CM30225 – Parallel Computing: MPI ON A DISTRUBUTED MEMORY ARCHITECTURE

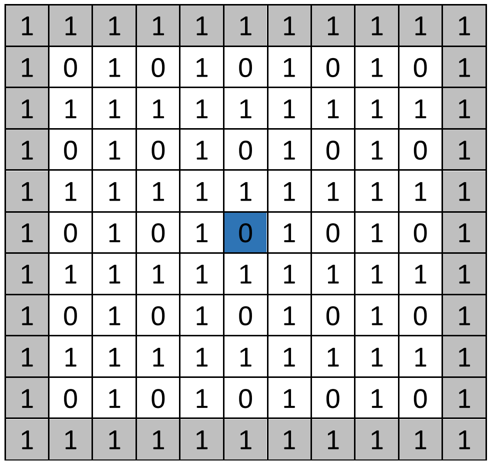
Candidate Number: 10025, *CM30225*, *University of Bath*

Abstract The development of a parallel solution to the relaxation method utilising MPI on a distributed memory architecture. This document will explain the development of the solution, as well as try to identify the final programs efficiency, scalability and shortfalls across varying quantities of data and computation resources. The key points that will be touched on are the messaging overheads required by MPI to communicate between processes; how these overheads can be reduced; and their effect on runtimes and optimality.

I. SAMPLE DATA & DESIGN APPROACH

THE realisation of the solution came from meticulous testing of not only the program, but MPI's inbuilt functions as well; this provided an intricate knowledge of the final builds which aided debugging and optimising.

The sample data used was originally pseudorandom, although this was changed to a fixed uniform pattern early on in testing. This has given exceptional reproducibility between tests and aided the analysis of results with varying matrix sizes. This data set is quite complex as all cells adjacent to a cell containing a zero will always finish the relaxation containing the value 1. All other cells (excluding the border shown in grey) will contain a value which will meet the precision criteria



1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1

Fig. 1. Sample data for 11x11 array

once relaxation is complete. The data set is always forced to have an odd integer dimension so that there is a central cell (shown in blue). The relaxation propagates towards this central cell with it always being the last cell to meet the precision criteria.

The design approach isolated useful features for programming and testing before they were integrated into the main design. Initially the approach was to use `MPI_Bcast` to share data between all processes. This was due to the set of data each process was responsible for not being sequential. It can be seen from Fig. 2 that to perform relaxation, *Thread 2* must acquire three rows of data from both *Thread 1* and *Thread 3*. The amount of data each process needed to acquire by MPI would also increase as the array size got bigger. To reduce the amount of data sent and received between processes, the data was organised into new sub-sections shown in Fig. 3.

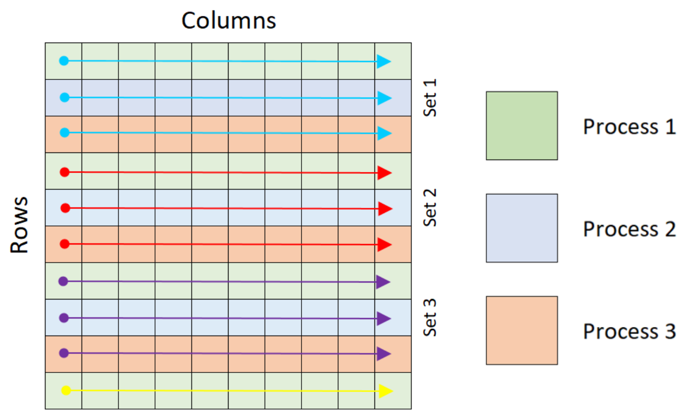


Fig. 2. Initial approach to segmenting data

This new approach to separating data would mean that each process would now only have to send and receive two rows of data using MPI and that the number of shared rows would also always stay the same, regardless of the size of the array.

