CM30225 Parallel Computing
Assessed Coursework Assignment 2

Adam Jaamour

07 january, 2019

1 Introduction

1.1 MPI Parallelism

This paper presents a solution to implementing MPI (Message Passing Interface) in a distributed memory configuration to the *relaxation technique* on square arrays using C, Balena and MPI. The relaxation technique is a solution to differential equations, which is achieved by having a square array of values and repeatedly replacing a value with the average of its four neighbours, excepting boundaries values which remain fixed. This process is repeated until all values settle down to within a given precision.

The core aspect of code is divided into three sections: the first section is common to all the process and focuses on parsing the command line arguments before initialising the MPI environment and start measure time. The second section is executed by the root/master process only, and the third one is executed by all the children/slave processes only. Code improvements are mentioned in Section 4.

1.1.1 Root Process

The root process starts by initialising and populating the square array to relax. It then determines what rows of the array each child process gets attributed to relax on their own before send the determined portion of the array to each child. It only sends the array once to avoid unnecessary large communications. The children processes then communicate between themselves to get rows from other processes.

The root process then waits for each child process to have received its own portion of the array and waits for each process to tell it if they relaxed the array within the precision. If they did not, then root process tells each child process to continue the iteration loop and relax their sub array one more time.

Once all the children processes tell the root process that they finished relaxing the array, the root process tells them to stop relaxing their sub arrays. It finally starts collecting each of the sub arrays from children processes one by one and merging them with the original square array to relax. Once the original array has been stitched back together, the root process stops the time measurement and waits for all child processes to exit gracefully before proceeding to exit itself.

1.1.2 Children Processes

The children processes start by receiving their assigned portion of the array. If it is the first time they are running, they receive their entire portion of the array that they will need to relax directly from the root process.

For all future iterations, they communicate with their neighbouring processes to retrieve the updated first and last rows of their portion of the array. For example, if there are 7 children processes, process number 5 will get the first row of its sub array from the last row of process 4, and it will get the last row of its sub array from the first row of process 6. Edge cases are considered for the first and last children processes for them to avoid waiting to receive or send a row to the process before or after them, which do not exist, which would otherwise cause the entire system to block. Once the processes have received their updated first and last rows, they relax their sub array and check if they relaxed it within the precision or not.

They then proceed to tell the root process if they are within precision or not, which then tells them to continue or stop relaxing their sub array based on what the other processes told the root process regarding their own progression with their sub arrays. Once the root process tells them to stop relaxing their sub arrays, each child process sends back their relaxed portion of the array to the root process to merge it back to the other portions relaxed by the other children processes. The children processes can finally exit gracefully.

1.1.3 Time Measurements

Time was measured using *MPI_WTime()* as it a high-resolution wall clock that measures the elapsed wall clock time rather than measure the CPU time. The elapsed time starts being measured after the initialisation of variables and the MPI environment and before the command line arguments parsing. This allows a precise time to be measured, excluding as much sequential overhead as possible.

```c
double start_MPI, end_MPI, elapsed_time;
rc = MPI_Init(NULL, NULL);
start_MPI = MPI_Wtime();
if (world_rank == root_process_id) { // root process
    /* manage children processes here */

    /* reconstruct array once relaxation finished */

    end_MPI = MPI_Wtime();
    elapsed_time = end_MPI - start_MPI;
    printf("Total time = %f seconds\n\n", elapsed_time);
} else {
    /* children processes here */
}
MPI_Finalize();
```

1.2 Source Code

- *main.c*: Contains the main loop for initialising and finalising the variables and the MPI environment, as well as the root and children code to perform relaxation.
- *array_helpers.c*: Contains functions for initialising, allocating and manipulating arrays.
- *print_helper.c*: Contains functions for printing data to the terminal.

1.3 How To Run

The program can be compiled using either:

- *mpicc -Wall -Wextra -Wconversion main.c -o distributed_relaxation -lm* or
- *make* (using the provided makefile).

The program can then be run using: *mpirun -np (num_processes) ./distributed_relaxation -d (dimension) -p (precision) -debug (debug mode)*, where:

- *-np* corresponds to the number of processes;
- *-d* corresponds to the dimensions of the square array;
- *-p* corresponds to the precision of the relaxation;
- *-debug* corresponds to the debug mode (0: only essential information, 1: row allocation logs, 2: process IDs logs, 3: initial and final arrays, 4: iteration debugging data).

Examples:

- *mpirun -np 4 ./distributed_relaxation -d 15 -p 0.1 -debug 1*
- *mpirun -np 16 ./distributed_relaxation -d 500 -p 0.01 -debug 0*

If values are absent or wrongfully entered, default values will be used.

2 Correctness Testing

Correctness testing is used to show that the program computes the same output when running sequentially and in parallel. To ensure that the same initial array is used across the test, the *srand* function with a specific seed is used (located in array_helpers.c):

```c
srand(1000);
sq_array[i * dim + j] = (double)(rand() % 10);
```

The first test consists in manually relaxing the initial array while showing the steps. Then, the same array is relaxed using the sequential program. Finally, the array is relaxed using the parallel program. If each test computes the same correct output (or within the specified precision), then the assumption that the program computes the correct final array regardless of the dimension, precision or number of threads can be made.

