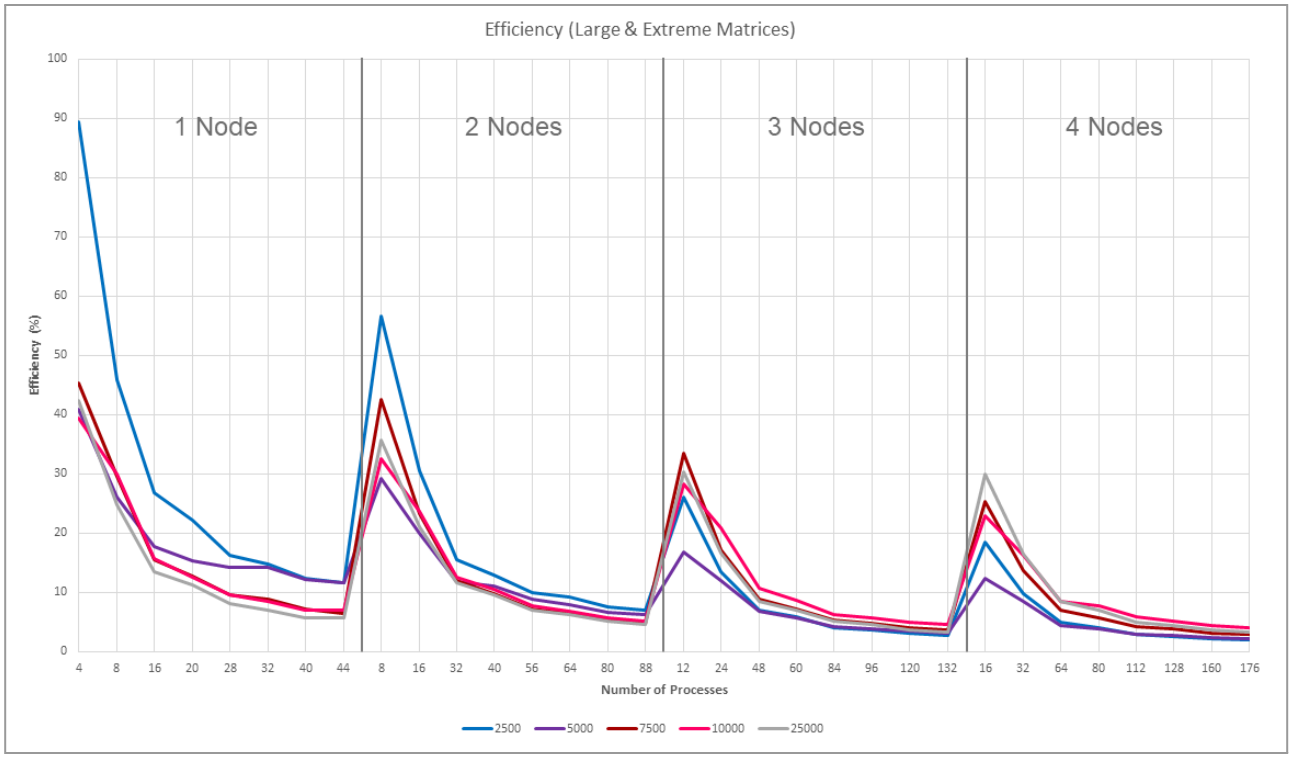
Efficiency

Speedup can be a misleading indicator of a program's effectiveness in practice. We can find how well the processing power at our disposal is being used by calculating the efficiency.

Efficiency is defined as: E_p = *Speedup value / Number of processors used* [4]



The efficiency follows a predictable pattern. Since there isn't much variation in speedup for any given matrix size for configurations that have 2-4 nodes, the larger number of processes that we are dividing by causes any increase in speedup to be negated when we find the efficiency. Using 3 nodes, each with 4 cores is just as effective as 4 nodes with 4 cores each when relaxing a 750x750 matrix.



A similar pattern emerges with large matrices, where adding a new node but keeping the cores per node constant yields negligible benefits. The average values are much higher here than with the smaller matrices.