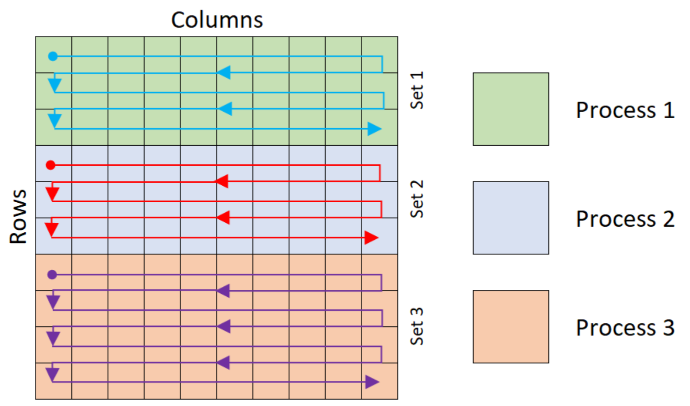


Fig. 3. Final approach to segmenting data

This change to the design was made after a working build using `MPI_Bcast` had been completed, this means that mush of the testing is broadcast related. Once the messaging overhead problems with using `MPI_Bcast` were identified it was replaced with `MPI_Send` and `MPI_Recv`. However, the broadcast function is still used to send local precision flag status' and after relaxation had been completed to send each processes entire chunk of data to all other processes in `MPI_COMM_WORLD`.

II. Software Design & Testing

A. *MPI_BCast*

Firstly, the MPI broadcast function was tested to identify how to send full rows

```
arr[myrank] = myrank;

printf("Processor %d: %d %d %d %d\n", myrank, arr[0], arr[1],
        arr[2], arr[3]);

if ( myrank == 0 )
  printf( "\n After BCAST\n");

for ( i = 0; i < nproc; i++ )
  MPI_Bcast(&arr[i], 1, MPI_INT, i, MPI_COMM_WORLD);
```

Code Extract 1. Sending process ranks between processes.

of data between processes. *Code Extract 1* shows implementing the broadcast function to send process ranks between processes and an extract from the *.out* file is shown in *Console Output 1*.

```
main reports 4 procs
Processor 0: 0 0 0 0
Processor 1: 0 1 0 0
Processor 2: 0 0 2 0
Processor 3: 0 0 0 3

 After BCAST
Processor 3: 0 1 2 3
Processor 0: 0 1 2 3
Processor 1: 0 1 2 3
Processor 2: 0 1 2 3
```

Console Output 1. Sending process ranks between processes.

With a slight modification, the code in *Extract 1* can be used to send a whole row of data, the results of this testing can be seen in *Console Output 2*. This test program populates the rows of a 5x5 array with the rank of the corresponding process, each process then broadcasts the data from its row so that the information is available within any process in `MPI_COMM_WORLD`.

```
INFO: Process 0 reports 3 total processes

INFO: Process 0 current arr:
 0.000000 | 0.000000 | 0.000000 |
 0.000000 | 0.000000 | 0.000000 |
 0.000000 | 0.000000 | 0.000000 |

INFO: Process 1 current arr:
 0.000000 | 0.000000 | 0.000000 |
 1.000000 | 1.000000 | 1.000000 |
 0.000000 | 0.000000 | 0.000000 |

INFO: Process 2 current arr:
 0.000000 | 0.000000 | 0.000000 |
 0.000000 | 0.000000 | 0.000000 |
 2.000000 | 2.000000 | 2.000000 |

INFO: Process 0 arr after BCast:
 0.000000 | 0.000000 | 0.000000 |
 1.000000 | 1.000000 | 1.000000 |
 2.000000 | 2.000000 | 2.000000 |
```

Console Output 2. Sending rows between processes.

B. *Data Import & Resize*

The sample data to be used took the form of a 5000x5000 array in the format shown in Fig. 1. The data is all stored in a single binary file, from which each process imports the full set of data; before then copying a specific chunk into the working arrays. It is important to ensure that the data has been imported correctly, which is why the *data* array is first populated with sentinel values and then checked to ensure the import has been executed correctly.

```
/* Populate malloc arrays with sentinel values */
for ( i = 0; i < dataSize; i++ ) {
  for ( j = 0; j < dataSize; j++ )
    data[i][j] = -1;

/* Import full data set */
importData( data, dataSize );

/* Verify that data was imported to all cells in data array */
if ( verifyArr( data, dataSize ) == 1 ) {
  printf ( "INFO: Process %d on %s successfully imported test
             data\n", myId, coreName );
}
else {
  printf ( "ERR: Process %d on %s failed to import test data\n",
             myId, coreName );
  MPI_Abort( MPI_COMM_WORLD, 5 );
}

int verifyArr( double** arr, int size )
{
  int i, j;
  for ( i = 0; i < size; i++ )
    for ( j = 0; j < size; j++ )
      // Cell contains sentinel value, test failed
      if ( arr[i][j] != 0 || arr[i][j] != 1 )
        return 0;

  // All cells contain non-sentinel values, test success
  return 1;
}
```

Code Extract 2. Parallel data import and verification.

