Delft University of Technology

Introduction To High Performance Computing
IN4049

Practical Lab Assignment

Author:
Aditya Shankar (5454360) - a.shankar-3@student.tudelft.nl

January 28, 2022

Part I

Poisson Solver

1 Building a parallel program for the Poisson Solver using MPI

This section explains the step-by-step code modifications made to convert the sequential program into a parallel one. Answers are provided for selected questions where necessary. The code can be found on GitHub

Step 1

Modification: The lines for MPI_Finalize( ) and MPI_Init( ) were added.
Test: When starting multiple processes, the result can be verified by counting the print statements on the console.

Step 2

Modification: As mentioned in the Reader.
Test: Each process prints out its rank and the output this time. The numbers vary from zero to P - 1, where P is the number of processors given as an argument to Prun.

Step 3

Modification: As given in the Reader.
Test: The execution times were printed out for each process. One additional remark is that the percentage of Elapsed time was sometimes larger than 100% of CPU time when printing. This may be because of other overheads when each process interacts with the CPU.

Step 4

Modification: As mentioned in the Reader.
Test: Multiple Output files are created as expected. A comparison was made using the diff command to ensure that results were the same.

Step 5

Modification: The If-statement was checked by passing:

```
if(proc_rank == 0)
```

wherever necessary. The broadcast statements were modified as shown below by using process 0 as the sender.

```
MPI_Bcast(gridsize,2,MPI_INT,0,grid_comm);
MPI_Bcast(&precision_goal,1,MPI_DOUBLE,0,grid_comm);
MPI_Bcast(&max_iter,1,MPI_INT,0,grid_comm);
```

Step 6

The communication size was obtained by passing the address of variable P as the second argument to MPI_Comm_Size. Additional modifications were made as shown below for creating a Cartesian grid of processors:

```
MPI_Cart_create(MPI_COMM_WORLD,2,P_grid,periods,reorder,&grid_comm);
MPI_Comm_rank(grid_comm,&proc_rank);
MPI_Cart_coords(grid_comm,proc_rank,2,proc_coord);
printf("(%i) (x,y) = (%i,%i)\n",proc_rank,proc_coord[X_DIR],proc_coord[Y_DIR]);
local_parity = (proc_coord[X_DIR] + proc_coord[Y_DIR])%2;
MPI_Cart_shift(grid_comm,Y_DIR,1,&proc_bottom,&proc_top);
MPI_Cart_shift(grid_comm,X_DIR,1,&proc_left,&proc_right);
```

Step 7

Modifications were made as mentioned in the Reader. When the code was executed with three processes, two were found to stop execution after one iteration. After debugging, this was found to be the case because all the phi values within the local subgrid were set to zero. On further inspection of input.dat, it was found that only specific coordinates in the entire process grid got a non-zero source value. If the local subgrid coordinates happened to be such that none of the points received this, then execution for that process would stop after one iteration because the precision goal has been reached. Input.dat indicates that only the coordinates (35,70), (62,75), and (37,25) receive a non-zero source value. All these points lie within the local grid of process 1 alone (when executed using three processes); hence processes 0 and 2 stop after one iteration. Furthermore, since border exchange has not been implemented yet, the non-zero values do not propagate to the local grids of processes 0 and 2 from process 1.

Step 8

The snippet below shows the modifications made to implement border exchanges:

```
 void Exchange_Borders(){
Debug("Exchange_Borders",0);
//for pushing data up and receiving from below
MPI_Sendrecv(&phi[1][dim[Y_DIR]-2],1,border_type[Y_DIR],proc_top,0,&phi[1][0],1
,border_type[Y_DIR],proc_bottom,0,grid_comm,&status);

//for pushing data down and receiving from above
MPI_Sendrecv(&phi[1][1],1,border_type[Y_DIR],proc_bottom,0,&phi[1][dim[Y_DIR]-1],1
,border_type[Y_DIR],proc_top,0,grid_comm,&status);

//for pushing data left and receiving from the right
MPI_Sendrecv(&phi[1][1],1,border_type[X_DIR],proc_left,0,&phi[dim[X_DIR]-1][1],1
,border_type[X_DIR],proc_right,0,grid_comm,&status);

//for pushing data right and receiving from the left
MPI_Sendrecv(&phi[dim[X_DIR]-2][1],1,border_type[X_DIR],proc_right,0,&phi[0][1],1
,border_type[X_DIR],proc_left,0,grid_comm,&status);


}
```

For every local subgrid, the points to be communicated do not include index 0 and the last index, i.e., DIM-1. This is because these are the ghost points received from the neighbouring processes. Therefore, the actual points to be communicated lie on indices 1 and DIM-2 because these are the "true" boundary points. Figure 1 illustrates this.

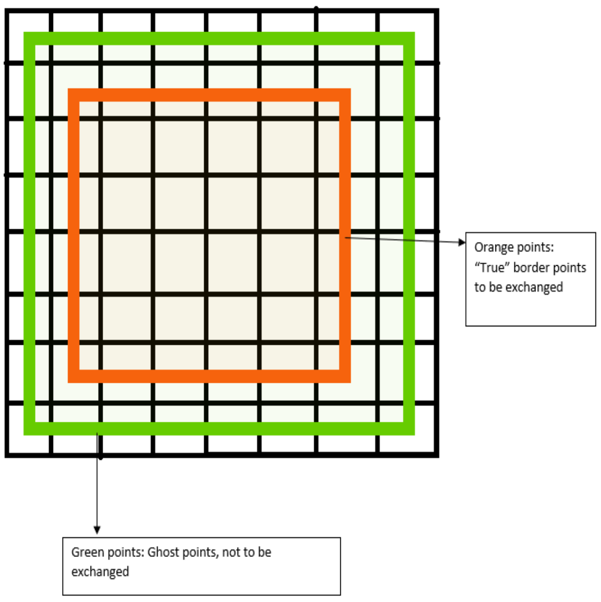

Orange points: "True" border points to be exchanged
Green points: Ghost points, not to be exchanged

Figure 1: Actual border points to be exchanged

Step 9

The snippet below shows how a global convergence criterion can be implemented:

```
global_delta = 2 * precision_goal;
if(use_precision_goal == 1){
  while (global_delta > precision_goal && count < max_iter)
  {
    Debug("Do_Step 0", 0);
    for(int i = 0;i<sweep_count;i++){
      delta1 = Do_Step(0); //perform multiple sweeps
    }
    Exchange_Borders();

    Debug("Do_Step 1", 0);
    for(int i = 0;i<sweep_count;i++){
      delta2 = Do_Step(1);//perform multiple sweeps
    }
    Exchange_Borders();

    delta = max(delta1, delta2);

    MPI_Allreduce(&delta,&global_delta,1,MPI_DOUBLE,MPI_MAX,grid_comm);
    if(count%100 == 0 && proc_rank == 0){
      printf("step number: %i, error: %f\n",count,global_delta);
    }
    count++;
  }
```

Since the criterion involves a global delta, the number of iterations to converge increases. The effective grid size is not limited to the local subgrid.

Step 10

- To ensure that processes write the correct coordinates to the output, it is necessary to add the offset to the local coordinates: This is shown below:

```
    Debug("Write_Grid", 0);
```

```
for (x = 1; x < dim[X_DIR] - 1; x++)
  for (y = 1; y < dim[Y_DIR] - 1; y++)
    fprintf(f, "%i %i %f\n", x + offset[X_DIR], y + offset[Y_DIR], phi[x][y]);
fclose(f);
```

- A similar change was made to ensure that processes use the global parity of points:

```
if ((x+y+offset[X_DIR]+offset[Y_DIR])%2 == parity && source[x][y] != 1)
```

2 Experiments on the Poisson solver

This section discusses the optimizations and the experimental results by modifying the code written in the previous exercise.

2.1

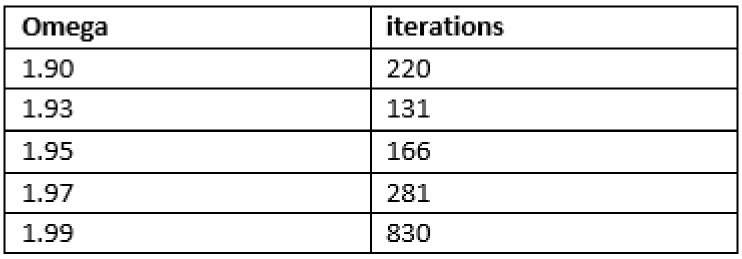
The snippet below shows the modification to the code to implement Successive Over Relaxation (SOR):

```
old_phi = phi[x][y];
double change = ((phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) * 0.25)
phi[x][y] = old_phi + omega*change;
```

2.2

The table 1 below shows the experimental results for five different values of omega and the corresponding number of iterations. Results indicate that an ω of 1.93 gives the least number of iterations, 131 in this case.

