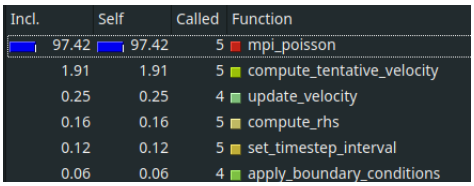
CS402 Coursework 2

u1805023

1 Introduction

The aim of this coursework is improve the execution speed of the cfd program by parallelisation using MPI and optionally OpenMP. The program consists of a main loop which calls five functions to move the simulation forwards by one time step. Profiling showed that the *poisson* function was by far the most expensive taking up 97-98% of the execution time, as can be seen in Figure 1, with computing the tentative velocity coming second with the other functions contributing a negligible amount to the time for each iteration.



Incl.	Self	Called	Function
97.42	97.42	5	mpi_poisson
1.91	1.91	5	compute_tentative_velocity
0.25	0.25	4	update_velocity
0.16	0.16	5	compute_rhs
0.12	0.12	5	set_timestep_interval
0.06	0.06	4	apply_boundary_conditions

Figure 1: Callgrind profile of time spend in each function

2 MPI

Two parallelisation approaches for MPI were considered. The first used the fact that *poisson* took up much of the execution time to only parallelise *poisson*, synchronising the entire matrix p at the end of the function. This would remove the need for any communication in the other functions which otherwise may be dominated by the latency of synchronisation. This however would would increase the cost of synchronisation in *poisson* as the area of one matrix has to be copied to every other process.

The second approach would synchronise the edges each pair of MPI process share for each matrix updated in a function call. This has the benefit of transferring much less data as only the perimeter of the process area has to be transferred which grows at a smaller rate than the area allocated to the process. The downside is that the synchronisation occurs more frequently and this method is heavily dependant on a low transfer start latency.

After latency tests using the *pingpong* tool showed the transfer time for small blocks was very low, the second method was chosen. Work was allocated to processes by dividing the mesh into a 2D grid of tiles and allocating each process one tile. To determine in what shape the grid should be cut, the tile shape initialisation function *init_tile_shape* tests all layouts and calculates the expected communication cost using the following formula.

$$comm_x = \begin{cases} 2 * t_{start} * t_{byte} * tile_{height} & \text{if } tiles_{numx} > 1 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$comm_y = \begin{cases} 2 * t_{start} * t_{byte} * tile_{width} & \text{if } tiles_{numy} > 1 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$comm = comm_x + comm_y \tag{3}$$

Each tile is approximately the same size, with the the right and bottom edge tiles possibly being smaller if the mesh axis sizes are not a multiple of the grid shape in that axis. This means those tiles may complete slightly faster an end up not being fully utilised. However as each tile edge is at most one cell above the minimum size

required to cover the area with p processors the difference in size between a normal tile and the smallest tile is at most p cells.

Tiles are assigned to MPI processes based on their rank using the following formula. This allows a process to calculate the ranks of neighbouring processes quickly.

$$tile_{posx} \equiv rank\%tiles_{numx} \tag{4}$$

$$tile_{posx} = \lceil \frac{rank}{tiles_{numx}} \rceil \tag{5}$$

To simplify debugging and minimise the number of changes to the code the full mesh is kept by each process rather than allocating a smaller chunk of memory the size of the tile. However as the process only operates on its own small region there should be minimal effects on processing time, for example due to the jump from the end of a tile column to the start of the next.

As the program uses a 5 point stencil when updating the matrices each tile requires the surrounding region of depth 1 to be updated. The function *halo_sync* handles this synchronisation for a given input matrix, swapping edges with each of the neighbouring processes. Given the 5 point stencil this should only require the edges to be copied, not the corners. However the velocity estimation function performs 4 diagonal accesses which means that the corners also have to be synced when the mesh is split into a 2D grid of tiles (1D is not affected as the border updates correct the data).

Each tile is initialised with two datatypes, one for columns, one for rows. While not all rows and columns are of equal width, all tiles in a row or column share the same dimensions along that axis.