What is particularly interesting about the efficiency results, and not strictly obvious from the graphs, is how efficiency changes with a fixed matrix size and fixed total number of processes, but a varying number of nodes and cores per node:

	100	250	500	750	1000	2500	5000	7500	10000	25000
1 Node, 16 Cores	0.25833	4.90545	13.8701	15.5569	19.5481	26.7784	17.6274	15.4946	15.5791	13.4939
2 Nodes, 8 Cores	0.14041	0.97776	2.89194	5.45734	9.95601	30.5505	19.9957	23.3317	23.6011	21.0895
4 Nodes, 4 Cores	1.32587	4.41388	13.9963	10.7365	5.95902	18.4114	12.3349	25.3356	22.8150	29.8720

	100	250	500	750	1000	2500	5000	7500	10000	25000
1 Node, 32 Cores	0.12134	2.47980	7.23083	7.95182	10.4910	14.6490	14.2382	8.76425	8.49883	7.04371
2 Nodes, 16 Cores	0.06295	0.46678	1.48409	2.78779	5.03393	15.3887	11.7191	11.9174	12.4424	11.6499
4 Nodes, 8 Cores	0.40702	1.84915	6.70494	4.69870	2.87995	9.66148	8.46630	13.6444	16.2202	16.3168

Taking excerpts from the raw data with a fixed number of 16 and 32 processes across 1, 2 and 4 node configurations, we can see something of a ‘shift’ effect, where the peak efficiency value across the matrix sizes shifts toward larger problem sizes, while the efficiency for small problem sizes gets worse before rising again. I imagine that the results for bigger matrices such as 50000x50000 and beyond would confirm this, and that adding even more nodes with less cores each would continue the trend.

Conclusion

The decision to avoid mutexes in favor of scattering the problem was a necessary and highly fruitful one. My defining conclusion when implementing shared memory was that Gustafson’s optimistic perspective - that parallelism is at its best when we are increasing the problem size - is correct. My distributed implementation supports the same conclusion, handling very large matrix sizes effectively.

The values for efficiency were considerably higher than the shared memory system, however the fact that they did not increase with more nodes is concerning. I believe this is due to my approach. In avoiding the overuse of message passing, I may have also not been making full use of the processing power available. Perhaps an implementation that does in fact pass the full matrix itself or differently allocated chunks around child processes more frequently would have greater speedup values.

The relationship between node count and core count with a fixed number of processes is noteworthy. We can conclude that a distributed system with multiple nodes is likely to be far more efficient than a shared memory system running the same number of processes - assuming a large problem size.

References

[1] Microsoft, 2018, MPI_Scatterv Function [Online]
Available from:
https://learn.microsoft.com/en-us/message-passing-interface/mpi-scatterv-function
[Accessed 15/12/22]

[2] Microsoft, 2018, MPI_Wtime Function [Online]
Available from:
https://learn.microsoft.com/en-us/message-passing-interface/mpi-wtime-function
[Accessed 15/12/22]

[3] Bradford R., 2022, *CM30225 Lecture Notes 06* [Online]
Available from:
https://people.bath.ac.uk/masrjb/CourseNotes/Notes.bpo/CM30225/slides06.pdf
[Accessed 27/10/22]

[4] Bradford R., 2022, *CM30225 Lecture Notes 07* [Online]
Available from:
https://people.bath.ac.uk/masrjb/CourseNotes/Notes.bpo/CM30225/slides07.pdf
[Accessed 28/10/22]

UNIVERSITY OF BATH

Appendix

Data For 1 Node:

Execution time tests (seconds) - Number of processors against matrix size

	100	250	500	750	1000	2500	5000	7500	10000	25000
Sequential	0.004966	0.025371	0.100158	0.20846	0.431973	2.70714	10.5143	28.4933	40.873	265.1665
4	0.134743	0.06738	0.048475	0.086486	0.196037	0.75652	6.419271	15.7211	25.92113	156.6439
8	0.113261	0.057724	0.044219	0.085668	0.165143	0.737261	5.031526	12.05127	17.08133	134.1928
16	0.120143	0.032325	0.045132	0.083749	0.138112	0.631838	3.727961	11.49321	16.3973	122.8172
20	0.122439	0.033018	0.044673	0.077578	0.135853	0.612668	3.421648	11.16659	16.418552	119.1237
28	0.124167	0.032234	0.043814	0.076292	0.130964	0.596103	2.63921	10.61527	15.184737	118.1827
32	0.127885	0.031972	0.043286	0.081923	0.128673	0.577499	2.307665	10.15963	15.028892	117.6432
40	0.132765	0.032465	0.048543	0.077412	0.126812	0.545334	2.159662	9.885704	14.531471	116.1235
44	0.136121	0.034961	0.052092	0.082134	0.110425	0.581209	2.261884	11.02688	14.751358	118.4239