Table 1: Iterations for convergence with different ω

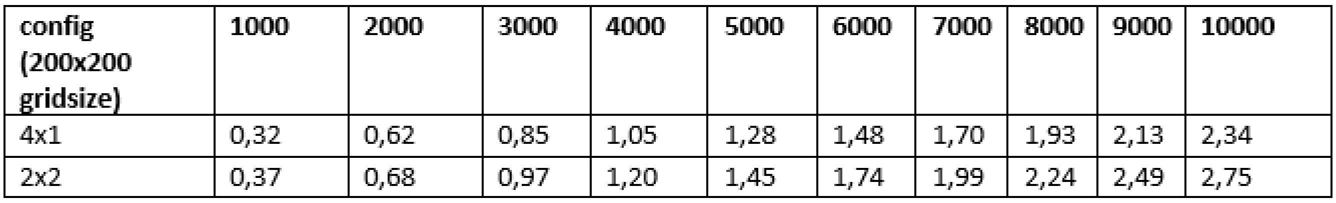


Omega	iterations
1.90	220
1.93	131
1.95	166
1.97	281
1.99	830

2.3

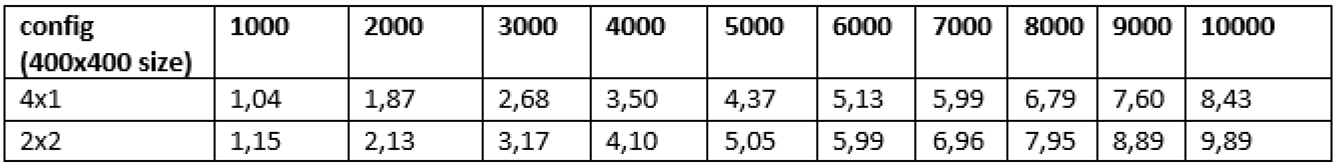
The three tables below [2,3,4] indicate the execution time (seconds) for different iterations, configurations, and grid sizes. (Comma is used to indicate the decimal point). A large number of iterations was chosen, i.e., starting from 1000 onwards, because the execution time is not modelled well by a linear model otherwise. Plotting the table values gives us the figures [2a,2b,2c]. Observe that the 2x2 execution time is greater than the 4x1 configuration in all three scenarios. The corresponding values of alpha and beta computed for the three sizes are shown in table 5 for the different configurations. These were obtained using a least-squares linear fit to the data. Observe that the alpha value does not consistently increase or decrease with the grid size, but beta increases with the size. Furthermore, the 2x2 configuration has a greater beta value than the 4x1 scenario; however, the alpha value does not show a clear pattern.

Table 2: Execution time for various configurations in 200x200 grid



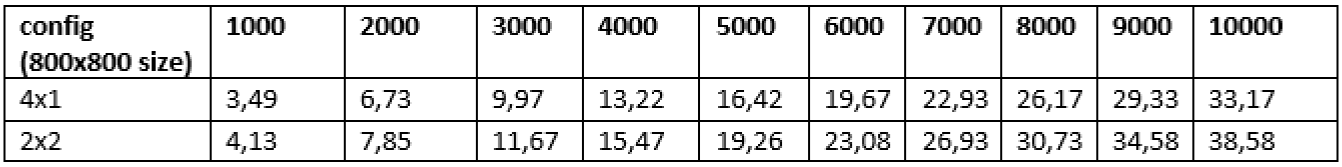
config (200x200 gridsize)	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
4x1	0,32	0,62	0,85	1,05	1,28	1,48	1,70	1,93	2,13	2,34
2x2	0,37	0,68	0,97	1,20	1,45	1,74	1,99	2,24	2,49	2,75

Table 3: Execution time for various configurations in 400x400 grid

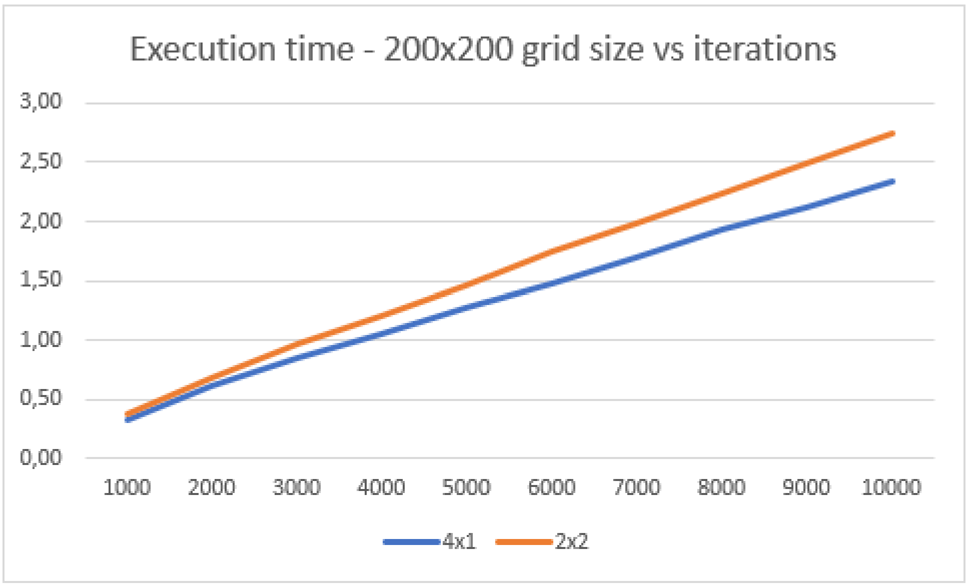


config (400x400 size)	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
4x1	1,04	1,87	2,68	3,50	4,37	5,13	5,99	6,79	7,60	8,43
2x2	1,15	2,13	3,17	4,10	5,05	5,99	6,96	7,95	8,89	9,89

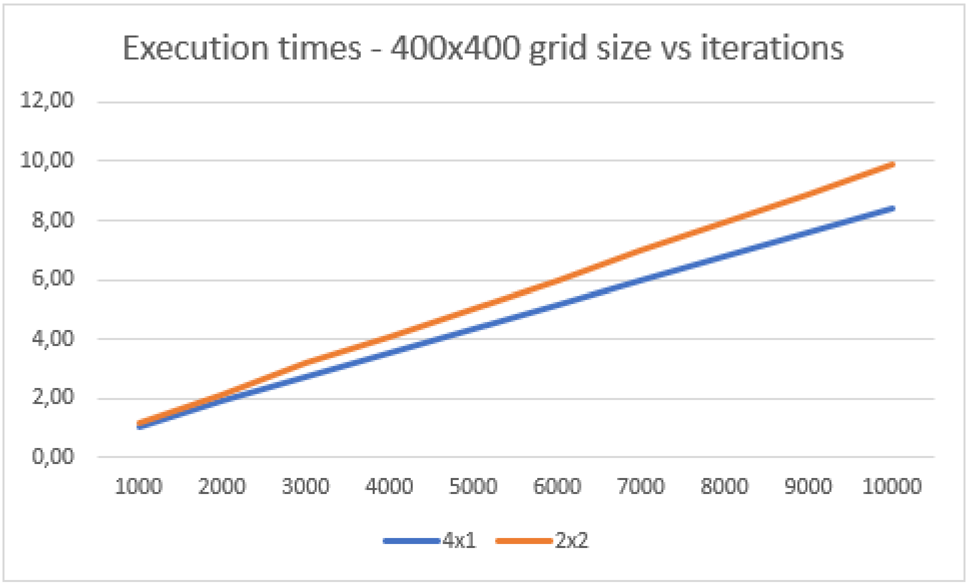
Table 4: Execution time for various configurations in 800x800 grid



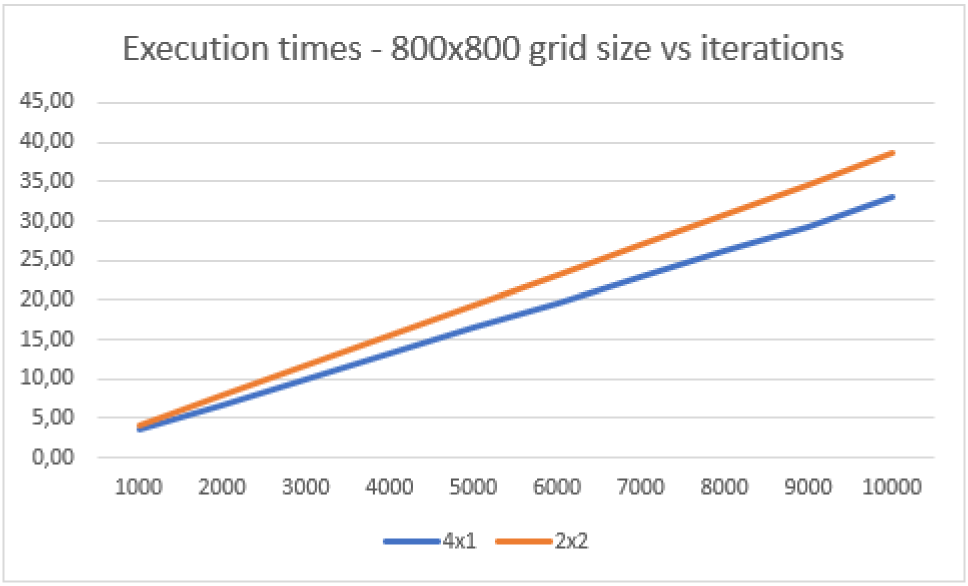
config (800x800 size)	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**	**10000**
4x1	3,49	6,73	9,97	13,22	16,42	19,67	22,93	26,17	29,33	33,17
2x2	4,13	7,85	11,67	15,47	19,26	23,08	26,93	30,73	34,58	38,58