Asynchronous transfer was chosen despite the overhead of having to check separately if the transfer has completed. If as synchronous transfer had been used there would be the risk of deadlock (via a dependency loop in the 2D grid) if processes did not perform the transfers in a specific order. Additionally as the edges of the tiles are fairly small, a significant proportion of the cost in transferring is the start up latency. This latency can be partly hidden by starting multiple transfers asynchronously and using the available network bandwidth rather than waiting for the neighbouring tile to receive the edge data and then respond.

Additional transfers also occur when calculating the next timestep interval as well as calculating $p0$ and the residual in the poisson function. To solve these calculations while the data is spread across multiple MPI processes the operation is performed locally first over tile data and then the results are further processed in an MPI reduction. In the case of the timestep interval function the operation performed is an element-wise maximum over an array of the local umax and vmaxes while the $p0$ and residual calculations use a sum reduction. As each rank needs the result of these calculations the function *MPI_Allreduce* is used to perform a reduction to one node followed by a broadcast of the result to all nodes.

3 OpenMP

While MPI sending and receiving latency within a node is very low, with pingpong latencies below $10\mu s$ for messages with up to 64KB, the overhead becomes noticeable as the core count within a node increases. Additional care would have to be taken to ensure all processes on a node have neighbouring tiles to avoid excessive transfers between nodes. Unless the synchronisation mechanism were changed, having an MPI process for each core in a machine would also result in many smaller packets being sent, resulting in greater overhead due to packet headers and the processing required for each packet.

Processing within a node is supported by OpenMP using shared memory instead of explicit sends and receives which should lead to lower overhead. However unlike MPI, OpenMP only spawns the extra threads when instructed to do so, meaning there is an extra cost when starting an OpenMP parallel section. Parallel regions must be chosen carefully to ensure the overhead does not outweigh the gains due to parallelisation.

Two functions were parallelised with OpenMP, calculation of tentative velocity as well as poisson. Parallelisation of the former was fairly straightforward, with an outer parallel region within which each loop used the created threads with a plain *for* directive. The use of a single outer parallel region means the threads only need to be created once for each of the two which are parallelised loops. The other two smaller loops were not parallelised as the small size means the overhead is too large to result in a gain in speed. Finally the MPI synchronisation function calls take place outside the loop to prevent messages being sent more than once. Since

only the matrices need to be shared, all other parameters were declared *firstprivate* so each thread has a private copy.

Parallelisation of the poisson function was slightly more complex as a number of MPI calls have to occur while multiple threads are running. The iteration loop is wrapped in a parallel region to avoid repeatedly spawning threads inside the loop. The first iteration over the mesh is split up among the threads using a *for* directive. After the for loop the mesh p must be synced with other MPI processes by a single thread and so is run inside a region with a *single* directive. The single thread region has an implicit barrier causing the other threads to wait for completion, ensuring they do not continue until the sync has completed. Summation of the residual is performed with another OpenMP for loop, this time using a reduction to sum up the tile's residual. Adding up all the residuals of all the tiles is performed in another single threaded region to prevent multiple sends. Due to the greater proportion of single threaded regions the scaling of the poisson function is expected to be less than optimal.

All parallelised sections use static scheduling as the cost of each iteration is approximately uniform. Areas of solid material will be faster to compute but these areas are assumed to be small compared to the area of the entire mesh and the benefits of using another scheduling algorithm such as guided or dynamic would be outweighed by the overhead introduced.

4 Other changes

An issue discovered while generating large new datasets on the cs402 partition was that the time required for each poisson step would start off low and then increase by 30-50x before dropping down to the normal value. This couldn't be reproduced on other partitions such as desktop-batch or locally. Testing various fixes revealed that denormalised floating point values were the cause of the issue as setting the flag to flush denormalised numbers to zero resolved the issue.