Two correctness tests using different parameters are carried out, with results in the Appendix:

- See Appendix A for Correctness Testing 1
- See Appendix B for Correctness Testing 2

The manual, sequential and parallel tests all output the same final array for both tests (or within the specific precision for the parallel output). It is therefore safe to assume that the program used for the scalability investigation in the next section computes the correct answer for any given square array irrespective of its size, the precision to relax to, and the number of processes to use.

3 Scalability Investigation

This section focuses on investigating the effects of changing problem size parameters such as the dimensions of the square array to relax and the precision to perform relaxation to, and changing the processing power parameters, such the number of cores to use on Balena and the number of parallel processes to run with. These tests are analysed by calculating the speedup and the efficiency of the parallel program while making references to Amdahl's Law and Gustafson's Law.

Throughout the following scalability investigations, data retrieved from runs on Balena uses the same randomly-generated initial array values to ensure accuracy throughout the various tests. Additionally, the number of parallel processes used in the tests do not exceed 64, as Balena allows up to 4 maximum cores, with 16 nodes each, to be used for testing.

3.1 Amdahl's Law: Speedup Investigation

This tests in this section focuses on fixed problem sizes. Using the elapsed time of the program when running sequentially and in parallel, the program's speedup can be calculated by comparing both time measurements using the following equation:

$$Speedup = \frac{Time_{sequential}}{Time_{parallel}} \tag{1}$$

Amdahl's Law states that the speedup of a parallel program should increase as more processing power is used, while using a fixed problem size (same array dimensions and precision). This first test uses a fixed size problem with dimension of 1024x1024 and a of precision 0.1, while using 2 to 64 parallel processes, as seen in Figure 1. The graph clearly shows that the program speeds up as the number of parallel processes increases. However, the more parallel processes there are, the weaker the speedup slope is: the speedup increases by +11 when using 48 processes instead of 32, but it only increases by +3 when using 64 processes rather than 48. This weaker speedup increase is probably due to the communication overhead of passing around large portions of array.

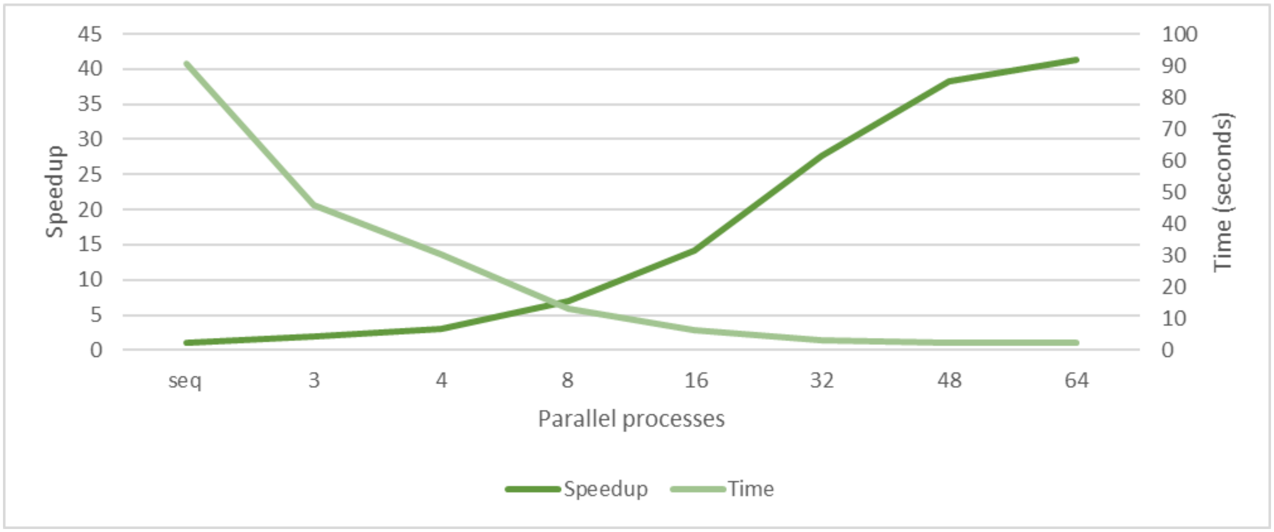

Figure 1: Graph depicting the speedup for a fixed problem size (dimension 1024 and precision 0.1) with increasing processing power.

To go deeper into the analysis of a fixed problem size, the same tests were run using larger square arrays (2048 instead of 1024) and user more precision (0.001 instead of 0.1). These larger problem sizes for the same amount of processing power (Balena cores) should increase the speedup for parallel program according to Amdahl's Law. The results can be seen in Figure 2. Indeed, increasing the array dimensions from 1024 to 2048 increases the speedup for the parallel program when using more parallel processes relative to smaller problem size (blue series VS yellow series). This is because the larger problem size benefits more from using an increased number of cores in parallel and slightly compensates the sequential overhead.