(a) Execution times 200x200 grid



(b) Execution times 400x400 grid



(c) Execution times 800x800 grid

Table 5: alpha and beta

size	4x1 (alpha)	4x1 beta	2x2 (alpha)	2x2 beta
200x200	0,158871	0,000220	0,150738	0,000261
400x400	0,229500	0,000820	0,213645	0,000966
800x800	0,150790	0,003265	0,206850	0,003822

2.4

We can use the result from the previous question to limit the number of experiments for this question. We observed that the grid size's execution times increased, so we can limit ourselves to comparing the different configurations for just one particular size, say 200x200. Furthermore, a 1x16 structure is symmetric to a 16x1 configuration, so it is not necessary to experiment for each of them separately. Similar assumptions hold for 8x2 and 2x8. Therefore, using a similar setup to the previous question, we get the following values of alpha and beta (table 6 for different configurations. For a large number of iterations, n, the beta value plays a higher role in computing the execution time, because the time is estimated using the formula: $\alpha + \beta n$. Since 16x1 has the smallest beta value, it is also expected to have the lowest execution time. Figure 3 indicates this is indeed the case. So, choosing a 16x1 or 1x16 configuration is the best choice. Note the similarity with the result of the previous question. Even there, the square design of 2x2 was performing worse than the 4x1 configuration. This may be because a more "square" configuration has to communicate with more neighbours. In the present case, a processor has to communicate with 2, 3, and 4 neighbours in the worst case for the 16x1, 8x2, and 4x4 configurations.

Table 6: alpha and beta

size = 200x200	16x1	8x2	4x4
alpha	0,069573	0,076220	0,086300
beta	0,00010039	0,000104	0,000106

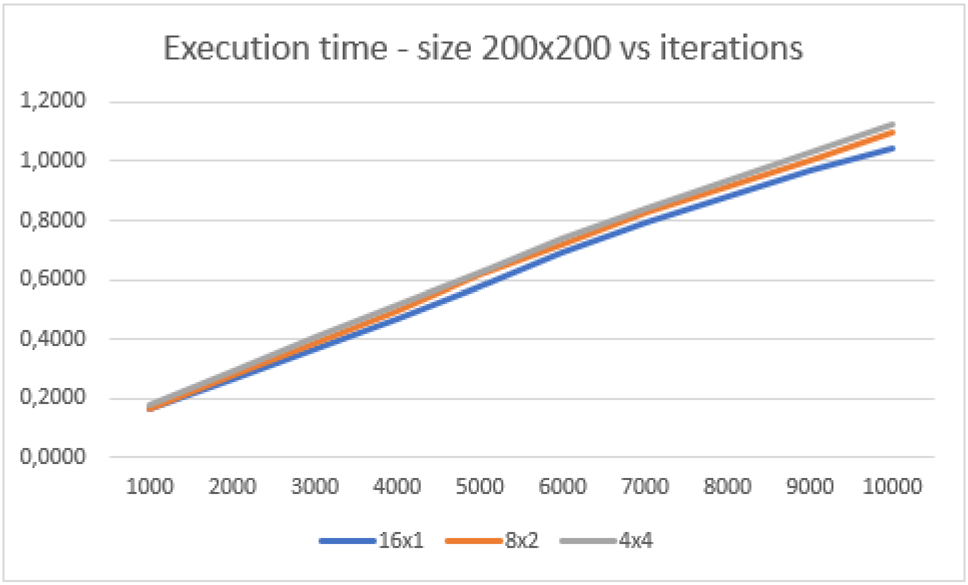
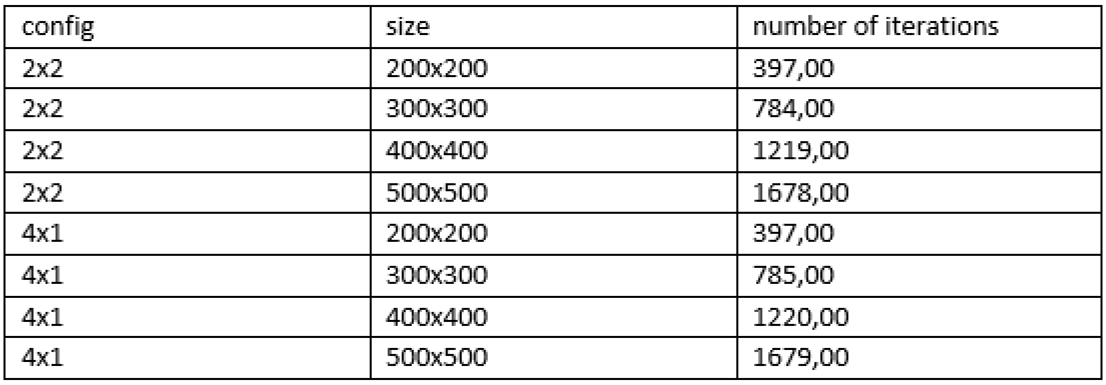


Figure 3: Execution times 800x800 grid

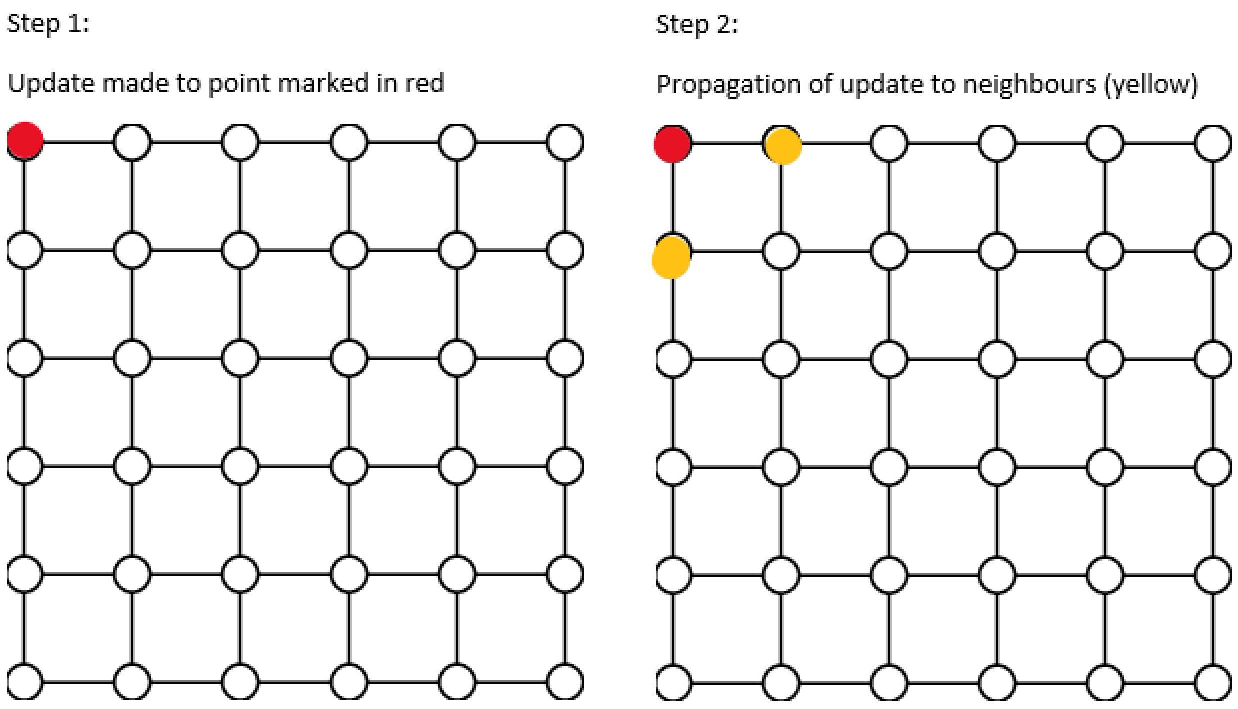
2.5

Table 7 shows the number of iterations for convergence for the 2x2 configuration and 4x1 configuration of processors using ω of 1.95. For both the 4x1 and the 2x2 design, there is a clear positive correlation between the grid size and the number of iterations for convergence. To understand why this is the case, let us consider the number of steps it takes for an update to propagate across the grid when information is updated at a point. Refer to the figures 4a, 4b and 5 shown below, which indicate that the time to propagate the update to all the grid points varies linearly with the grid size.