Speedup calculations = Sequential time / Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
4	0.036855	0.376536	2.066178	2.410332	2.203527	3.5784116	1.637927	1.812424	1.5768216	1.692797
8	0.043845	0.439522	2.265044	2.433347	2.6157511	3.671888	2.089684	2.364340	2.3928464	1.976011
16	0.041334	0.784872	2.219223	2.489104	3.127700	4.284547	2.820388	2.479142	2.4926664	2.159033
20	0.040558	0.768399	2.242025	2.687102	3.179708	4.418608	3.072875	2.551656	2.4894399	2.225975
28	0.039994	0.787088	2.285981	2.732396	3.298410	4.541396	3.983881	2.684180	2.6917160	2.243699
32	0.038831	0.793538	2.313865	2.544584	3.357137	4.687696	4.556250	2.804560	2.719628	2.253988
40	0.037404	0.781487	2.063284	2.692864	3.406404	4.964187	4.868493	2.882273	2.812722	2.283485
44	0.036482	0.725694	1.922713	2.538047	3.9119130	4.657773	4.648470	2.583985	2.7707957	2.239128

Efficiency calculations = Sequential time / P processors * Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
4	0.92138	9.41340	51.6544	60.2583	55.0881	89.4602	40.9481	45.3106	39.42054	42.3199
8	0.54807	5.49403	28.3130	30.4168	32.6968	45.8986	26.1210	29.5542	29.91058	24.7001
16	0.25833	4.90545	13.8701	15.5569	19.5481	26.7784	17.6274	15.4946	15.57916	13.4939
20	0.20279	3.84199	11.2101	13.4355	15.8985	22.0930	15.3643	12.7582	12.44720	11.1298
28	0.14283	2.81102	8.16422	9.75855	11.7800	16.2192	14.2281	9.58635	9.613272	8.01321
32	0.12134	2.47980	7.23083	7.95182	10.4910	14.6490	14.2382	8.76425	8.498838	7.04371
40	0.09351	1.95371	5.15821	6.73216	8.51601	12.4104	12.1712	7.20568	7.031807	5.70871
44	0.09120	1.81423	4.80678	6.34511	9.77978	11.6444	11.6211	6.45996	6.926989	5.59782

UNIVERSITY OF BATH

Data For 2 Nodes:

Execution time tests (seconds) - Number of processors against matrix size

	100	250	500	750	1000	2500	5000	7500	10000	25000
Sequential	0.00497	0.02537	0.10016	0.20846	0.43197	2.70714	10.51430	28.49330	40.87300	265.1665
8	0.19016	0.17373	0.22044	0.25338	0.31502	0.59742	4.50862	8.37202	15.71921	93.10496
16	0.22104	0.16217	0.21646	0.23874	0.27118	0.55382	3.28642	7.63265	10.82388	78.58363
32	0.24652	0.16985	0.21090	0.23368	0.26816	0.54974	2.80371	7.47152	10.26553	71.12864
40	0.23442	0.16385	0.20752	0.23199	0.26538	0.52743	2.36390	7.31740	9.81912	69.43130
56	0.26180	0.17175	0.21654	0.22898	0.25865	0.49134	2.12659	7.17295	9.54166	67.71824
64	0.27466	0.15963	0.20953	0.22625	0.25124	0.46297	2.08417	7.02638	9.35241	67.01239
80	0.25192	0.15756	0.19985	0.21388	0.24788	0.45266	1.98434	6.81573	9.17863	65.81736
88	0.27557	0.15399	0.19796	0.22857	0.24165	0.43820	1.93142	6.75329	9.12868	65.79288

Speedup calculations = Sequential time / Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
8	0.02611	0.14604	0.45437	0.82271	1.37126	4.53142	2.33204	3.40340	2.60019	2.84804
16	0.02247	0.15644	0.46271	0.87317	1.59296	4.88809	3.19932	3.73308	3.77619	3.37432
32	0.02014	0.14937	0.47491	0.89209	1.61086	4.92441	3.75014	3.81359	3.98158	3.72799
40	0.02118	0.15484	0.48264	0.89859	1.62775	5.13269	4.44787	3.89391	4.16259	3.81912
56	0.01897	0.14772	0.46255	0.91040	1.67014	5.50967	4.94421	3.97233	4.28364	3.91573
64	0.01808	0.15894	0.47801	0.92135	1.71937	5.84736	5.04483	4.05519	4.37032	3.95698
80	0.01971	0.16102	0.50116	0.97468	1.74269	5.98058	5.29864	4.18052	4.45306	4.02882
88	0.01802	0.16476	0.50595	0.91203	1.78758	6.17785	5.44382	4.21917	4.47743	4.03032