As the sample data consists of only zeros and ones, it is very easy to verify if the data has been imported correctly. The `verifyArr` function ensures that the starting data is identical across all processes and provides a written confirmation in the console. *Console Output 3* shows down-sizing the working arrays to 5x5 across three processes, this output also shows the actual contents of the arrays for further verification that the import and resize has been completed correctly. In normal operation however, there would be no need for this.

```
INFO: Process 0 reports 3 total processes

INFO: Process 0 on node-sw-163 successfully imported test data
INFO: Process 2 on node-sw-163 successfully imported test data
INFO: Process 1 on node-sw-163 successfully imported test data

INFO: Process 0 copied following data to arr:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

INFO: Process 1 copied following data to arr:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

INFO: Process 2 copied following data to arr:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
```

Console Output 3. Parallel data import and resize.

C. *Data Exchange*

As explained in *Section 1*, the method by which each process exchanged information changed roughly halfway through development. Initially, all processes would have all data available to them at the end of a single relaxation cycle; made possible with the `MPI_Bcast` function. This was then updated to use `MPI_Send` and `MPI_Recv` to exchange pairs of data between processes. These would be the rows of information that bordered the chunk of data each process was responsible for editing. (i.e. the first row would be sent to the process ranked before and the last row would be sent to the process ranked after.)

Code Extract 3 contains the code that was used to broadcast information between processes for the block separation shown in Fig. 2.

```
for ( i = 1; i < d - 1; i++ ) {
  MPI_Bcast(*&arr[i], d, MPI_DOUBLE, (i - 1) % nProcs,
            MPI_COMM_WORLD);
}
```

Code Extract 3. Broadcasting information throughout `MPI_COMM_WORLD`.

Code Extract 4 is the updated send and receive solution, which greatly reduces messaging overheads when using many processes. First, all the even numbered processes send their last row of modifiable data to the next ranked process, while all the odd numbered processes wait to receive it. The even numbered processes then send their first row of modifiable data to the preceding ranked process, with the odd numbered processes receiving again. This is then repeated with the odd and even number processes switching roles.

```
for ( i = 0; i < 2; i++ ) {
  /* Sending Information */
  if ( myId % 2 == i) {
      // If process number is not last in MPI_COMM_WORLD
    if ( myId != nProcs - 1 ) {
      // Send last editable row to the next ranked process
      MPI_Send ( *&arr[iEnd - 1], d, MPI_DOUBLE, myId + 1, 0,
                 MPI_COMM_WORLD );
    }
    // If process number is not first in MPI_COMM_WORLD
    if ( myId != 0 ) {
      // Send first editable row to the preceding process
      MPI_Send ( *&arr[iStart], d, MPI_DOUBLE, myId - 1, 1,
                 MPI_COMM_WORLD );
    }
  }
  /* Recieving Information */
  else {
    // If process number is not first in MPI_COMM_WORLD
    if ( myId != 0 ) {
    // Receive row from preceding ranked process
      MPI_Recv ( *&arr[iStart - 1], d, MPI_DOUBLE, myId - 1, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    }
    // If process number is not last in MPI_COMM_WORLD
    if ( myId != nProcs - 1 ) {
      // Receive row from the next ranked process
      MPI_Recv ( *&arr[iEnd], d, MPI_DOUBLE, myId + 1, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    }
  }
}
```

Code Extract 4. MPI Send and Receive replacement for BCast

Both *Code Extract 3* and *Code Extract 4* produce the same results; provided that, when using send and receive, each process broadcasts its chunk information once relaxation is complete. The results from one iteration of relaxation are shown in *Console Output 4*; whereas *Console Output* 5 shows the information available to *Process* 0 had no broadcast occurred.

```
INFO: Process 0 reports 2 total processes

INFO: Process 0 on node-sw-163 successfully imported test data
INFO: Process 1 on node-sw-163 successfully imported test data

INFO: Process 0 imported following data to arr:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

INFO: Process 0 arr after one relax iteration:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 1.000000 | 0.500000 | 1.000000 | 1.000000 |
 1.000000 | 0.500000 | 1.000000 | 0.500000 | 1.000000 |
 1.000000 | 1.000000 | 0.500000 | 1.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
```

Console Output 4. Process 0 data before and after one relax iteration using BCast