Table 7: Iterations for convergence for different sizes and configurations



config	size	number of iterations
2x2	200x200	397,00
2x2	300x300	784,00
2x2	400x400	1219,00
2x2	500x500	1678,00
4x1	200x200	397,00
4x1	300x300	785,00
4x1	400x400	1220,00
4x1	500x500	1679,00


Step 1:
Update made to point marked in red
Step 2:
Propagation of update to neighbours (yellow)

(a) (b)

Figure 4: Propagation of update over timesteps 1 and 2

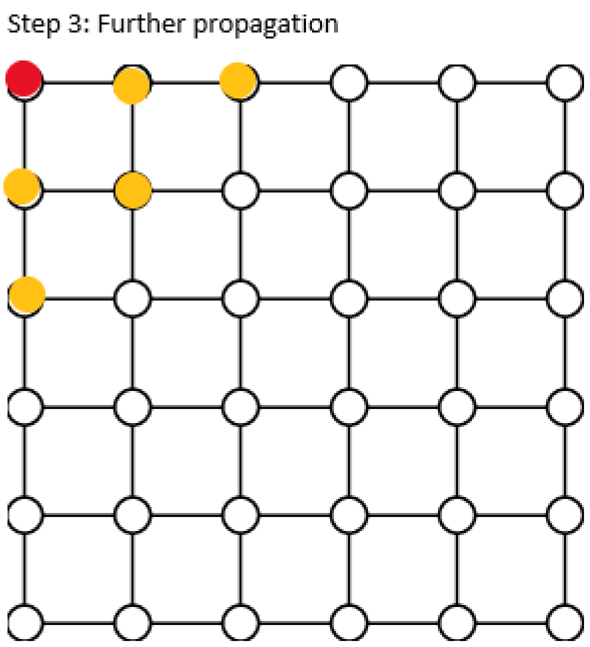
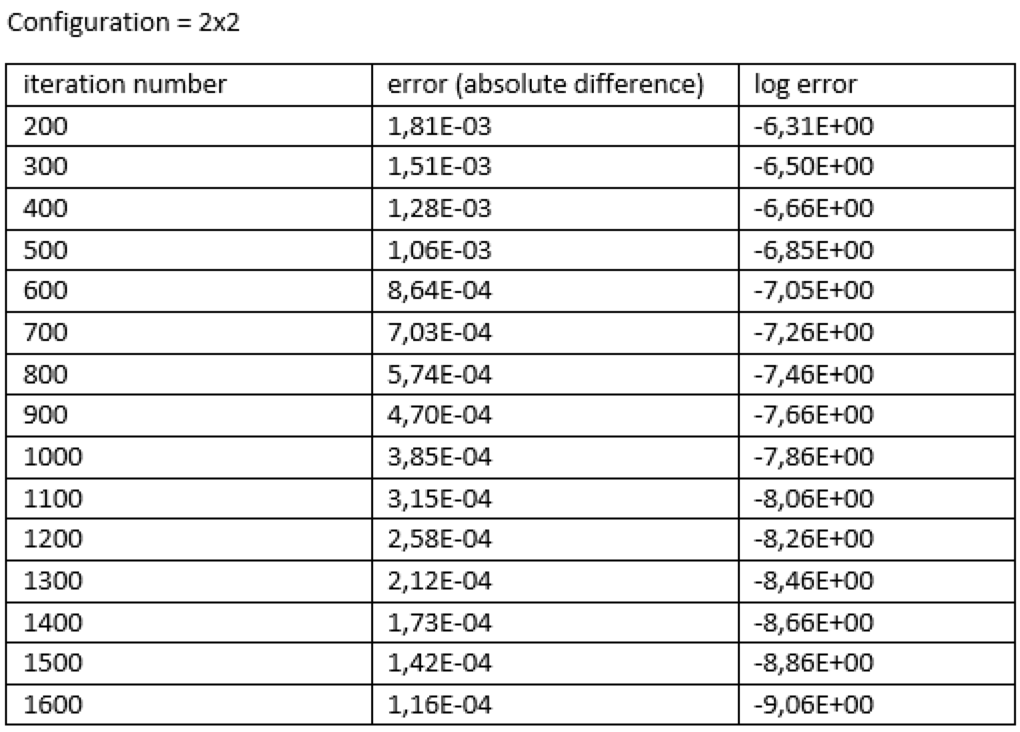


Figure 5: Propagation update timestep 3

2.6

Tables 8 and 9 show the absolute error and the natural logarithm of the error versus the number of iterations for different configurations and a grid size of 500x500. ω was set to 1.95. Both figures 6a and 6b indicate that the logarithm of the error varies linearly with the number of iterations. Therefore, the error must exponentially decay as the iterations increase.

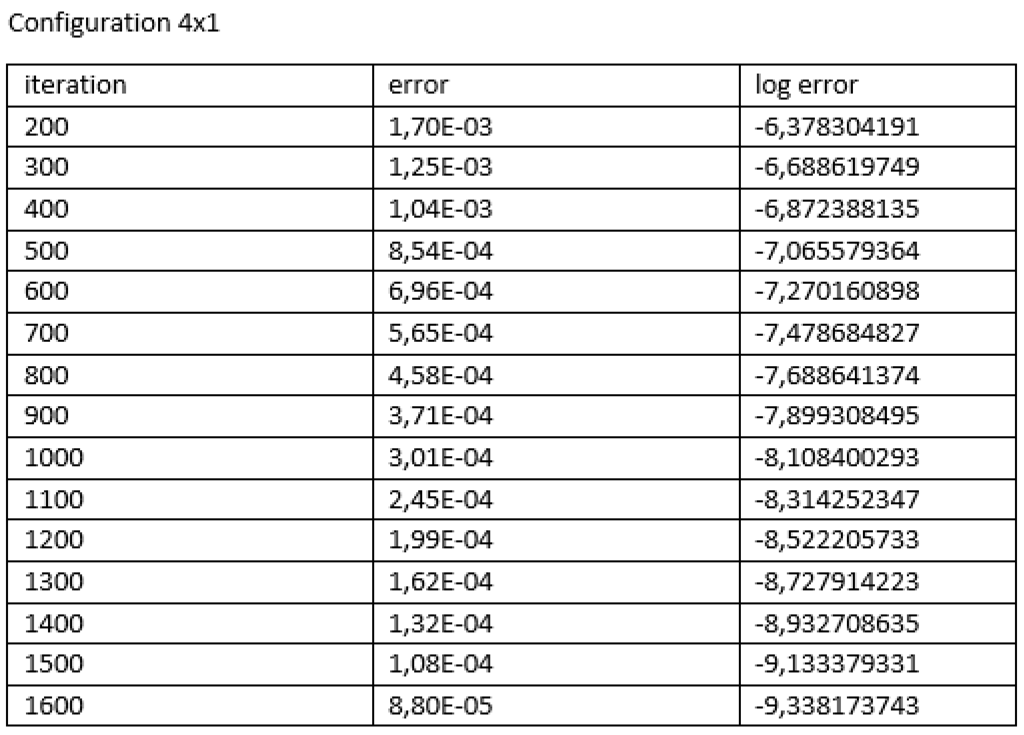
Table 8: Error and its logarithm for a 2x2 configuration



Configuration = 2x2

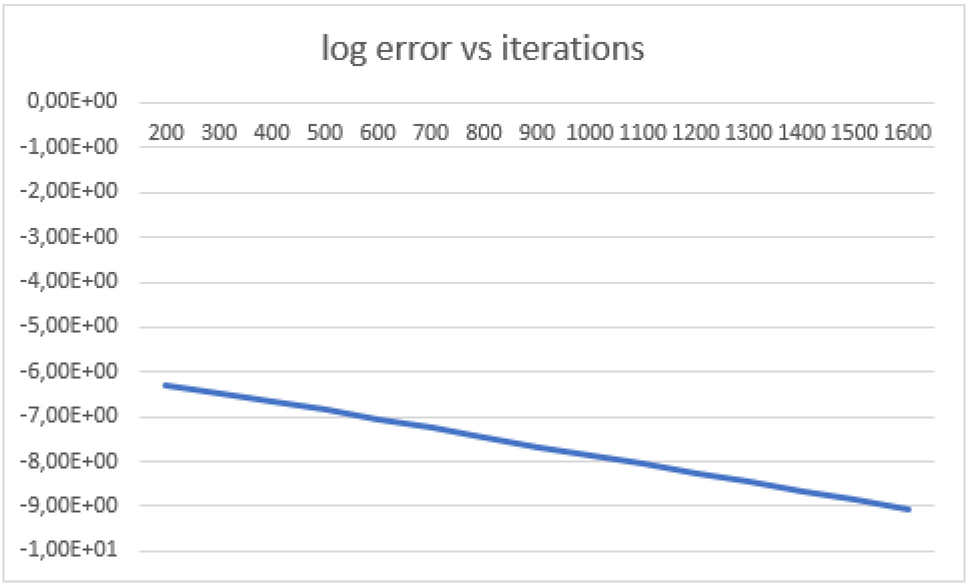
iteration number	error (absolute difference)	log error
200	1,81E-03	-6,31E+00
300	1,51E-03	-6,50E+00
400	1,28E-03	-6,66E+00
500	1,06E-03	-6,85E+00
600	8,64E-04	-7,05E+00
700	7,03E-04	-7,26E+00
800	5,74E-04	-7,46E+00
900	4,70E-04	-7,66E+00
1000	3,85E-04	-7,86E+00
1100	3,15E-04	-8,06E+00
1200	2,58E-04	-8,26E+00
1300	2,12E-04	-8,46E+00
1400	1,73E-04	-8,66E+00
1500	1,42E-04	-8,86E+00
1600	1,16E-04	-9,06E+00