Efficiency calculations = Sequential time / P processors * Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
8	0.32643	1.82548	5.67957	10.28392	17.14070	56.64279	29.15054	42.54247	32.50242	35.60048
16	0.14042	0.97777	2.89194	5.45734	9.95601	30.55054	19.99576	23.33175	23.60117	21.08952
32	0.06295	0.46679	1.48409	2.78779	5.03394	15.38878	11.71917	11.91746	12.44243	11.64995
40	0.05296	0.38710	1.20661	2.24647	4.06937	12.83173	11.11966	9.73478	10.40648	9.54780
56	0.03387	0.26379	0.82598	1.62572	2.98239	9.83870	8.82895	7.09344	7.64935	6.99238
64	0.02825	0.24834	0.74688	1.43962	2.68652	9.13650	7.88255	6.33623	6.82862	6.18278
80	0.02464	0.20128	0.62644	1.21835	2.17836	7.47573	6.62329	5.22565	5.56632	5.03603
88	0.02048	0.18723	0.57494	1.03640	2.03134	7.02028	6.18616	4.79451	5.08798	4.57991

UNIVERSITY OF BATH

Data For 3 Nodes:

Execution time tests (seconds) - Number of processors against matrix size

	100	250	500	750	1000	2500	5000	7500	10000	25000
Sequential	0.00497	0.02537	0.10016	0.20846	0.43197	2.70714	10.51430	28.49330	40.87300	265.1665
12	0.02336	0.08527	0.05139	0.15947	0.48153	0.86726	5.20125	7.11395	12.03568	72.76356
24	0.02813	0.09298	0.05299	0.16212	0.52733	0.83466	3.68736	6.88653	8.16674	66.78468
48	0.02971	0.09499	0.05419	0.16124	0.50267	0.81256	3.24864	6.79736	8.02827	64.87651
60	0.03013	0.09836	0.55192	0.15891	0.51828	0.77875	3.06781	6.65368	7.89295	63.35764
84	0.02934	0.10786	0.56192	0.16413	0.51798	0.79988	2.98564	6.49969	7.88675	62.64674
96	0.28684	0.09783	0.61013	0.16467	0.51446	0.78874	2.88712	6.38205	7.61456	61.57900
120	0.31839	0.09737	0.57129	0.16599	0.48187	0.74311	2.61326	5.93786	6.96723	61.14140
132	0.32276	0.10357	0.57214	0.16708	0.49762	0.74883	2.59065	5.91865	6.88766	60.69372

Speedup calculations = Sequential time / Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
12	0.21256	0.29753	1.94886	1.30718	0.89708	3.12150	2.02150	4.00527	3.39599	3.64422
24	0.17656	0.27287	1.89027	1.28581	0.81917	3.24341	2.85145	4.13754	5.00481	3.97047
48	0.16713	0.26710	1.84826	1.29286	0.85936	3.33160	3.23653	4.19182	5.09113	4.08725
60	0.16483	0.25795	0.18147	1.31180	0.83347	3.47628	3.42730	4.28234	5.17842	4.18523
84	0.16925	0.23523	0.17824	1.27010	0.83395	3.38444	3.52162	4.38379	5.18249	4.23273
96	0.01731	0.25935	0.16416	1.26594	0.83966	3.43222	3.64179	4.46460	5.36774	4.30612
120	0.01560	0.26056	0.17532	1.25584	0.89645	3.64300	4.02345	4.79858	5.86646	4.33694
132	0.01539	0.24497	0.17506	1.24764	0.86807	3.61516	4.05855	4.81415	5.93424	4.36893