In *Console Output 5*, Process 0 has amended the values shown in purple and received the values shown in green from Process 1. Theses green values are required by Process 0 to perform the next iteration of relaxation. The values shown red have been amended by Process 1 but have not been sent to Process 0 to reduce messaging overhead.

```
INFO: Process 0 reports 2 total processes

INFO: Process 0 on node-sw-163 successfully imported test data
INFO: Process 1 on node-sw-163 successfully imported test data

INFO: Process 0 imported following data to arr:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

INFO: Process 0 arr after one relax iteration:
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
 1.000000 | 1.000000 | 0.500000 | 1.000000 | 1.000000 |
 1.000000 | 0.500000 | 1.000000 | 0.500000 | 1.000000 |
 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 |
 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
```

Console Output 5. Process 0 data before and after one relax iteration without using BCast

D. Flags

The last aspect to implement was to enable processes to discern whether they had successfully reached the required level of precision. This could be achieved simply by broadcasting a specific processes precision flag status within an array to the remaining processes. It is then possible for the processes to individually establish whether they are all done or not.

```
for ( i = 0; i < nProcs; i++ )
  MPI_Bcast(&prec_Flags[i], 1, MPI_INT, i, MPI_COMM_WORLD);

printf("Processor %d: %d %d %d %d %d\n", myId, prec_Flags[0],
  prec_Flags[1], prec_Flags[2], prec_Flags[3], prec_Flags[4]);

for ( i = 0; i < nProcs; i++ )  {
  if ( prec_Flags[i] == 1 ) {
    stop = false;
    break;
  }
}

if ( stop == true ) {
  printf ( "INFO: Process %d STOP == TRUE\n", myId );
  break;
}

prec_Flags[myId] = 0;
stop = true;
```

Code Extract 5. Broadcasting local flag status' to `MPI_COMM_WORLD`.

Console Output 6 is a highly reduced printout from the debugging of this section of the code but still clearly shows the flags status' being communicated between processes.

```
INFO: Process 0 reports 5 total processes

INFO: Process 4 on node-sw-163 successfully imported test data
INFO: Process 3 on node-sw-163 successfully imported test data
INFO: Process 1 on node-sw-163 successfully imported test data
INFO: Process 0 on node-sw-163 successfully imported test data
INFO: Process 2 on node-sw-163 successfully imported test data

Processor 0: 1 1 1 1 1
Processor 1: 1 1 1 1 1
Processor 1: 1 1 1 1 1
Processor 1: 1 1 1 1 1

...

Processor 2: 0 1 1 1 0
Processor 2: 0 1 1 1 0
Processor 2: 0 0 1 0 0
Processor 3: 0 1 1 1 0
Processor 3: 0 0 1 0 0
Processor 3: 0 0 0 0 0
Processor 2: 0 0 0 0 0
Processor 4: 0 0 0 0 0
Processor 0: 0 0 0 0 0
Processor 1: 0 0 0 0 0

INFO: Process 3 STOP == TRUE
INFO: Process 4 STOP == TRUE
INFO: Process 1 STOP == TRUE
INFO: Process 0 STOP == TRUE
INFO: Process 2 STOP == TRUE
```

Code Extract 6. Testing flag operation.

III. Results

A. *Array Size: 901*

To gain meaningful results for speedup and efficiency, an array of 901 x 901 had was used as it was one of the largest arrays that could be solved by a single process within the 15-minute time limit.

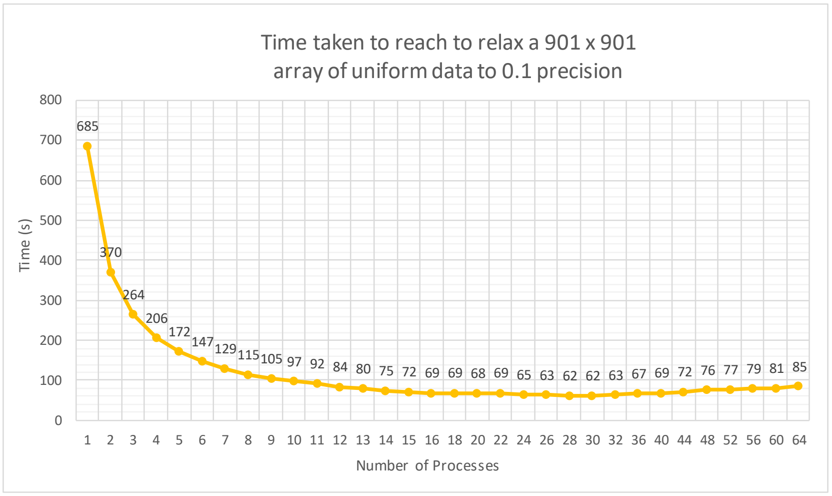


Fig. 4. Relaxation benchmarking

The graph in Fig. 4 shows that increasing the number of processes continues to shorten the runtime up until roughly 28 processes, at which point the messaging overhead appears to be greater than the benefit of increasing processing power. The speedup and efficiency graphs for this data are shown in Fig. 5 and Fig. 6.