Table 9: Error and its logarithm for a 4x1 configuration

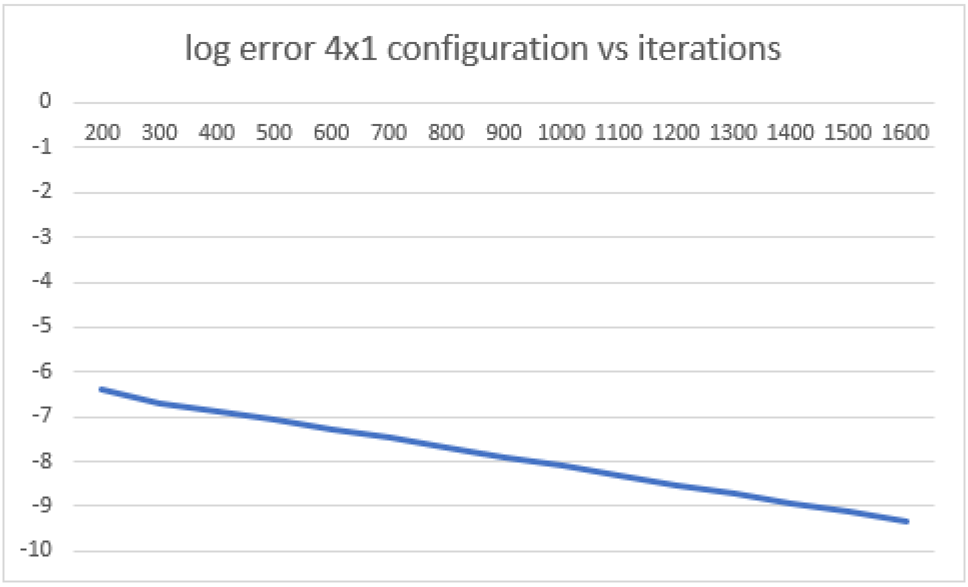


Configuration 4x1

iteration	error	log error
200	1,70E-03	-6,378304191
300	1,25E-03	-6,688619749
400	1,04E-03	-6,872388135
500	8,54E-04	-7,065579364
600	6,96E-04	-7,270160898
700	5,65E-04	-7,478684827
800	4,58E-04	-7,688641374
900	3,71E-04	-7,899308495
1000	3,01E-04	-8,108400293
1100	2,45E-04	-8,314252347
1200	1,99E-04	-8,522205733
1300	1,62E-04	-8,727914223
1400	1,32E-04	-8,932708635
1500	1,08E-04	-9,133379331
1600	8,80E-05	-9,338173743



(a)



(b)

Figure 6: Logarithm of error for two configurations

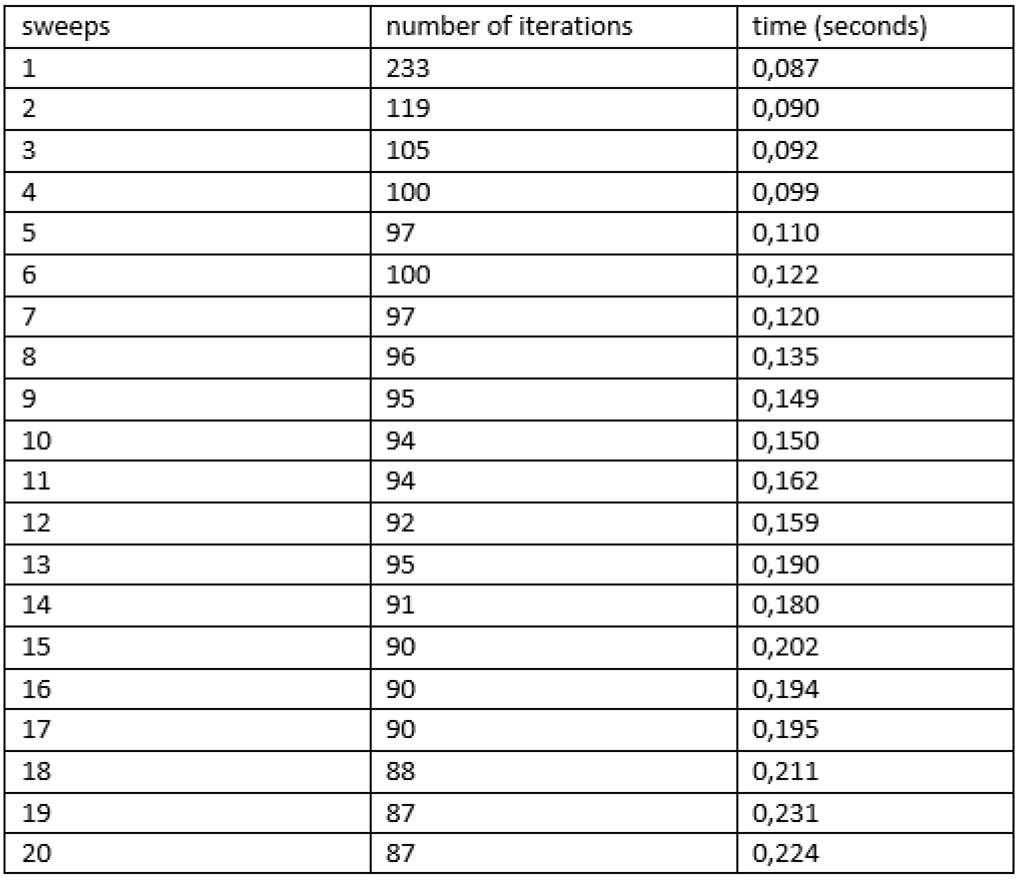
2.8

Multiple sweeps can be implemented by running a for-loop for each Do-Step. The snippet is shown below:

```
Debug("Do_Step 0", 0);
for(int i = 0;i<sweep_count;i++){
  delta1 = Do_Step(0); //perform multiple sweeps
}
Exchange_Borders();
//Exchange_Borders_Half_data(0);
Debug("Do_Step 1", 0);
for(int i = 0;i<sweep_count;i++){
  delta2 = Do_Step(1);//perform multiple sweeps
}
Exchange_Borders();
```

Table 10 shows the number of iterations for convergence and the time for the different number of sweeps. The chosen configuration was 4x4 with $\omega = 1.95$ and a grid size of 200x200. Figure 7 shows that initially the number

Table 10: Iterations for convergence for different sweeps



sweeps	number of iterations	time (seconds)
1	233	0,087
2	119	0,090
3	105	0,092
4	100	0,099
5	97	0,110
6	100	0,122
7	97	0,120
8	96	0,135
9	95	0,149
10	94	0,150
11	94	0,162
12	92	0,159
13	95	0,190
14	91	0,180
15	90	0,202
16	90	0,194
17	90	0,195
18	88	0,211
19	87	0,231
20	87	0,224

of iterations required for convergence drops rapidly, and then it slows down. This may be because increasing the number of sweeps between communications leads to re-computation using stale data and does not propagate the updated values across the grid. However, we observe that the time for convergence increases as we increase the number of sweeps. This is a result of the increasing number of computations without much improvement in terms of convergence.