Efficiency calculations = Sequential time / P processors * Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
12	1.77132	2.47939	16.2405	10.89317	7.47564	26.01247	16.84581	33.37728	28.29989	30.36851
24	0.73565	1.13696	7.87614	5.35756	3.41322	13.51422	11.88103	17.23977	20.85338	16.54362
48	0.34818	0.55646	3.85055	2.69347	1.79032	6.94084	6.74277	8.73296	10.60653	8.51510
60	0.27472	0.42991	0.30245	2.18633	1.38912	5.79380	5.71216	7.13723	8.63070	6.97539
84	0.20149	0.28004	0.21219	1.51202	0.99280	4.02910	4.19240	5.21880	6.16963	5.03896
96	0.01803	0.27015	0.17100	1.31869	0.87465	3.57523	3.79353	4.65063	5.59140	4.48554
120	0.01300	0.21714	0.14610	1.04653	0.74704	3.03583	3.35287	3.99882	4.88872	3.61412
132	0.01166	0.18558	0.13262	0.94518	0.65763	2.73876	3.07466	3.64708	4.49564	3.30979

UNIVERSITY OF BATH

Data For 4 Nodes:

Execution time tests (seconds) - Number of processors against matrix size

	100	250	500	750	1000	2500	5000	7500	10000	25000
Sequential	0.00497	0.02537	0.10016	0.20846	0.43197	2.70714	10.51430	28.49330	40.87300	265.1665
16	0.02341	0.03593	0.04473	0.12135	0.45307	0.91898	5.32748	7.02896	11.19685	55.47963
32	0.03813	0.04288	0.04668	0.13864	0.46873	0.87562	3.88094	6.52586	7.87463	50.78475
64	0.03971	0.04399	0.05067	0.13697	0.45826	0.86099	3.69837	6.39736	7.59737	49.20579
80	0.04013	0.04519	0.05186	0.13975	0.44985	0.85525	3.37642	6.24786	6.58256	48.10674
112	0.03934	0.04567	0.05397	0.14081	0.44678	0.82447	3.29987	6.06725	6.31997	47.57622
128	0.38684	0.04836	0.05383	0.14164	0.43867	0.82210	3.08968	5.95280	6.28754	47.18197
160	0.41839	0.05190	0.05178	0.14598	0.44186	0.80346	2.88877	5.78794	5.91793	46.56836
176	0.42276	0.05486	0.05299	0.14433	0.43265	0.78365	2.91433	5.70756	5.92907	46.37625

Speedup calculations = Sequential time / Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
16	0.21214	0.70622	2.23942	1.71784	0.95344	2.94583	1.97360	4.05370	3.65040	4.77953
32	0.13025	0.59173	2.14558	1.50358	0.92159	3.09168	2.70922	4.36621	5.19047	5.22138
64	0.12504	0.57678	1.97659	1.52191	0.94264	3.14424	2.84296	4.45392	5.37989	5.38893
80	0.12375	0.56147	1.93120	1.49170	0.96026	3.16533	3.11404	4.56049	6.20928	5.51205
112	0.12623	0.55553	1.85574	1.48049	0.96686	3.28350	3.18628	4.69625	6.46728	5.57351
128	0.01284	0.52467	1.86057	1.47176	0.98473	3.29295	3.40304	4.78654	6.50063	5.62008
160	0.01187	0.48887	1.93419	1.42805	0.97762	3.36936	3.63972	4.92288	6.90663	5.69414
176	0.01175	0.46245	1.89024	1.44435	0.99843	3.45454	3.60780	4.99220	6.89366	5.71772

Efficiency calculations = Sequential time / P processors * Parallel time

	100	250	500	750	1000	2500	5000	7500	10000	25000
16	1.32588	4.41388	13.9963	10.73651	5.95902	18.41141	12.33498	25.33565	22.81502	29.87206
32	0.40703	1.84916	6.70495	4.69870	2.87996	9.66149	8.46630	13.64442	16.22021	16.31682
64	0.19538	0.90123	3.08843	2.37798	1.47288	4.91287	4.44212	6.95924	8.40608	8.42020
80	0.15469	0.70183	2.41400	1.86462	1.20033	3.95667	3.89255	5.70061	7.76160	6.89006
112	0.11271	0.49601	1.65691	1.32186	0.86327	2.93170	2.84489	4.19308	5.77436	4.97635
128	0.01003	0.40990	1.45357	1.14981	0.76932	2.57262	2.65863	3.73948	5.07862	4.39069
160	0.00742	0.30555	1.20887	0.89253	0.61101	2.10585	2.27483	3.07680	4.31665	3.55883
176	0.00667	0.26276	1.07400	0.82065	0.56729	1.96281	2.04988	2.83648	3.91685	3.24871