CM30225 Parallel Computing
Assessed Coursework Assignment 2

Jay Rabjohns

December 2023

1 Introduction

Here we investigate the parallelisation of matrix relaxation using MPI to distribute the problem over an Azure cluster.

2 Parallel Approach

The main limiting factor of distributed matrix relaxation is the communication among nodes in between iterations. These are where the majority of the program's synchronous overhead will come from, since the latency of communication over a network far outweighs the latency of locally performing math operations. It is for this reason it is important to think carefully about the chosen approach to inter-node communication.

The naive and perhaps simplest way is to make the root node responsible for receiving and distributing progress among nodes between iterations. This could be achieved using `MPI_Scatterv`, which sends messages from the root to all other processes, and `MPI_Gatherv`, which does the opposite. This very naturally models a superstep but it leads to $2n$ communications being performed by the root node while all other nodes only perform 2. This gives the root a workload a factor of n larger compared to other nodes, creating a potential scalability bottleneck.

It is instead better to use point-to-point communications to synchronise nodes with their neighbours each iteration. Nodes are assigned a chunk of the matrix to work on and overlapping cells are communicated between neighbours between iterations. It's worth noting that `MPI_Scatterv` and `MPI_Gatherv` are still useful for the initial distribution and reconsolidation of work at the beginning and end of the program, because all nodes have to communicate with the root node regardless.

Processes should only communicate the bordering rows between neighbours, minimising the data being sent.

2.1 Partitioning

The simplest partitioning scheme is row partitioning, as shown in Figure 2, however, it may be better to use some other form of partitioning depending on the problem size.

Another form of partitioning would be to split the problem into two dimensions, creating a sort of grid similar to Figure 1. It may seem counter-intuitive at first, but this approach scales better to larger problem sizes for a fixed number of nodes, since each cell of the grid has a higher ratio of area to surface area, minimising communication volumes at partition boundaries.

This way of partitioning problems is common with MPI applications, meaning there are plenty of functions to help make the setup easier. For example, it is possible to create a custom communicator

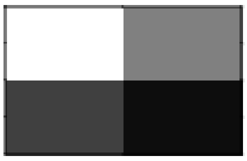


Figure 1: Cartesian Partitioning

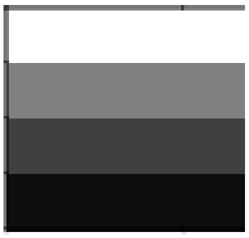


Figure 2: Row Partitioning

representing this topology using `MPI_Cart_create`, as shown in Figure 5, which is a grid of nearest neighbours generalised to any number of dimensions. Allegedly, implementations of this topology can make use of specialised hardware if it exists, which could lead to further performance improvements. However, this isn't investigated here.

The primary benefit of doing this is to be able to use generalised MPI functions such as `MPI_Neighbour_alltoall` , which use the neighbours defined by an underlying topology to efficiently perform all communications in a single MPI call. There is even a non-blocking version of the function `MPI_Ineighbour_alltoall`.

For the sake of simplicity, and because the cluster on which the program will be run is limited to 4 nodes, I partitioned the problem as rows. If the cluster was larger, the problem had more than two dimensions, or if there was some requirement to consider possible growth of the cluster, it may be worth using another strategy but for now, rows suffice.

2.2 Fairness

Each partition should ideally be of equal size. We achieve this by first dividing the total rows by the number of processes, excluding the first and last for of the problem because they will always remain the same. The remaining rows are then added to processes of rank < number of remaining rows. This algorithm is shown in Figure 3.

2.3 Point-to-point Communications

Our goal is to perform what is known as a halo exchange, a generalised term for the sharing of overlapping data between nearest neighbours. One approach is to use a virtual topology and `MPI_Neighbour_alltoall` as described above, but an alternative approach is to perform point-to-point

```
    int total_rows = (problem_size - 2);
    int rows_per_proc = total_rows / n_procs;
    int remainder_rows = total_rows % n_procs;
    int row_counts[n_procs]
    for (int i = 0; i < n_procs; i++)
    {
        row_counts[i] = rows_per_proc + (i < remainder_rows ? 1 : 0);
    }
```