In Fig. 5, there is a slight shoulder present between 16 and 22 processes, this could be explained by the fact that after 16 processes another core is required. This means that data can no longer be sent and received exclusively within the CPUs cache memory and the program must resort to slower forms of data transfer. After this slump, relative speedup continues to gradually increase, eventually peaking somewhere between 28 and 30 processes to agree with the time plot in Fig. 4.

Fig. 6 shows no signs of any untoward data as efficiency generally decreases as the number of processes is increased. After the speedup peak at 30 processes, the curve of the fall in efficiency seems to smooth out (i.e. more predictable), implying that the addition of one processor will see the same relative decrease in efficiency. In fact, for the last 16 processes, the addition of every 4 see's a 10% decrease in efficiency.

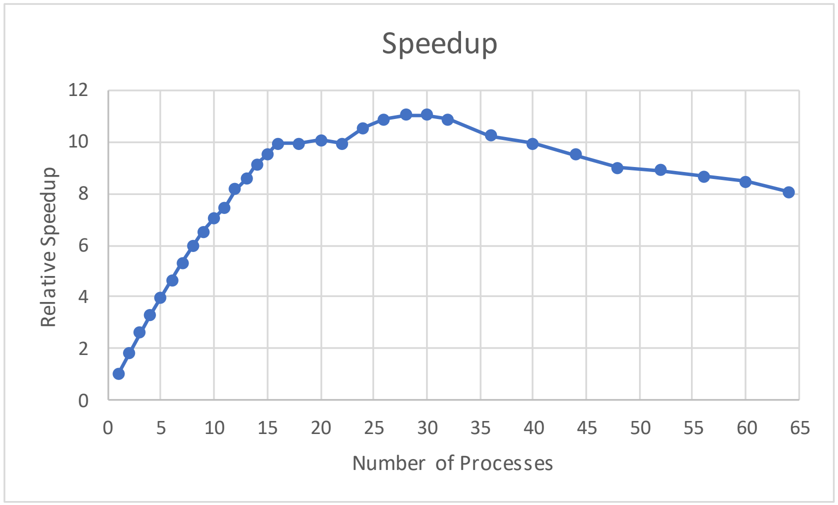


Fig. 5. Relative speedup when performing relaxation on 901 x 901 array of uniform data

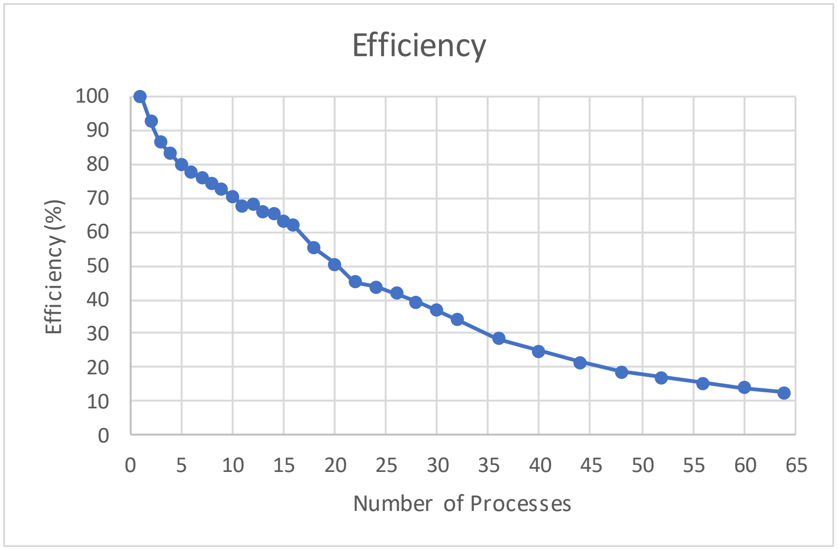


Fig. 6. Efficiency when performing relaxation on 901 x 901 array of uniform data

B. *Comparing Across Array Sizes*

When performing relaxation on larger arrays, it is possible to see from Fig. 7 that speedup continues to occur for greater numbers of processes. Fig. 7 also shows that both Amadahl's and Gustafson's laws are both agreed with by this data.