The same result should have been expected by using more precision (0.001 instead of 0.1), but in this test array sizes of 1024 could not be used with a precision greater than 0.01 without timing out on Balena (exceeding 10 minutes).

However, according to Amdahl's Law, there is a limit on how much speedup can be achieved, no matter how much additional processing power is used. This upper bound is due to the sequential aspects of the code that cannot be avoided. For example, communications between the processes, where one child process send another child process one of its rows to continue the relaxation, will always be sequential and cannot be stopped, no matter how much processing power is available.

3.2 Gustafson's Law: Speedup Investigation

The following tests in this section now focus on using fixed processing power (a fixed number of Balena cores and parallel processes running on those cores) and will use varying problem

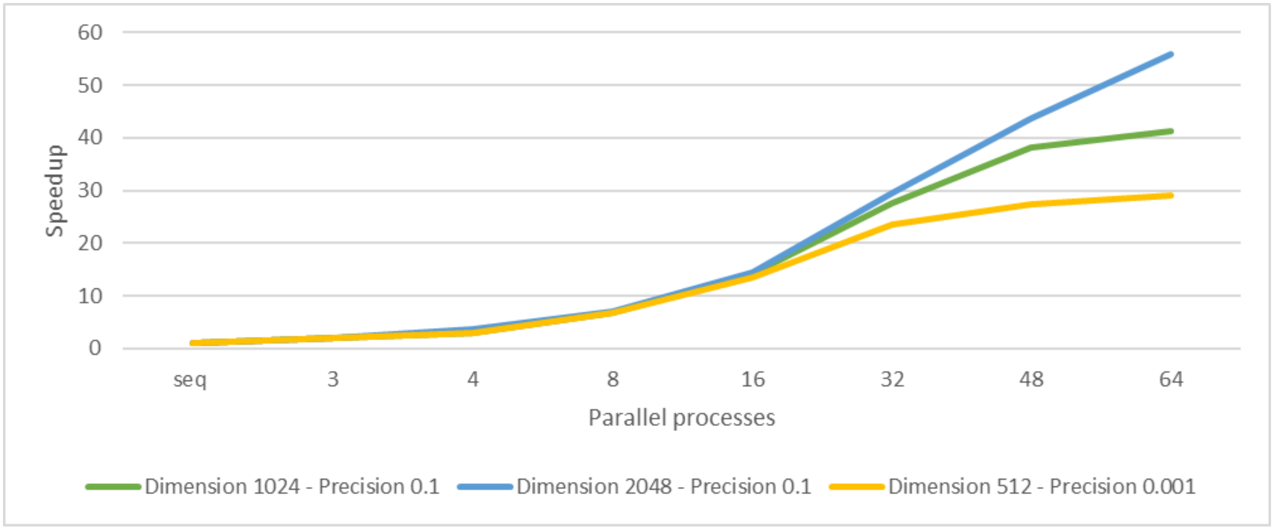

Figure 2: Graph depicting the speedup for a fixed problem size with increasing processing power using different parameters. In green, the results from Figure 1 to compare with the larger problem sizes.

sizes, particularly array dimensions and precision. For the next tests, 2 Balena cores were used with 32 parallel processes.

Gustafson's law states that as the problem size gets larger, the time needed for the sequential portion of the code decreases relatively. This means that as bigger arrays and higher precisions are used, the speedup should start trending towards a number. If the sequential section of the code can be completely eliminated with the use of additional parallel processes, then the speedup can be equal to the number of parallel processes being used. This is considered to be a perfect speedup scenario, where for example if 16 parallel processes are used, then a speedup of 16 would be achieved. Getting a speedup above the number of parallel processes being used would be classified as a super-linear speedup, a case where the speedup is higher than the number of processes used. This can be caused when running a program with low communication overhead, which does not apply to this array-relaxation problem, or when running in cache memory, which is does not apply to this problem as well. The purpose of the tests in this section is to examine how close the speedup can get to the number of parallel processes being used when increasing the array dimensions and the precision.

The first test focuses on varying the problem size through the dimension of the square array to relax, starting from 64x64 and going up to 2048x2048, while using 32 parallel processes and a precision of 0.1. According to Gustafson's Law, the speedup cannot be larger than the number of parallel processes used, so it should not exceed 32 (actually 31 as 1 root process manages 31 children processes in this solution). The results are plotted in Figure 3.