Another issue was encountered was poor scaling beyond 6 threads on the cs402 partition despite being able to achieve reasonable scaling on the grace compute nodes which contain the same hardware. As the scaling was similar for 1-6 threads

5 Results

5.1 Testing

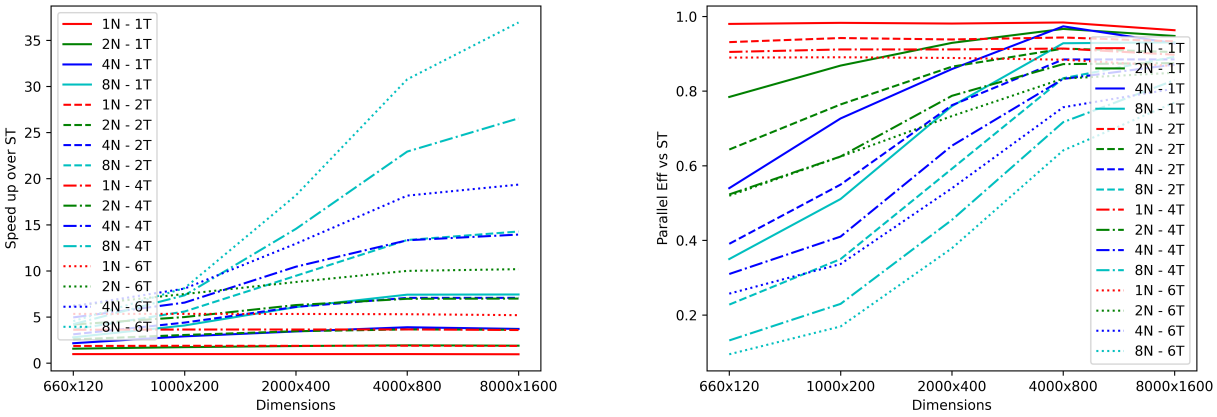
To measure the performance of the implementation, tests were run for a number of different mesh dimensions, number of nodes and OpenMP threads. Each of the meshes used was initialised as a new mesh and not loaded from a file. Timing measurements were made using *MPI_Wtime* on MPI process 0. Since each process must synchronise at the end of each function the timings for one process will be very close to those for the slowest process.

5.2 Scaling with mesh dimension

The mesh size is an important factor which affects the speed up parallelisation is able to provide. For small meshes the overhead of managing OpenMP threads and synchronising with MPI nodes becomes a noticeable factor. As can be seen in Figure 2a for small meshes the scaling is poor, with system performing more than 6 times faster than the single threaded version. For large numbers of threads the performance is worse with more nodes due to the extra overhead in communicating. For example when running 6 threads the 8 node configuration is slower than all others. Interestingly the overhead for small meshes for a single node is minimal with a similar speed up regardless of the mesh size.

However as the mesh size increases the proportional effect of the overhead drops off, allowing for much better scaling, with around 36 times scaling for 8 nodes of 6 threads. For very large meshes another interesting pattern can be observed. For configurations with the same total thread count (nodes multiplied by OpenMP threads) the scaling is better for those with more nodes. For example 8N2T outperforms 4N4T and 8N1T outperforms 4N2T and 2N4T. This could be due to a variety of factors including effectively more available cache for each thread or lower latency transfers due to the same amount of data being transferred by more links.

Parallel efficiency is a good way to visualise this overhead, shown in Figure 2b. Again the overhead for a single node remains constant for various mesh sizes but increases with thread count. It also shows that the overhead



(a) Speed up vs ST

(b) Parallel efficiency

Figure 2: Effects of varying mesh dimensions on performance

for OpenMP is small as the 1N1T configuration achieves around 96% of the single threaded implementation's performance, an overhead of around 4.1%. For more than one node the parallel efficiency initially is lower and decreases with more threads. As expected the overhead for an equal number of total threads split across more nodes is greater than the same number across a smaller set of nodes, e.g. 8N1T vs 4N2T.

The efficiency increases as the mesh size increases to around 75-80% for the worse case of 8N6T.