With *Data Size 901*, adding more than roughly 18 processes to the task doesn't see any significant reduction in the time taken (it is in fact detrimental after a certain number of processes). This is because all the parallel aspects of the program are being completed in optimal time, where as the sequential parts taken the residual amount of time; agreeing with Amdahl's law.

When increasing the array size, each process has a larger workload to complete, so the parallel part of the program becomes more time consuming. This means that when new processes are added to the work load the speedup is true to Gustafson's law as there is sufficient work for all processes to be busy all of the time. The curve of each of the graphs is the transition between Amadahl's and Gustafson's when the workload ceases to be enough for Gustafson's law to apply

I would suggest that a more linear speedup would be present for a lower number of processes if it were possible to plot time taken to solve a 2001 x 2001 using 1-64 processes.

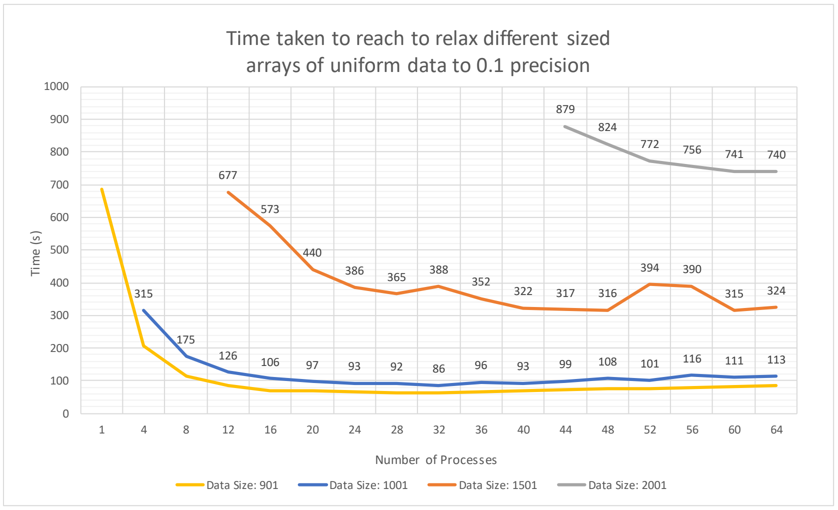


Fig. 7. Time taken to perform relaxation on varying sizes of arrays

IV. Scalability Conclusions

From the data collected, the final solution shows high potential for scalability. Increasing the number of processors shows and decrease in efficiency, where as a larger problem size shows an increase in efficiency. This means that if number of processes and problem size are increased simultaneously, system efficiency is maintained. Fig. 8 shows that efficiency is ~60% when using 12 processes on a 501 x 501 array, but is also ~60% when using 16 processors on a 901 x 901 array.

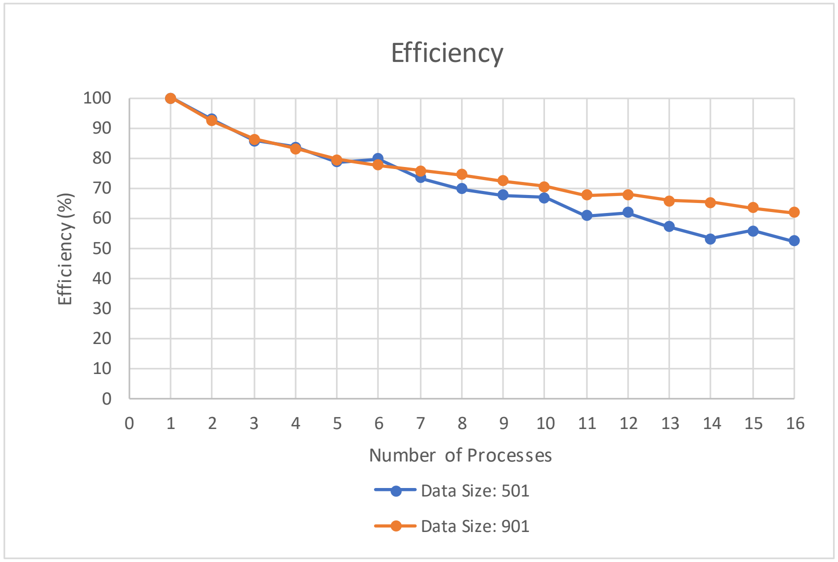


Fig. 8. Conservation of efficiency when increasing both problem size and processes.