The results from this array dimension scaling shows that the speedup increases as the array gets larger and larger, and how it starts trending towards a specific value when the array gets larger, as predicted by Gustafson's Law. Unfortunately, larger arrays could not be used to

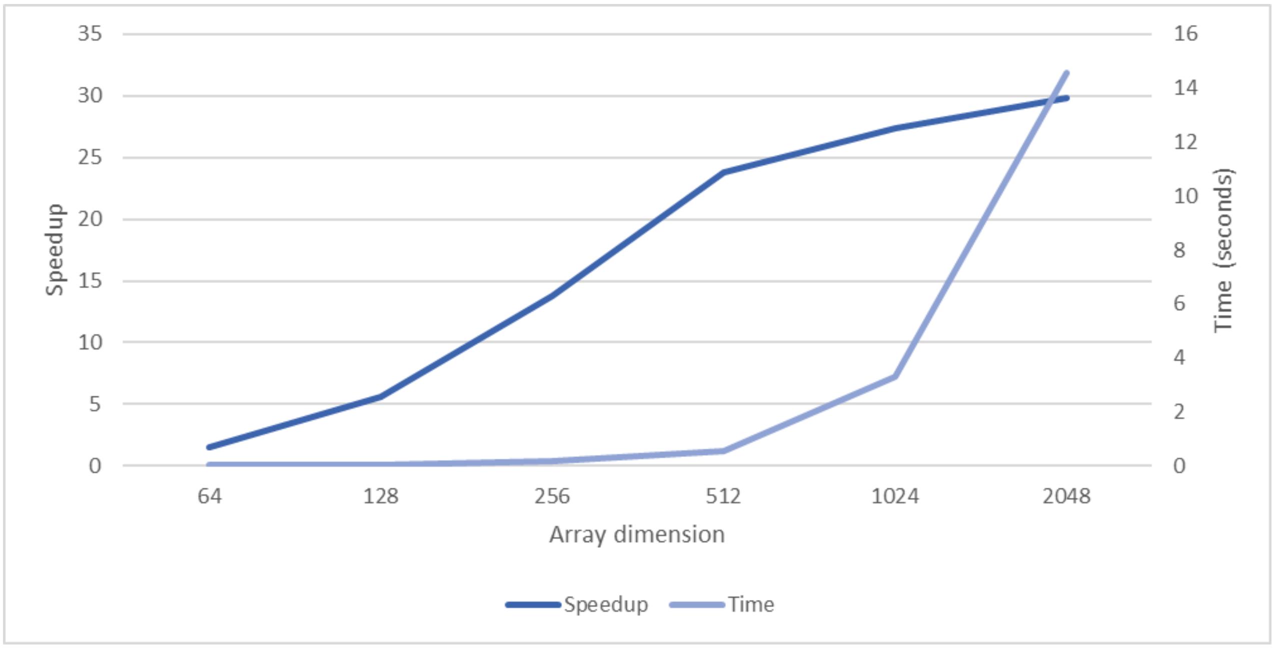

Figure 3: Graph depicting the speedup for an increasing problem size (array dimension) with fixed processing power (2 cores - 32 parallel processes) and a precision of 0.1.

get more data, as the using 4096x4096 arrays would cause the sequential program to time out (limit of 10 minutes on Balena), preventing the speedup from being calculated using the time to complete the sequential code. However a second test was done using increasing precision instead of increasing array dimension.

The second test uses an increasing problem size through the precision value, starting from 0.1 to 0.000001. The number of cores remains 2 (32 parallel processes) and array dimensions remain at 256x256 throughout the test. To re-iterate and emit the same hypothesis, according to Gustafson's Law, the speedup cannot be larger than the number of parallel processes used, so it should not exceed 32. The results are shown in Figure 4.

The results confirms the hypothesis and the results from the increasing array dimension test. It depicts that as the problem size (precision) increases, so does the speedup until it starts trending towards a speedup value of 14. This clearly confirms the point made by Gustafson's Law, with the speedup trending towards a specific number. However, although 32 parallel processes are being used, the speedup starts stagnating towards $14 < 31$. This means that there is a portion of the code that cannot be improved by implementing parallelism, it is always run sequentially. The solution would need to be improved to increase the parallelism. If it were improved, the speedup would increase and trend closer towards 31 rather than 14, without exceeding (unless smart solutions or caches are used).