Figure 3: Algorithm for fairly distributing work among partitions

```c

// The matrix is composed like this:
// fffffff <- first_row, we receive this from neighbour above
// ttttttt <- top_row, we send this to neighbour above
// #######
// ####### <- rest of the matrix
// #######
// bbbbbbb <- bot_row, we send this to neighbour below
// lllllll <- last_row, we receive this from neighbour below
MPI_Datatype row_t;
MPI_Type_contiguous(p_size, MPI_DOUBLE, &row_t);
MPI_Type_commit(&row_t);

int proc_above = (rank >= 1) ? rank - 1 : MPI_PROC_NULL;
int proc_below = (rank < n_procs - 1) ? rank + 1 : MPI_PROC_NULL;

MPI_Sendrecv(top_row, 1, row_t, proc_above, DEFAULT_TAG,
             first_row_prev, 1, row_t, proc_above, DEFAULT_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(bot_row, 1, row_t, proc_below, DEFAULT_TAG,
             last_row_prev, 1, row_t, proc_below, DEFAULT_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Figure 4: Point-to-point communications using Sendrecv operations.

communications between nodes manually. The advantages of this for this particular problem is that it will be conceptually easier to understand specific communications between neighbours than many MPI specific concepts.

Naively point-to-point communications can be achieved using `MPI_Send` and `MPI_Recv` using the global communicator `MPI_COMM_WORLD`. Each process sends and receives overlapping rows to processes of rank ± 1 the current rank. Due to these being blocking calls some strategy is needed to organise which nodes send first and which receive first to avoid deadlock. A common algorithm for this is for processes with an even rank to receive first and processes with an odd rank to send first.

This implementation is poor because it should be possible to overlap these communications with each other to reduce the total time communicating. One way to overlap communications is to use `MPI_Sendrecv` which performs both a send and a receive simultaneously. Figure 4 shows an exerpt of how this was implemented for part of the investigation in Section 3. Some tricks have been used to simplify the implementation, `row_t` is a custom MPI type representing a contiguous block of memory which better matches the mental model of communicating rows at a time rather than many double values. `MPI_PROC_NULL` is a special source or destination which signifies no communication should take place, it is useful for handling boundary cases. All variables pointing to rows are pointers to the local matrix, avoiding unnecessary copies.

Another approach is to use non-blocking send and receive functions, `MPI_Isend` and `MPI_Irecv`, which will return locally before the communication is guaranteed to have completed. The MPI implementation then handles the scheduling and sending of communication. It is possible to later wait for the communications to complete using `MPI_Waitall`.

Using non-blocking calls it is also possible to overlap computations with communications, by precomputing the first and last rows of the process' local matrix, it is possible to commence communications almost immediately at the start of an iteration. Then, while performing local calculations MPI is handling the communications in the background.

There seems to be no clear-cut way of performing point-to-point communications, some sources cite that the non-blocking variants for MPI communications are rarely implemented efficiently and either

```c
    int periods[2] = {1,1};
    int dimensions[2] = {0,0};
    int n_procs;

    MPI_Comm grid_comm;
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    MPI_Dims_create(n_procs, 2, dimensions);
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods, 1, &grid_comm);

```

Figure 5: Creating a custom communicator representing a 2D grid.

cause large buffers to be reserved on the receiving side or multiple *rendezvous* communications to determine when the primary communication should happen. Since communication is the bottleneck of this problem, this is worrying for performance. Some sources are claiming that they have profiled both and found `MPI_Sendrecv` to be faster! We decided to investigate this for ourselves and the results can be found in Section 3.

2.4 Convergence

To calculate whether the problem has converged globally, we have opted to use `MPI_Allreduce` along with a logical AND operation. This executes with the same semantics as all processes' flags being collected into an array, reduced to true if they are all true and false otherwise, then being redistributed to all processes. A non blocking version exists, `MPI_Iallreduce`, but we opted to not use it because it seemed rather pointless to potentially send multiple rendezvous messages for what is ultimately a very small message. I don't think it would have made a measurable difference and it would have increased the complexity of the program, so I opted for the blocking approach. An outline of the algorithm used can be seen in Figure 6.