5.3 Scaling with OpenMP threads

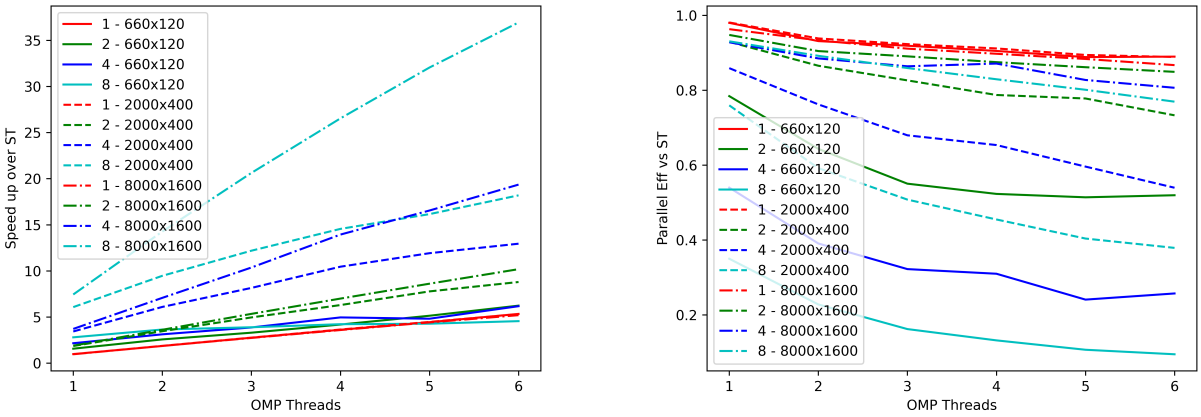
Scaling with OpenMP threads is expected to be noticeably less than linear as many sections are not parallelised. Only the two most expensive functions, compute tentative velocity and poisson, are parallelised due to the cost of initialising the threads and the poisson function contains number of single threaded sections. For configurations with few nodes the speed up in Figure 3a is close to linear, consistent with the expectation that a lower amount of overhead due to no MPI syncs. For multi-node configurations the scaling increase is less and curves down as the overhead becomes a proportionally bigger factor due to more communication and faster processing of the parallel sections. This effect is most noticeable with the 8 node configuration with the small meshes hardly scaling.

The parallel efficiency graph in Figure 3b confirms these observations with smaller meshes resulting in much lower parallel efficiency than larger meshes or fewer nodes.

Looking at the time taken for the parallelised loops in Figure 4a it can be seen that the time spent generating the pressure matrix is the most expensive of the measured loops with the residual calculation following that. These measurements are cumulative over a single timestep which explains the difference between those loops and the compute tentative velocity function which only runs once per timestep. Overhead due to OpenMP is present but doesn't significantly effect the timings, contributing only around an extra 6% for large meshes. This is reflected in the parallel efficiency in Figure 4b with efficiencies of ¿93% for the poisson function loops. The efficiency for computing the tentative velocity is much lower for larger meshes, decreasing to 75% for 6 threads, likely due to the overhead of creating the processes being spread over a shorter period of time. However the smallest mesh shows excellent scaling in contradiction to the expectation that larger meshes will be more efficient at using parallel resources.

5.4 Scaling with nodes

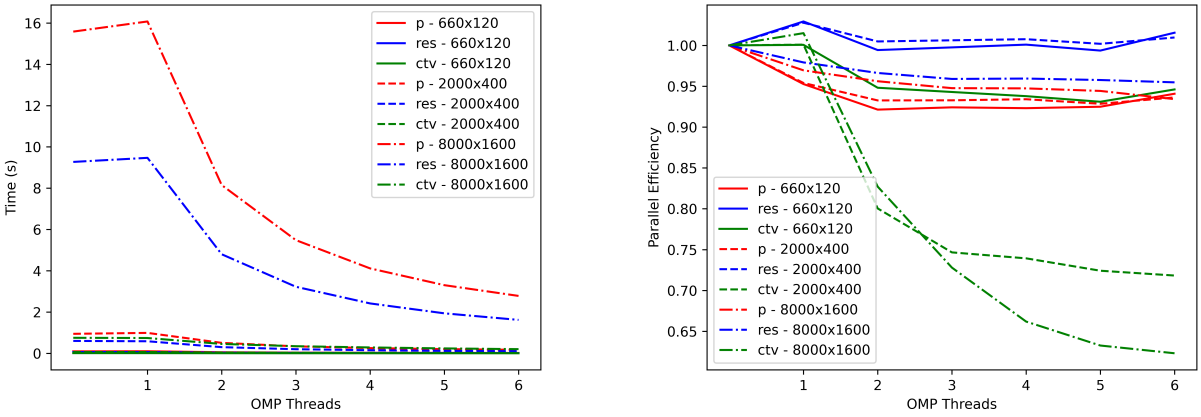
Scaling a program across multiple nodes introduces potentially significant overhead due to the cost of transferring data over the network compared to copying data between threads in shared memory. More nodes not only introduce more overhead but also can increase the effective proportion of the overhead if the actual computation is sped up.