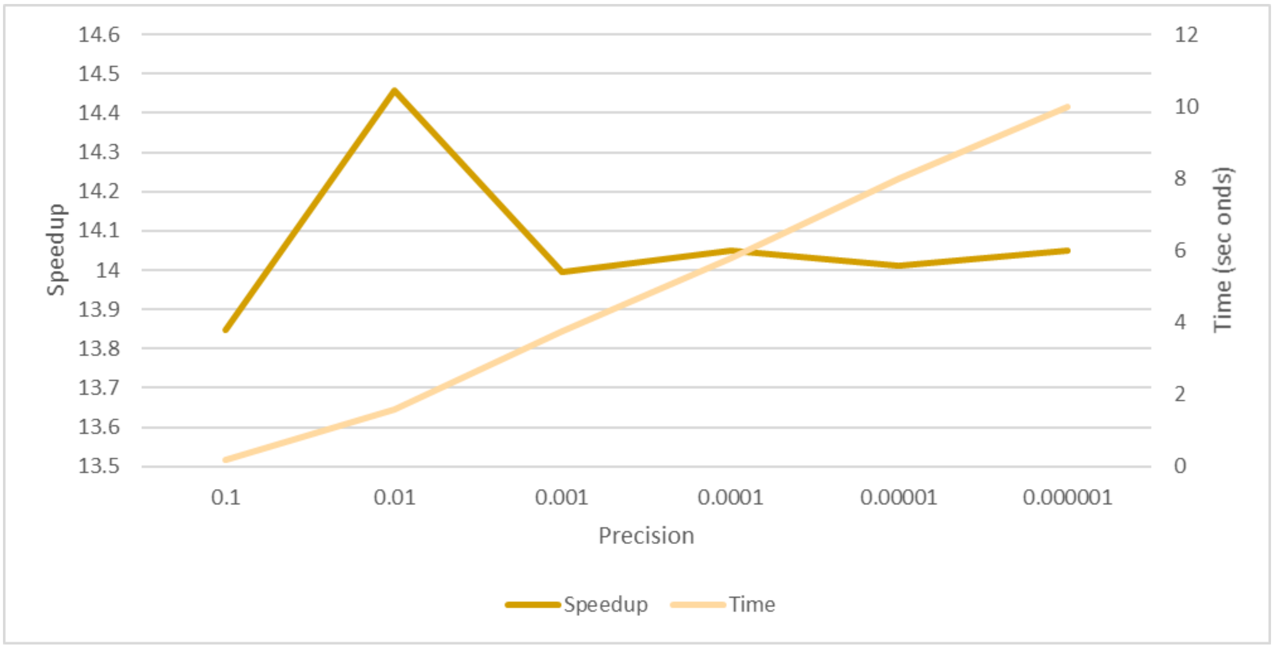

Figure 4: Graph depicting the speedup for an increasing problem size (precision) with fixed processing power (2 cores - 32 parallel processes) and a 256x256 array.

3.3 Amdahl's Law & Gustafson's Law: Efficiency Investigation

Using the same test data carried out in Sections 3.1 and 3.2, a new metric can be used by using the calculated speedup: *efficiency*.

As opposed to speedup, which only indicates the relative increase (or decrease) in speed when running in parallel rather than sequentially, efficiency can measure how efficiently the processing power is being used. A high efficiency means that the processors' time is being used efficiently, whereas a low efficiency indicates that the processors' time is being used inefficiently, thus costing money. Indeed, simply adding more and more parallel processes will not make the program run more efficiently, it will even slow it down at some point. The efficiency, expressed as a percentage %, can be calculated using the previously calculated speedup as follow:

$$Efficiency = \frac{Speedup}{ParallelProcesses} \tag{2}$$

3.3.1 Varying Fixed Problem Sizes - Increasing Processing Power

The first efficiency test uses the results from the fixed problem size and increasing processing power from Section 3.1. Amdahl's Law states that more parallel processes being used should lead to an increasing speedup, whereas Gustafson's Law states that the efficiency should start stagnating and eventually declining as more parallel processes are being used, despite the increased speedup predicted by Amdahl's Law. Results for this test can be found in Figure 5.

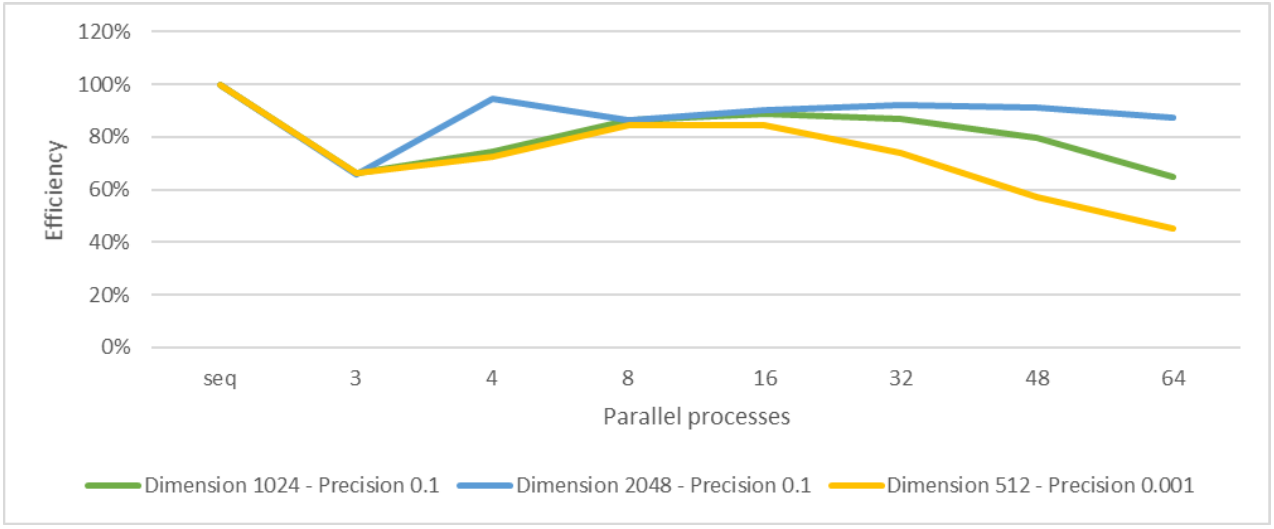

Figure 5: Graph depicting the efficiency for a fixed problem size with increasing processing power using different parameters.