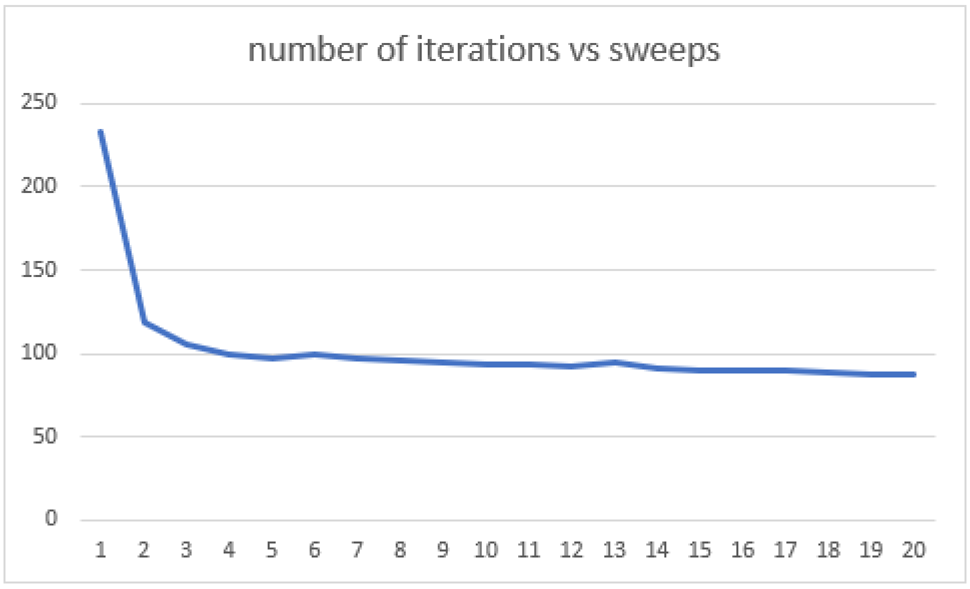


Figure 7

2.9

The problem with the naive if-implementation in the Do_step is that the parity condition is checked for all x and y after entering the for-loop. Hence this can be improved by incrementing the index of y by two instead of 1 because the parity alternates. Furthermore, for every alternate value of x, i.e., the row, the starting y index for that row must be shifted by 1 or zero depending on the parity of the first element. This has been done by using an additional variable called "skip" that takes a value of zero or one depending on the parity of the first element in the local grid. The code snippet for this modification is shown below:

```
int skip;
if((offset[X_DIR] + offset[Y_DIR])%2 == parity){
  skip = 1;
}
else{
  skip = 0;
}
for (x = 1; x < dim[X_DIR] - 1; x++){
  for (y = 1+(skip+x)%2; y < dim[Y_DIR] - 1; y = y + 2){
    if (source[x][y] != 1)
    {
              old_phi = phi[x][y];
       double change = ((phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1])
              phi[x][y] = old_phi + omega*change;
              if (max_err < fabs(old_phi - phi[x][y])){
                  max_err = fabs(old_phi - phi[x][y]);
       }
    }
  }
}
```

Table 11 shows the execution time by avoiding the check and after the modification to the condition for a fixed number of iterations (5000). As expected, the check-avoiding version has a lower execution time. The performance improvement becomes more significant as the grid size increases: this is also likely because the time spent in the loop increases with the size.

Table 11: Execution time with and without loop-avoiding modification

config (omega = 1.95)	size	iterations	avoiding check	with parity check
2x2	200x200	5000,00	1,04	1,110
2x2	400x400	5000,00	3,03	3,680
2x2	800x800	5000,00	11,10	13,660
4x1	200x200	5000,00	0,96	1,090
4x1	400x400	5000,00	2,99	3,650
4x1	800x800	5000,00	10,99	13,460

2.11

The time spent in communication is the sum of overhead time per transfer and the actual transfer time. For uniformity, the 4x1 configuration has been chosen for differing grid sizes and ten trials (by maintaining a counter variable). Results are shown in table 12. In *exchange_ borders ()*, the only thing required to be measured is the elapsed time for the x-y direction along with the size of the data transferred. This is done by timing the send-recv for either the left-to-right or right-to-left communication since both are equal by symmetry arguments. Additional timer functions were created for this purpose, and the size of the data transferred was also calculated.

Table 12: total transfer time for 10000 transfers and different sizes per transfer (depends on the grid size)

config = 4x1			
grid size	no. transfers	total time (seconds)	size (bytes)
200x200	10000	0,048	1600
400x400	10000	0,056	3200
600x600	10000	0,062	4800
800x800	10000	0,072	6400

Using a linear model to estimate the total time, we get the following relation: $T = N * (Ov + S/Bw)$, where T is the total transfer time, N is the number of transfers, Ov is the overhead per-transfer, S is the size communicated per-transfer, and Bw is the bandwidth (bytes-per-second). On substituting a linear fit to the data the obtained values are: $Ov = 4$ microseconds, $Bw = 2.051$ GB/sec.

2.12

A modified function was implemented to reduce the number of points to communicate. (view code on GitHub). The code keeps track of the size of data to transfer and receive for each of the four directions, taking into account the parity of the first element to be shared. If the first element has a parity different from that to be sent, the indices to be transferred are shifted ahead by 1. Checks have also been made to ensure that the sum of the number of points received along a border and the points sent sum up to the row/column dimension. This function takes the parity as input to know what points are to be transferred (red/black).

Four processors were tested in the 4x1 and 2x2 configurations to test the performance, and the total execution time was noted. Omega was chosen as 1.95. The earlier communication version and the modification were tested for 10000 iterations for different grid sizes. Comparing the two results (see table 13, we see that the reduction in communication has reduced the execution time by almost up to 43 percent in the best-case scenario. This is a significant improvement, so it is definitely worth the additional programming complexity.

Part II
Parallel Finite Element Problem

Important code snippets have been provided for the necessary questions. For the complete code, see GitHub

Table 13: Older vs. modified border communication method

config (200x200 size)	transferring red and black	transferring only required	percent reduction
4x1	2,34	1,69	27,78
2x2	2,75	1,77	35,64

config (400x400 size)	transferring red and black	transferring only required	percent reduction
4x1	8,43	5,78	31,44
2x2	9,89	5,92	40,14

config (800x800 size)	transferring red and black	transferring only required	percent reduction
4x1	33,17	21,81	34,24
2x2	38,58	22,16	42,56

4.1 Modification to enable communications

```
void Exchange_Borders(double *vect)
{
  for(int i = 0;i < N_neighb;i++){
    MPI_Sendrecv(vect,1,send_type[i],proc_neighb[i],0,vect,1,recv_type[i],proc_neighb[i]
    ,0, grid_comm, &status);
  }
  // Please finish this part to realize the purpose of data communication among
  neighboring processors. (Tip: the function "MPI_Sendrecv" needs to be used here.)
}
```

4.2

Table 14 shows the time split (seconds) between computations, communications with neighbours, global communication, and the idle time for a 1x4 and a 2x2 configuration. The number of iterations was kept fixed at 5000 for different grid sizes of 100, 200, and 400. The times were obtained by monitoring the specific sections of code using separate timers.

Table 14: Time-splits for different operations with 4 processors

config	size	Tot. time	compute	neighbour	global comm	idle
1x4	100x100	1,268	1,100	0,038	0,095	0,036
1x4	200x200	3,362	3,123	0,046	0,120	0,073
1x4	400x400	12,770	12,251	0,064	0,278	0,177
2x2	100x100	1,285	1,080	0,067	0,101	0,037
2x2	200x200	3,380	3,096	0,082	0,135	0,067
2x2	400x400	12,830	12,220	0,124	0,296	0,190

4.3

When an NxN grid is partitioned among P processors, each processor gets N^2/P points. Figure 8 shows the connections between the points along the border between processors. Though this figure shows only 2 points

along the lattice, it is understood that there are, in general, $N/\sqrt{P}$ points along the border. Let us consider the number of points each of these processors would send to its neighbours (approximately, i.e., not counting the corner cases for points along the edges of a processor):
Processor 1: $2N/\sqrt{P}$ points are sent to 2 and 4
Processor 2: $2N/\sqrt{P}$ points sent to processors 1,3, and 5. One point sent to processor 4.
Processor 3: $2N/\sqrt{P}$ points sent to processors 2 and 6. One sent to processor 5.
Processor 4: $2N/\sqrt{P}$ points sent to 1, 5, and 7.
Processor 5: $2N/\sqrt{P}$ points sent to 2, 4, 6, and 8. One point sent to 3 and 7 each.
Processor 6: $2N/\sqrt{P}$ points sent to 3, 5, and 9. One sent to 8.
Processor 7: $2N/\sqrt{P}$ points sent to 4 and 8. One sent to 5.
Processor 8: $2N/\sqrt{P}$ points sent to 5, 7, and 9. One sent to 6.
Processor 9: $2N/\sqrt{P}$ points sent to 6 and 8.

In general, if there are P processors, then there will be 4 in the corners, $\sqrt{P} - 2$ along each of the 4 edges, and P - $4(\sqrt{P} - 2)$ - 4 on the inside of the lattice.