```c
    bool has_converged = false;
    while (!has_converged){
    /* Other work... */

    // Check if converged on local problem
    // Ignore first and last rows as they belong to other processes
    has_converged = matrix_has_converged(precision, n_rows - 2, p_size,
                                         *top_row, *top_row_prev);

    bool all_have_converged = false;

    // Reduce convergence flags with logical-and operation
    // Send type is an MPI_BYTE because MPI_LAND doesn't support C booleans
    MPI_Allreduce(&has_converged, &all_have_converged, 1, MPI_BYTE,
                  MPI_LAND, MPI_COMM_WORLD);

    // Force chunk to continue iterating if flag is false elsewhere
    has_converged = all_have_converged;
    }

```

Figure 6: Checking if the matrix has converged globally

3 Scalability Investigation

As mentioned before, we have investigated the effects of using a blocking `MPI_Sendrecv` vs pairs of `MPI_Isend` and `MPI_Irecv` where the receive is posted first and the same but when send is posted first. This was to verify online claims that the blocking `MPI_Sendrecv` can be as performant or even more performant than the non-blocking operations due to poor implementation.

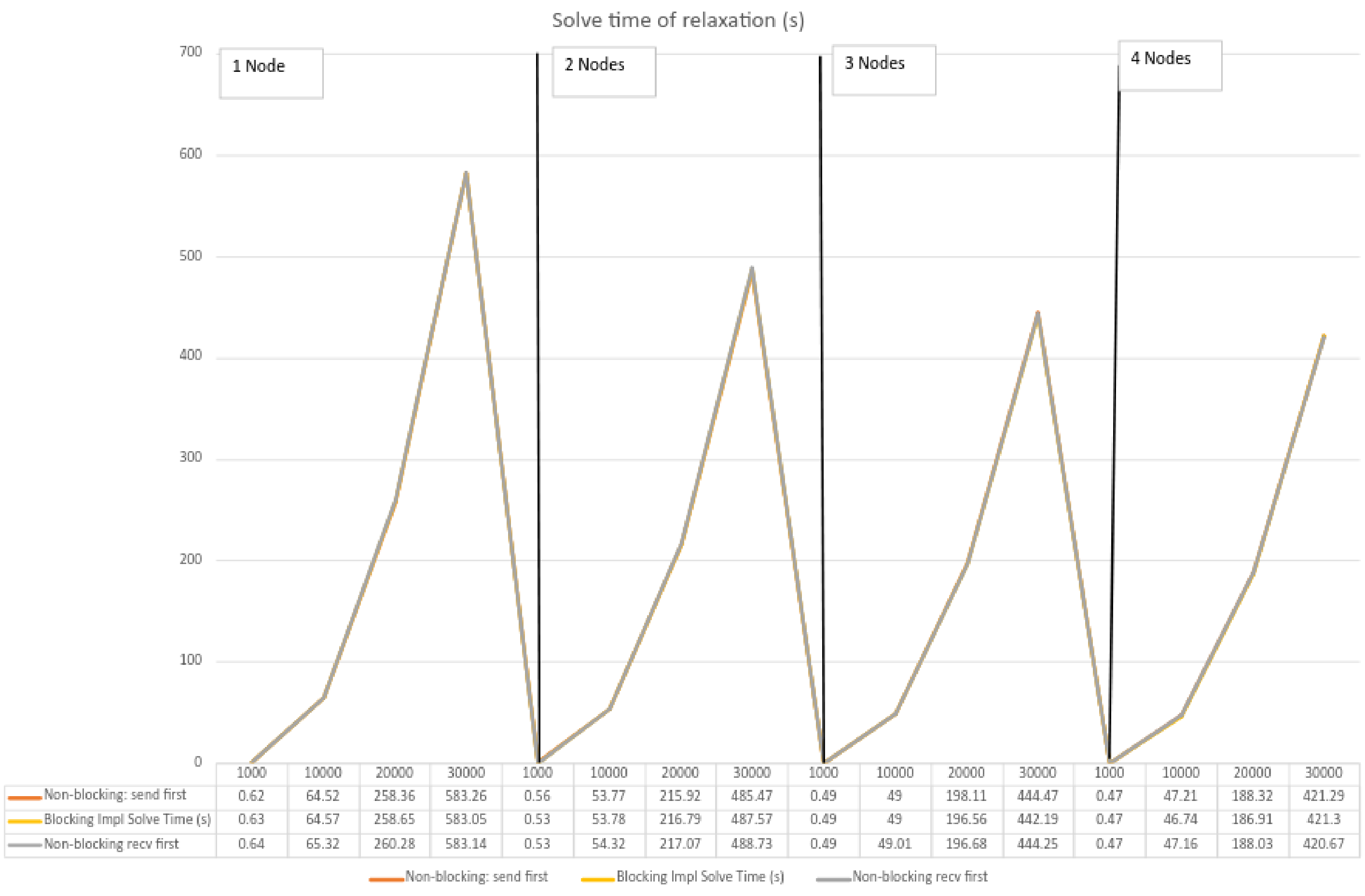