The results from this test are expected. Despite the increased speedup from Figure 2 (Amdahl's Law), as more parallel processes are used, the efficiency eventually decreases (Gustafson's Law), but at different moments based on the problem size.

According the plots in Figure 5, with larger problem sizes (blue series - Dimension 2048 - Precision 0.1), the efficiency is higher and decreases less than for smaller problem sizes (yellow series - Dimension 512 - Precision 0.001). Indeed, the efficiency reaches 92% for the large problem size running with 32 parallel processes and decreases towards 87% when using 64 parallel processes, whereas it only reaches 84% for the smaller problem size when running on 16 parallel processes and decreases towards 45% when running on 64 parallel processes.

This test explains two different aspects from Gustafson's Law. The first is that a large increasing number of parallel processes used decreases the efficiency of the program as they are not being efficiently used when there are more of them. This can be due to the overhead lost in idling, for example because of the excessive use of *MPI_Wait()* in the root process, and it can also be due to sequential communications between all the processes, such as the children processes' communications with the root to specify if they finished relaxing their portion of the array, and communications between themselves to share parts of the array needed to continue relaxing the array. The second point is that the program is more efficient on a fixed-problem size when that problem is bigger (blue series VS yellow series in Figure 5) as the larger number of parallel processes is more efficiently used when communicating and passing large amounts of data between each other, overcoming some of the time lost in sequential portions of the code.

3.3.2 Increasing Problem Size - Fixed Processing Power

This efficiency test uses the results from the increasing problem size and fixed processing power from section 3.2. According to Gustafson's Law, the efficiency should increase as the number of parallel processes increases for a fixed number of parallel processes (32 used for this test). The results can be found in Figure 6.

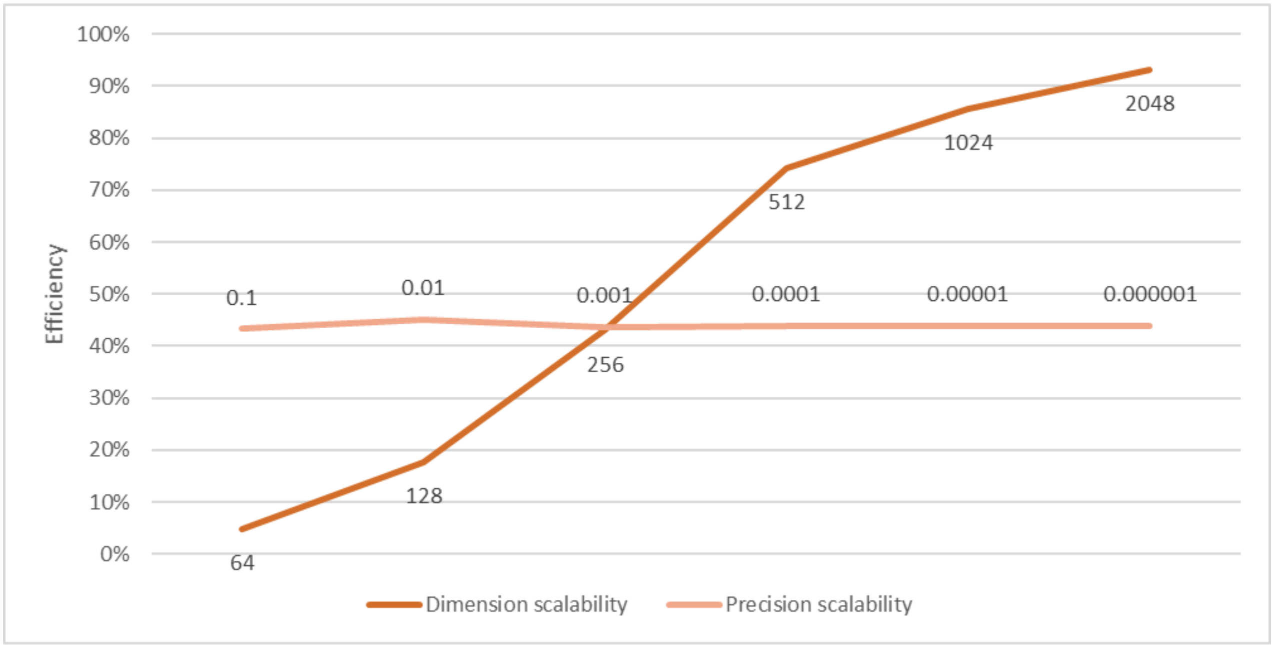

Figure 6: Graph depicting the efficiency for an increasing problem size with fixed processing power (2 cores - 32 parallel processes) using different parameters.