For each processor at the corner along the secondary diagonal (similar to 1 and 9), the total data sent is $4N/\sqrt{P}$ For corner processors along the primary diagonal (like 3 and 7), the total data sent is $(4N/\sqrt{P}) + 1$. For points along the edges (like 2,4,6,8), the total data sent is $(6N/\sqrt{P}) + 1$ For the remaining internal points, the data sent by each is $(8N/\sqrt{P}) + 2$

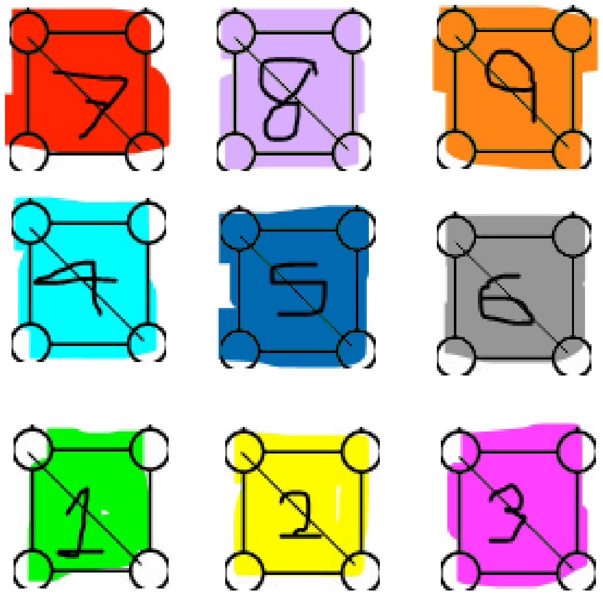


Figure 8: Connections between processors and points for triangulated grid

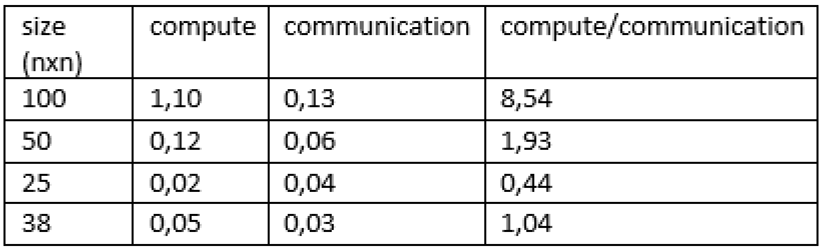
4.4

The asymmetry exists because, apart from the directly adjacent processors, a processor can only communicate with those located at the upper left corner or the bottom right corner. Figure 8 illustrates this and shows how points are connected along the "primary" diagonal, i.e., 2 communicates with the bottom right point of 4, and 8 communicates with the upper left point of 6. As a result, the central processor (i.e., number 5) communicates with six processors: 4 along the east, west, north, and south direction. The other two are along the northwest and southeast direction. For a corner point, the answer depends on the corner. It communicates with two processors if it is located at the bottom left or the upper right. Whereas if it is located at the upper left or the bottom right, it communicates with three processors.

4.5

a) Multiple timers were started to estimate the compute-to-communication ratio to measure the computation time, global communication, and neighbour-to-neighbour communication times. Table 16 shows the computation, total communication time (all in seconds), and the data locality ratio for a 1x4 configuration with different grid sizes.

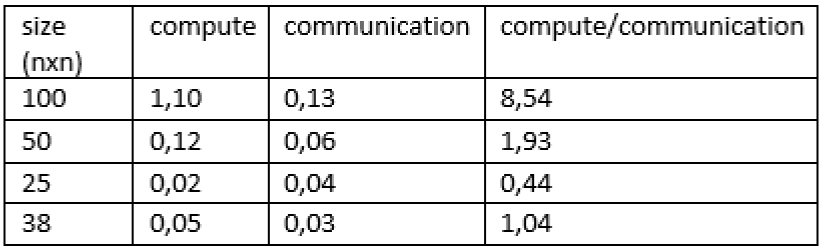
Table 15: Data locality ratio for different grid sizes with 1x4 configuration



size (nxn)	compute	communication	compute/communication
100	1,10	0,13	8,54
50	0,12	0,06	1,93
25	0,02	0,04	0,44
38	0,05	0,03	1,04

The closest grid size approximation to make the compute-to-communication ratio equal to one was 38x38. This was determined empirically, and the ratio came out to be 1.04

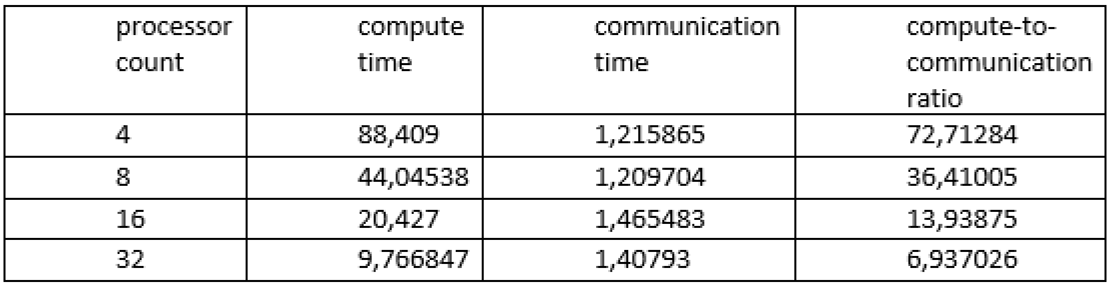
Table 16: Time-splits for different operations with 4 processors



size (nxn)	compute	communication	compute/communication
100	1,10	0,13	8,54
50	0,12	0,06	1,93
25	0,02	0,04	0,44
38	0,05	0,03	1,04

b) A similar procedure was applied to determine the number of processors required to make the ratio unity for a 1000x1000 grid size (see table 17). All processors are assumed to be arranged on a 1 x K configuration, where K is the number of processors. Due to resource limitations, it was not possible to spawn more than 32 processors. However, the data was used to extrapolate the required number of processors. On taking the logarithm with base 2 for both the number of processors and the compute-to-communication ratio, we get the plot shown in fig 9. On fitting a linear regression line and extrapolating this, we get the required number of processors as approximately 167.

Table 17: Data locality ratio for 1000x1000 grid



processor count	compute time	communication time	compute-to-communication ratio
4	88,409	1,215865	72,71284
8	44,04538	1,209704	36,41005
16	20,427	1,465483	13,93875
32	9,766847	1,40793	6,937026

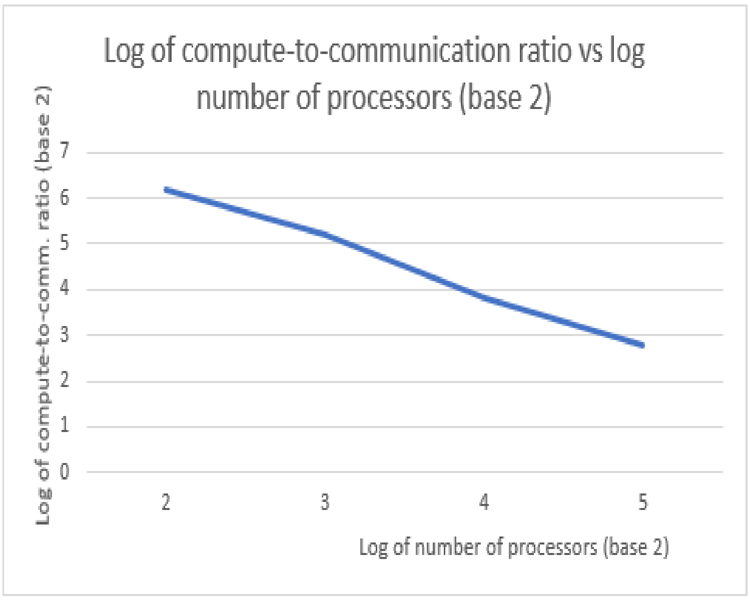


Figure 9: Data locality ratio vs number of processors on a logarithmic scale

4.6

To see the effect of using adapt, it is better to use an odd number of processors in the configuration because then the 100x100 points will not be evenly divided across each processor. The same holds for the 200x200 and the 400x400 grid sizes. The selected design was thus chosen to be 3x3. Table 18 shows the experimental results for the iterations, total time, and the compute time (in seconds) when using adapt. The results do not differ significantly compared to the "without adapt" scenario shown in table 19. Observe that the number of iterations for convergence when using adapt is slightly larger than without adapt for all three cases. The total time and compute time are almost the same in both cases.

Table 18: Results with adapt mode enabled

With adapt			
size	iterations	tot time	compute time
100x100	146	0,070	0,014
200x200	278	0,182	0,100
400x400	532	0,922	0,713

Table 19: Results with adapt disabled

without adapt			
size	iterations	tot time	compute time
100x100	141	0,069	0,013
200x200	274	0,192	0,098
400x400	529	0,919	0,709

Part III

Power Method on GPU

Important code snippets for this can be found in the Appendix IV. For the complete program, please visit link

Step 1

For making use of global/device memory, additional functions were created. For the code, see *Globmem_ Av_ Product(), Globmem_ FindNormW (), Globmem_ NormalizeW (), and Globmem_ ComputeLamda ().* Table 20 below shows the time comparison (in seconds) when using global memory vs shared memory for the default matrix size of 5000x5000 and a fixed number of iterations (1000). Observe that as expected, the runtime when using shared memory is greater than double that of when using global memory. This is because shared memory is located on-chip, unlike the global device memory. This is analogous to the relation between cache and main memory in computers. However, like cache, the drawback of shared memory is that it is much smaller than the device memory.