The results in Figure 8 show the speedup for each implementation on a range of problem sizes in comparison to running on a single node. The speedups are similar, but clearly, the non-blocking implementation with receives posted first is the best approach. It is still surprising that the communication is so comparable, signifying that either the online claims were correct or the implementation we have provided still has significant synchronous overheads. The non-blocking implementation with receives posted first was marginally better performing, so we further tested the effects of using multiple MPI processes per node. Solve times are illustrated in Figure 9 and Speedups are in Figure 10.

We see that increasing the number of tasks per node to 16 has the largest positive effect on speedup, with the highest speedup being 1.6. This is not very high, meaning there is still a fair amount of synchronous overhead in the provided implementation. Looking back, it would be interesting to see if switching `MPI_Allreduce` to `MPI_Iallreduce` would have a larger effect than previously thought. This is something for future investigation.

The speedup results don't come close to the Amdahl limit, which assuming a limit of 0 synchronous overheads would be 4 for an implementation using a single task per node with 4 nodes.

While the results show relatively poor speedup, it may still be useful to use a distributed implementation of matrix relaxation. Gustafson's law implies that the target of parallelism is to work with larger problem sizes, and from 8 we can clearly see that the program is easily able to handle large problem sizes with little to no negative effect on speedup.

The efficiency of the program running with a single task per node is illustrated by 11. It drops as expected when increasing the number of nodes.



	1000	10000	20000	30000	1000	10000	20000	30000	1000	10000	20000	30000	1000	10000	20000	30000
Non-blocking: send first	0.62	64.52	258.36	583.26	0.56	53.77	215.92	485.47	0.49	49	198.11	444.47	0.47	47.21	188.32	421.29
Blocking Impl Solve Time (s)	0.63	64.57	258.65	583.05	0.53	53.78	216.79	487.57	0.49	49	196.56	442.19	0.47	46.74	186.91	421.3
Non-blocking recv first	0.64	65.32	260.28	583.14	0.53	54.32	217.07	488.73	0.49	49.01	196.68	444.25	0.47	47.16	188.03	420.67

Figure 7: Solve time of relaxation for two parallel approaches and a blocking approach

4 Testing

By manually calculating a handful of small problem size examples we are able to develop a synchronous implementation and be reasonably confident that it is correct. This can then be used to compare results with the parallel implementation. This is referred to from here on as 'Test 1'. Results from each comparison are checked within the precision of the problem.