The efficiency of an increasing precision does show any relevant information, which may be due to the array of 256x256 being to small to show the real benefit of using multiple parallel processes (unfortunately, larger arrays could not be used as the sequential program timed out after 10 minutes on Balena). However, the efficiency does improve for increasing array dimension, and starts slowing down from arrays of sizes 512x512 and bigger. This slowdown is likely due to the increased communications overhead of moving large portions of arrays between processes, an increased number of iterations necessary and longer precision checks to see if processes finished relaxing their portions of the array. The benefits of running multiple processes is too weak compared to the overhead in communications caused by large arrays.

4 Conclusion

In conclusion, distributed memory architecture and MPI solutions scale well to large size problems, as shown by the results in this report. The larger the array or the precision was, the higher the speedup and efficiency were. The tricky part is to find the right amount of parallel processes to use with the right problem size to use the processors efficiently, thus at a relatively low cost.

The MPI solution presented in this report could be further improved by making use of the root process, which is used to manage the children processes in this solution. Instead of only managing the children processes, the root process could also assign a portion of the array to itself for it to relax while the children are relaxing their own portion. This would avoid having a process remain idle while the others relax the array.

Having a more efficient solution could have allowed larger arrays and larger precisions to be used, thus allowing more data to be gathered and displayed on the graphs.

Appendix A Correctness Testing 1

Parameters:

- Array dimension: 5x5 (populated with rounded doubles ranging from 0 to 10)
- Precision: 0.1
- Processes for parallel test: 3

Original array to relax:

0	9	3	8	4
1	3	0	8	5
8	3	4	7	3
3	6	0	1	2
7	7	1	6	6

Manual testing

Step 1:

0	9	3	8	4
1	3.25	4.5625	6.140625	5
8	5.3125	4.21875	3.589844	3
3	3.828125	2.511719	3.525391	2
7	7	1	6	6

Step 2:

0	9	3	8	4
1	4.96875	4.582031	5.292969	5
8	5.253906	3.984375	3.950684	3
3	4.441406	3.237793	3.797119	2
7	7	1	6	6

Step 3:

0	9	3	8	4
1	4.958984	4.309082	5.314941	5
8	5.346191	4.210938	4.08075	3
3	4.645996	3.413513	3.873566	2
7	7	1	6	6

Step 4:

0	9	3	8	4
1	4.913818	4.359924	5.360168	5
8	5.442688	4.324219	4.139488	3
3	4.71405	3.477959	3.904362	2
7	7	1	6	6

Step 5 - Final array (output):

```
0     9          3          8         4
1  4.950653   4.40876   5.387062   5
8  5.497231   4.380859  4.168071   3
3  4.743797   3.507255  3.918831   2
7     7          1          6         6
```

Sequential program ouput

```
0     9          3          8         4
1  4.950653   4.40876   5.387062   5
8  5.497231   4.380859  4.168071   3
3  4.743797   3.507255  3.918831   2
7     7          1          6         6
```

Parallel MPI program output (with 3 parallel processes)

```
0     9          3          8         4
1  4.950653   4.40876   5.387062   5
8  5.497231   4.380859  4.168071   3
3  4.743797   3.507255  3.918831   2
7     7          1          6         6
```

Appendix B Correctness Testing 2

Parameters:

- Array dimension: 7x7 (populated with rounded doubles ranging from 0 to 10)
- Precision: 1
- Processes for parallel test: 4

Original array to relax:

```
0  9  3  8  4  1  3
0  8  5  8  3  4  7
3  3  6  0  1  2  7
7  1  6  6  8  7  0
4  6  6  4  6  1  2
2  4  7  4  2  4  5
6  0  2  9  4  0  2
```

Manual testing

Step 1:

```
0      9          3          8          4          1      3
0  4.250000  5.312500  4.078125  3.269531  3.317383  7
3  3.562500  3.718750  3.699219  4.242188  5.389893  7
7  5.640625  5.339844  5.259766  5.625488  3.003845  0
4  4.910156  5.312500  5.143066  3.442139  3.111496  2
2  3.477539  3.697510  4.960144  4.100571  3.053017  5
6      0          2          9          4          0      2
```

Step 2:

```
0      9          3          8          4          1      3
0  4.468750  3.816406  4.696289  4.063965  4.363464  7
3  4.207031  4.265625  4.615967  4.923828  4.822784  7
7  5.364258  5.050537  5.108765  4.119644  3.013481  0
4  4.538574  4.607422  4.529617  3.965332  3.007957  2
2  2.559021  3.531647  5.290459  4.077202  3.021290  5
6      0          2          9          4          0      2
```

Step 3 - Final array (output):

```
0      9          3          8          4          1      3
0  4.255859  4.054443  5.183594  4.617722  4.360126  7
3  4.221436  4.485596  4.925446  4.621399  4.748752  7
7  5.202637  4.851105  4.606453  4.051666  2.952094  0
4  4.092270  4.251160  4.528351  3.916294  2.972419  2
2  2.405979  3.486899  5.273113  4.052674  3.006273  5
6      0          2          9          4          0      2
```