Table 20: Shared memory vs Global memory

memory type	total time (sec)	iterations
shared	0,97	1000
global	2,24	1000

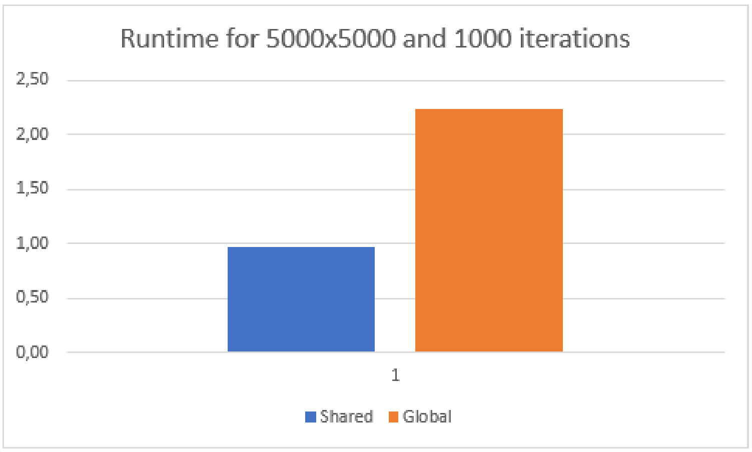


Figure 10: Global memory vs Shared memory execution time

Step 2

Table 21 shows how the computation-only time (seconds) varies with the matrix size and the number of threads within each block. The tests could not be done for 128 threads per block because the shared memory sizes exceeded the allocation size limits. Instead, analysis has been done using threads-per-blocks ranging from 2 to 64. The results show that the runtimes increase with increasing matrix dimensions. However, there is no clear correlation between increasing the number of threads per block and the total execution time. This is because the number of blocks is adjusted so that the total number of threads spawned across the entire grid remains approximately the same. Hence, if there were any differences between the cases, it would result from differing shared-memory sizes.

Table 21: Pure computation times for different dimensions and threads-per-block

1000 iterations	threads-per-block		GPU pure compute time			
matrix size	2	4	8	16	32	64
50	0,007657	0,007449	0,00681	0,007027	0,006504	0,006941
500	0,00786	0,00787	0,00787	0,007603	0,007193	0,007323
2000	0,00944	0,00824	0,00808	0,007808	0,008147	0,008025

Step 3

An additional timer was created to measure the time spent in CPU-GPU memory transfers. For code, see *mt_start* and *mt_end*.The speedup was calculated by taking the ratio of the execution times when running only on the CPU vs when running on the GPU. Table 22 shows the speedups when including and excluding communication times for different matrix sizes. As the size increases, the speedup increases in both cases. This is not surprising because as the workload increases, the effect of increasing parallelism becomes more evident. However, we notice that the speedup values between the two scenarios differ significantly. This is because communication costs account for a large fraction of the total times, as shown in figure 11. This indicates that the code is highly memory-bound. This also implies that the memory access times alone are greater than the total execution times when just running on a CPU. Hence for small sizes, using a GPU is not an efficient option.

Table 22: Speedup values when including and excluding communication costs for different matrix sizes

Matrix size	Speedup (including memory access)	Speedup (Excluding memory access)
50x50	0,0006	0,0200
500x500	0,0637	1,9536
2000x2000	0,6529	19,0817

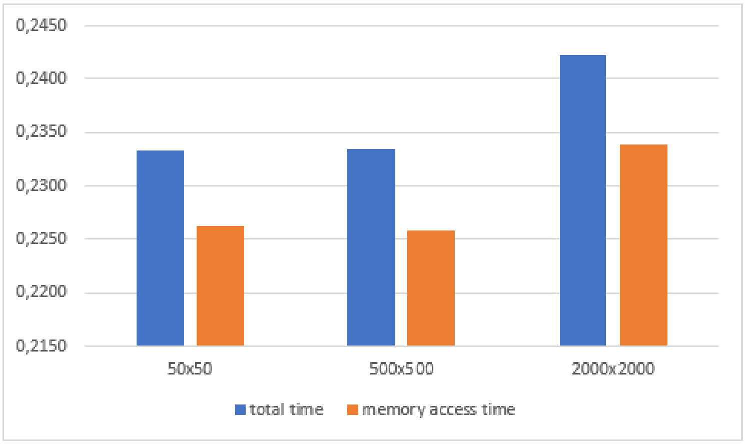


Figure 11: Total execution times vs memory transfer times for different sizes

Part IV

Appendix

Global memory implementation functions for exercise 3

```
//equivalent functions making use of global memory instead of shared memory
__global__ void Globalmem_Av_Product(float* g_MatA, float* g_VecV, float* g_VecW, int N)
{
        int bx = blockIdx.x;
        int tx = threadIdx.x;
        int globalIndex = bx*BlockSize+tx;
        if(globalIndex < N){
                int a_start_index = globalIndex*N;
                float sum = 0.0;
                for(int i = 0;i < N;i++){
                        sum = sum + g_MatA[a_start_index + i] * g_VecV[i];
                }
                g_VecW[globalIndex] = sum;
        }
}

__global__ void Globalmem_FindNormW(float* g_VecW, float* g_NormW, int N)
{
        int bx = blockIdx.x;
        int tx = threadIdx.x;
        int globalIndex = bx*BlockSize+tx;
        if(globalIndex < N){
                float squareTerm = g_VecW[globalIndex]*g_VecW[globalIndex];
                atomicAdd(g_NormW,squareTerm);
        }
}



__global__ void Globalmem_ComputeLamda(float* g_VecV, float* g_VecW, float* g_Lamda,int N)
{
        int bx = blockIdx.x;
        int tx = threadIdx.x;
        int globalIndex = bx*BlockSize+tx;
        if(globalIndex<N){
                float product = g_VecV[globalIndex]*g_VecW[globalIndex];
                atomicAdd(g_Lamda,product);
        }
}

__global__ void Globalmem_NormalizeW(float* g_VecW, float* g_NormW, float* g_VecV,int N)
{
        int bx = blockIdx.x;
        int tx = threadIdx.x;
        int globalIndex = bx*BlockSize+tx;
        if(globalIndex<N){
                        g_VecV[globalIndex] = g_VecW[globalIndex]/g_NormW[0];
        }
}
```

While-loop code for shared memory implementation of exercise 3

```
int k = 0;
while(abs(OldLamda - *h_Lamda) >= EPS && k <= max_iteration){
  Av_Product<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_MatA, d_VecV, d_VecW, N)
  cudaDeviceSynchronize();
```

```
    OldLamda = *h_Lamda;
    *h_Lamda = 0;

    cudaMemcpy(d_Lamda, h_Lamda, norm_size, cudaMemcpyHostToDevice);

    ComputeLamda<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_VecV, d_VecW, d_Lamda,
    N);
    cudaDeviceSynchronize();

    cudaMemcpy(h_Lamda, d_Lamda, norm_size, cudaMemcpyDeviceToHost);

    FindNormW<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_VecW, d_NormW, N);
    cudaDeviceSynchronize();

    cudaMemcpy(h_NormW, d_NormW, norm_size, cudaMemcpyDeviceToHost);
    float root;
    root = sqrtf(*h_NormW);

    //printf("sqrt is %f\n",root);
    *h_NormW = root;

    cudaMemcpy(d_NormW, h_NormW, norm_size, cudaMemcpyHostToDevice);

    NormalizeW<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_VecW, d_NormW, d_VecV,
    N);
    cudaDeviceSynchronize();
    *h_NormW = 0;

    cudaMemcpy(d_NormW, h_NormW, norm_size, cudaMemcpyHostToDevice);
    k++;
}
```

While-loop code for device memory implementation of exercise 3

```
while(abs(OldLamda - *h_Lamda) >= EPS && k <= max_iteration){

Globalmem_Av_Product<<<blocksPerGrid, threadsPerBlock>>>(d_MatA, d_VecV, d_VecW, N);

cudaDeviceSynchronize();

OldLamda = *h_Lamda;
*h_Lamda = 0;

cudaMemcpy(d_Lamda, h_Lamda, norm_size, cudaMemcpyHostToDevice);
Globalmem_ComputeLamda<<<blocksPerGrid, threadsPerBlock>>>(d_VecV, d_VecW, d_Lamda, N);

cudaDeviceSynchronize();
cudaMemcpy(h_Lamda, d_Lamda, norm_size, cudaMemcpyDeviceToHost);

Globalmem_FindNormW<<<blocksPerGrid, threadsPerBlock>>>(d_VecW, d_NormW, N);
cudaDeviceSynchronize();

cudaMemcpy(h_NormW, d_NormW, norm_size, cudaMemcpyDeviceToHost);
float root;

root = sqrtf(*h_NormW);
*h_NormW = root;

//printf("sqrt is %f\n",root);
cudaMemcpy(d_NormW, h_NormW, norm_size, cudaMemcpyHostToDevice);

Globalmem_NormalizeW<<<blocksPerGrid, threadsPerBlock>>>(d_VecW, d_NormW, d_VecV, N);
cudaDeviceSynchronize();


fprintf(stderr,"GPU lamda at %d: %f \n", k, *h_Lamda);

//fprintf(stderr,"tolerance at %d: %f \n", k, abs(OldLamda - *h_Lamda));
*h_NormW = 0;

cudaMemcpy(d_NormW, h_NormW, norm_size, cudaMemcpyHostToDevice);
k++;

}
```