Another test is to check that the solution from the parallel implementation has converged. This will be referred to as 'Test 2'. You might think that Test 1 implies Test 2 but that isn't the case, there are edge cases where either can be true while the other is false. For this reason, we have included both in the supplied test tables.

We have included test tables for all three tested implementations. All tests are run on matrices of the form (1) because this tends to converge in a reasonable amount of time, allowing the testing of larger matrices in the fixed task time window provided on the shared cluster.

$$A := \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 0 \end{bmatrix} \tag{1}$$

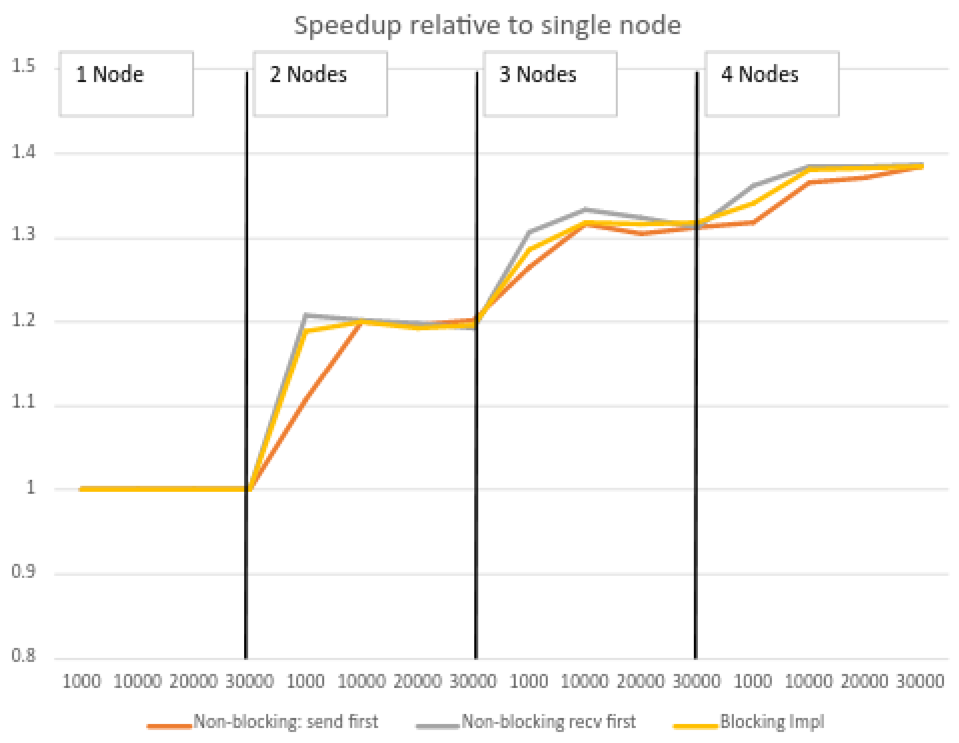
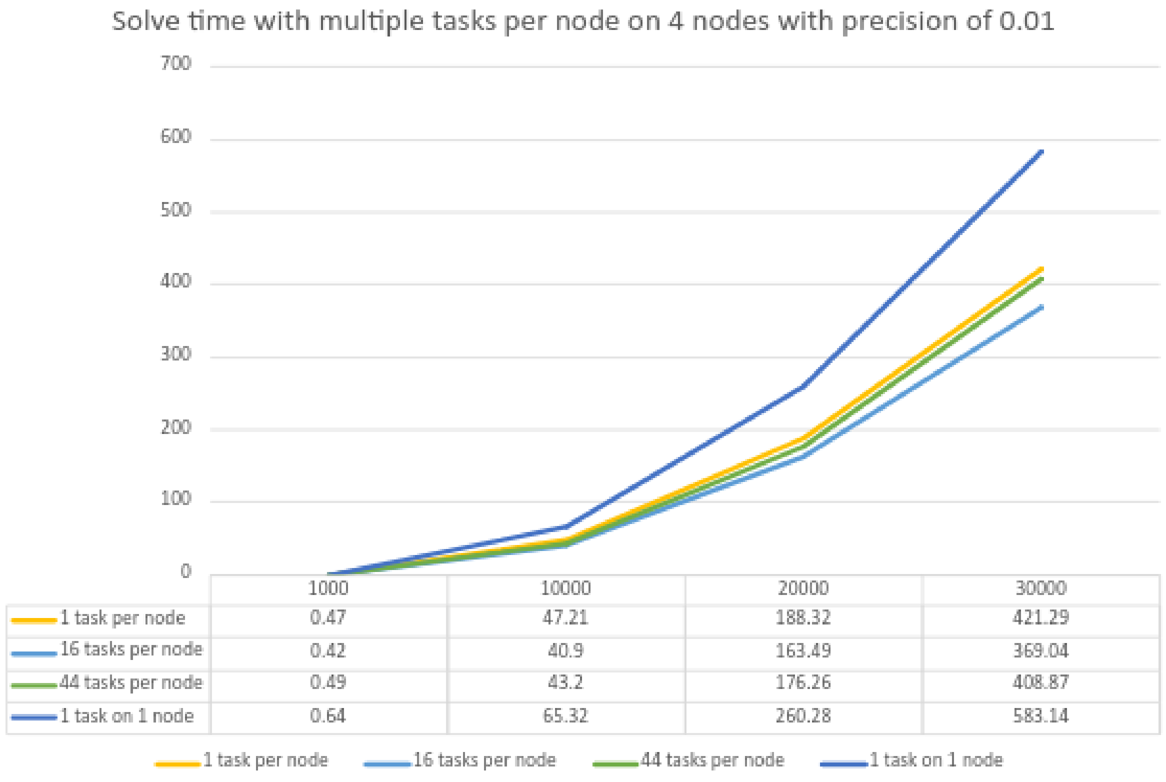


Figure 8: Speedup of each implementation relative to a single node



	1000	10000	20000	30000
1 task per node	0.47	47.21	188.32	421.29
16 tasks per node	0.42	40.9	163.49	369.04
44 tasks per node	0.49	43.2	176.26	408.87
1 task on 1 node	0.64	65.32	260.28	583.14

Figure 9: Solve time for multiple tasks per node on 4 nodes with precision of 0.01

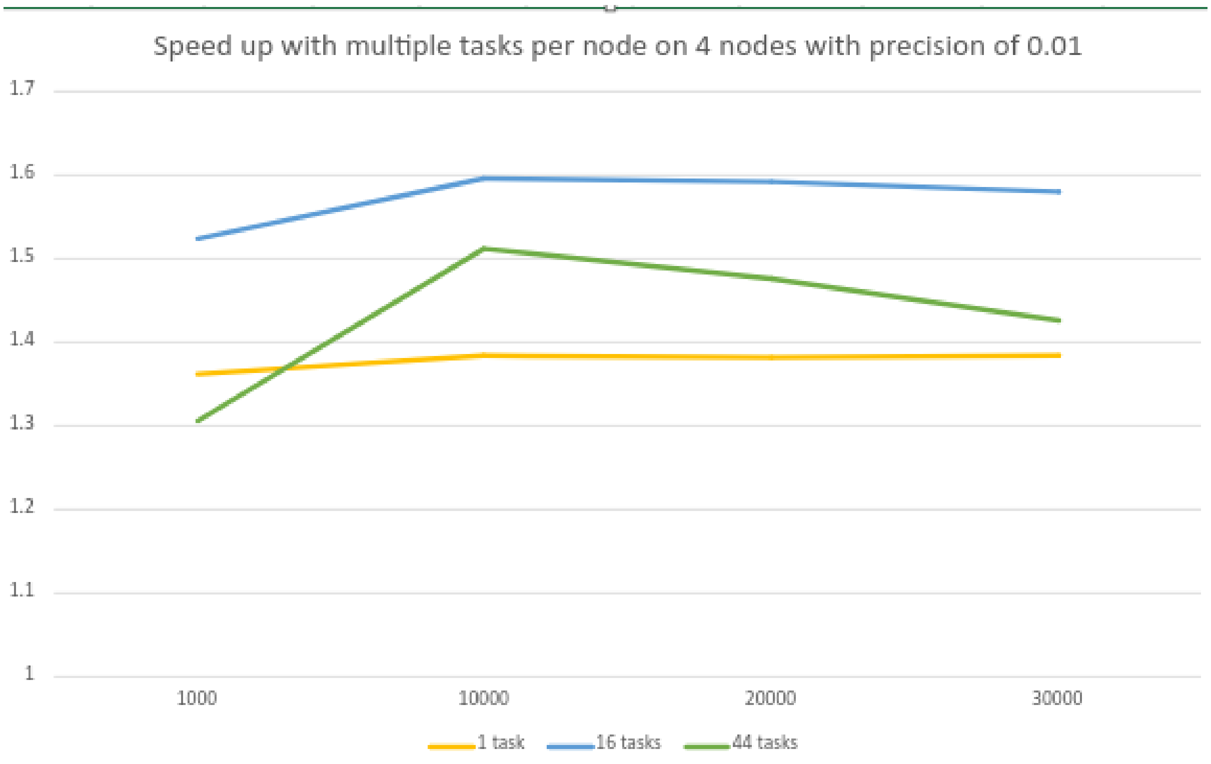


Figure 10: Speedup for multiple tasks per node on 4 nodes with precision of 0.01

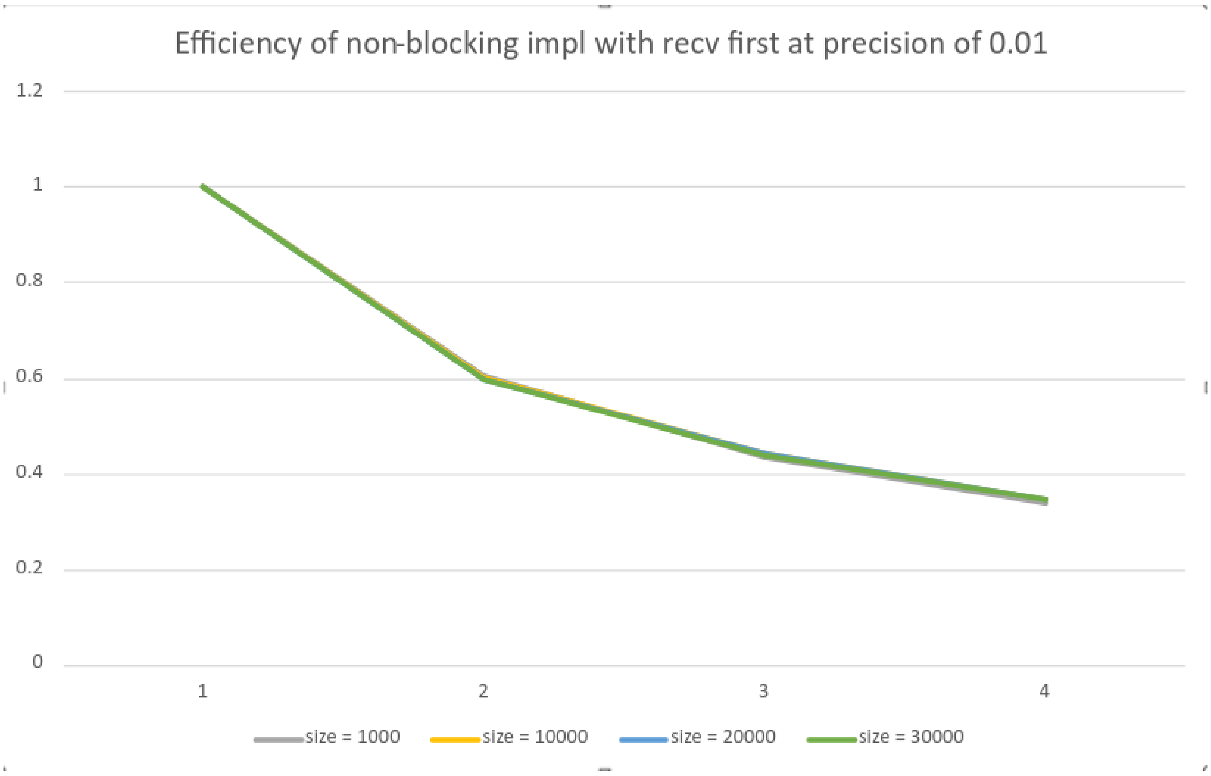
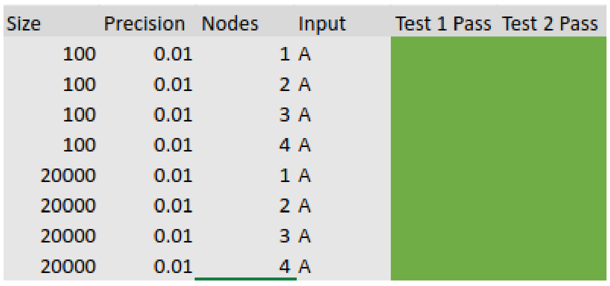