Sequential program ouput

```
0      9          3          8          4          1      3
0  4.255859  4.054443  5.183594  4.617722  4.360126  7
3  4.221436  4.485596  4.925446  4.621399  4.748752  7
7  5.202637  4.851105  4.606453  4.051666  2.952094  0
4  4.092270  4.251160  4.528351  3.916294  2.972419  2
2  2.405979  3.486899  5.273113  4.052674  3.006273  5
6      0          2          9          4          0      2
```

Parallel MPI program output (with 4 parallel processes)

```
0      9          3          8          4          1      3
0  4.281250  4.537109  5.089844  4.750977  4.107422  7
3  4.392578  4.224609  5.111328  4.058594  4.894531  7
7  4.950195  4.849609  4.221680  4.397461  2.668945  0
4  3.988281  4.238281  4.857422  3.728516  3.248047  2
2  2.467773  3.603516  5.319336  4.325195  2.950195  5
6      0          2          9          4          0      2
```

Appendix C Makefile

```
CC          = mpicc
CFLAGS      = -Wall -Wextra -Wconversion
LDFLAGS     = -lm
OBJFILES    = main.o array_helpers.o print_helpers.o
TARGET      = distributed_relaxation

all: $(TARGET)

$(TARGET): $(OBJFILES)
	$(CC) -o $(TARGET) $(OBJFILES) $(LDFLAGS)

clean:
	clear
	rm -rf $(OBJFILES) $(TARGET) *~
```

Appendix D SLURM script

```
#!/bin/bash
#SBATCH --account=cm30225
#SBATCH --partition=teaching
#SBATCH --job-name=cw2
#SBATCH --output=relaxation.%j.out
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=16
#SBATCH --mail-type=END
#SBATCH --mail-user=XXXXX@bath.ac.uk
module load intel/mpi
mpirun -np 64 ./distributed_relaxation -d 512 -p 0.001 -debug 0
```

Appendix E Raw Data

Processes	Dimension	Precision	Time	Speedup	Efficiency
seq	1024	0.1	90.656413	1	100%
3	1024	0.1	45.765167	1.980904232	66.0301%
4	1024	0.1	30.490147	2.973301933	74.3325%
8	1024	0.1	13.141328	6.898573188	86.2322%
16	1024	0.1	6.397168	14.17133535	88.5708%
32	1024	0.1	3.268929	27.73275681	86.6649%
48	1024	0.1	2.372338	38.21395307	79.6124%
64	1024	0.1	2.191589	41.3656087	64.6338%

Processes	Dimension	Precision	Time	Speedup	Efficiency
seq	2048	0.1	433.871217	1	100%
3	2048	0.1	220.559591	1.967138291	65.5713%
4	2048	0.1	114.970783	3.773751954	94.3438%
8	2048	0.1	62.648354	6.92550066	86.5688%
16	2048	0.1	30.087359	14.42038223	90.1274%
32	2048	0.1	14.719159	29.4766309	92.1145%
48	2048	0.1	9.931715	43.68542764	91.0113%
64	2048	0.1	7.761698	55.89900779	87.3422%
seq	512	0.001	311.057349	1	100%
3	512	0.001	156.823862	1.983482265	66.1161%
4	512	0.001	106.923304	2.909163273	72.7291%
8	512	0.001	46.117531	6.74488296	84.3110%
16	512	0.001	22.949032	13.55426882	84.7142%
32	512	0.001	13.136245	23.67932	73.9979%
48	512	0.001	11.384876	27.32197953	56.9208%
64	512	0.001	10.712251	29.03753366	45.3711%
32	64	0.1	0.026814	1.531140449	4.7848%
32	128	0.1	0.035821	5.641327713	17.6291%
32	256	0.1	0.170539	13.76465207	43.0145%
32	512	0.1	0.540164	23.77632719	74.3010%
32	1024	0.1	3.312326	27.38906104	85.5908%
32	2048	0.1	14.552712	29.81984416	93.1870%
32	256	0.1	0.170962	13.84661504	43.2707%
32	256	0.01	1.580953	14.45631274	45.1760%
32	256	0.001	3.739775	13.99652787	43.7391%
32	256	0.0001	5.799107	14.04875078	43.9023%
32	256	0.00001	8.001268	14.01266362	43.7896%
32	256	0.000001	9.983635	14.04976113	43.9055%
seq	64	0.1	0.041056	1	100%
seq	128	0.1	0.202078	1	100%
seq	256	0.1	2.34741	1	100%
seq	512	0.1	12.843116	1	100%
seq	1024	0.1	90.721499	1	100%
seq	2048	0.1	433.959604	1	100%
seq	256	0.1	2.367245	1	100%
seq	256	0.01	22.854751	1	100%
seq	256	0.001	52.343865	1	100%
seq	256	0.0001	81.470209	1	100%
seq	256	0.00001	112.119077	1	100%
seq	256	0.000001	140.267687	1	100%