Figure 11: Efficiency in comparison to using a single node, using a single task per node

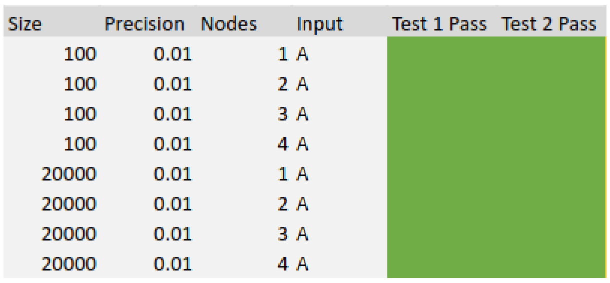
4.1 Errors

Almost all MPI functions return an error value which can be checked and dealt with manually, as seen in Figure 16. Due to the MPI calls being used in this project, it isn't possible to recover elegantly



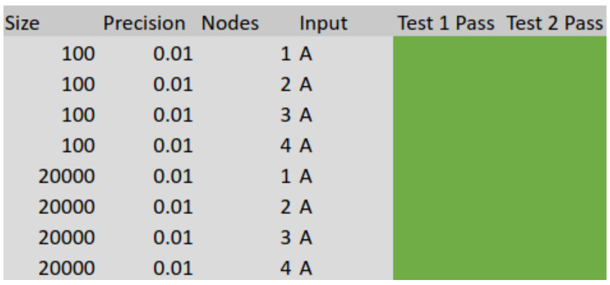
Size	Precision	Nodes	Input	Test 1 Pass	Test 2 Pass
100	0.01	1	A	🟩	🟩
100	0.01	2	A	🟩	🟩
100	0.01	3	A	🟩	🟩
100	0.01	4	A	🟩	🟩
20000	0.01	1	A	🟩	🟩
20000	0.01	2	A	🟩	🟩
20000	0.01	3	A	🟩	🟩
20000	0.01	4	A	🟩	🟩

Figure 12: Test table for the non-blocking implementation with send posting first. Input A is given by (1).



Size	Precision	Nodes	Input	Test 1 Pass	Test 2 Pass
100	0.01	1	A	🟩	🟩
100	0.01	2	A	🟩	🟩
100	0.01	3	A	🟩	🟩
100	0.01	4	A	🟩	🟩
20000	0.01	1	A	🟩	🟩
20000	0.01	2	A	🟩	🟩
20000	0.01	3	A	🟩	🟩
20000	0.01	4	A	🟩	🟩

Figure 13: Test table for the blocking implementation. Input A is given by (1).



Size	Precision	Nodes	Input	Test 1 Pass	Test 2 Pass
100	0.01	1	A	🟩	🟩
100	0.01	2	A	🟩	🟩
100	0.01	3	A	🟩	🟩
100	0.01	4	A	🟩	🟩
20000	0.01	1	A	🟩	🟩
20000	0.01	2	A	🟩	🟩
20000	0.01	3	A	🟩	🟩
20000	0.01	4	A	🟩	🟩

Figure 14: Test table for the non-blocking implementation with receive posting first. Input A is given by (1).

```
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_ARE_FATAL);
```

Figure 15: Setting the MPI error handler to crash on encountering an error.

```
    int rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS)
    {
        fprintf(stderr, "MPI_Init failed \n");
        // Handle error
    }
```

Figure 16: Manually handling MPI error return codes

if one fails. For instance, if a communication fails due to a network problem, there isn't a way to recover other than to retry for a bit. It would instead be better if the program intentionally crashed in these scenarios so the programmer could be alerted and look at the logs to figure out what happened. Figure 15 illustrates how to do this. Technically speaking MPI_ERRORS_ARE_FATAL is the default behaviour of MPI, so this line isn't strictly necessary but I think it signifies that this behaviour is intentional and the lack of checking return values is also intentional.

A An Example of a Passed Test Output

```
row counts: 2500, 2500, 2499, 2499,
row starts: 0, 2500, 5000, 7499,
send counts: 2502, 2502, 2501, 2501,
size of row: 80000b
[3] total iterations: 360
[0] total iterations: 360
[1] total iterations: 360
[2] total iterations: 360
Solved in 1112.751228s

[SYNC] solved in 360 iterations in 271s
PASS Solutions match within a precision of 0.001000
PASS Final matrix has successfully converged with a precision of
   0.001000
```