(a) Speed up vs ST

(b) Parallel efficiency

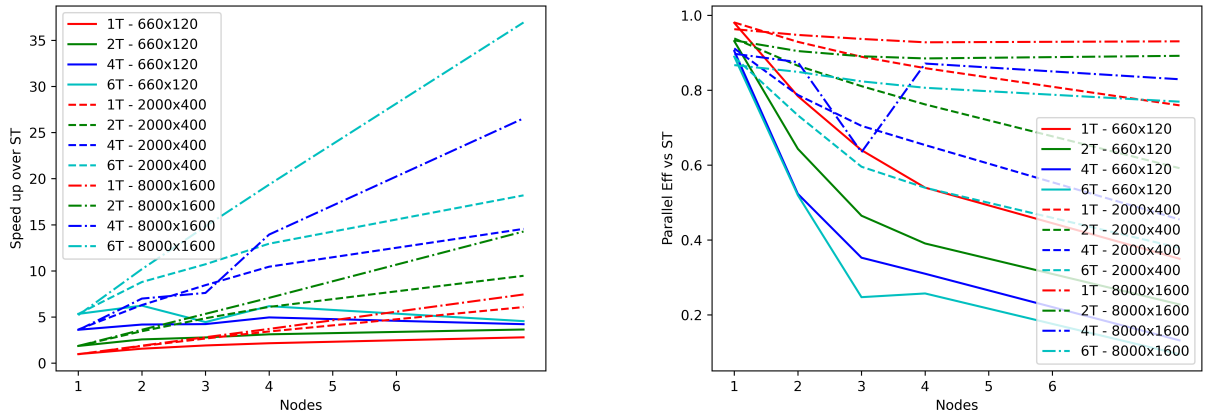
Figure 3: Effects of number of threads on performance



(a) Cumulative time for each loop for each timestep

(b) Parallel efficiency

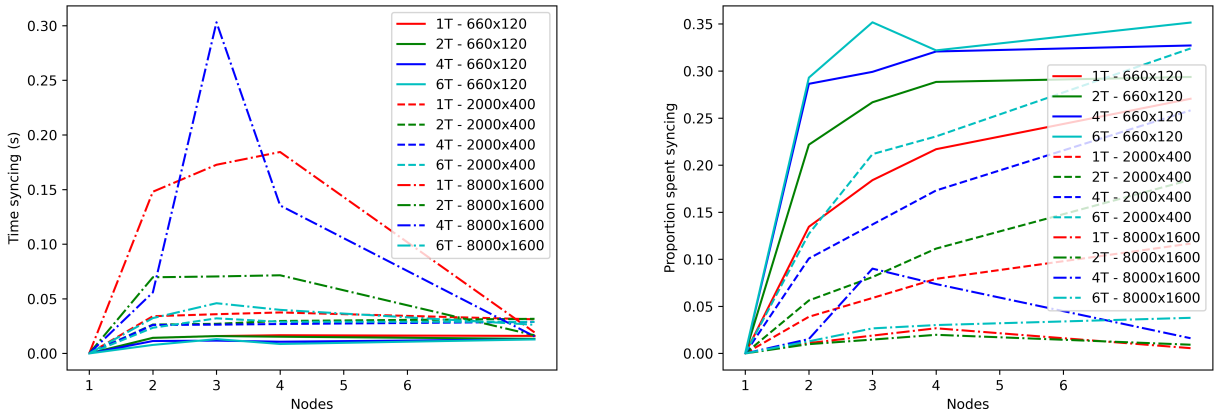
Figure 4: Parallel efficiency of loops



(a) Speed up vs ST

(b) Parallel efficiency

Figure 5: Effects of varying number of nodes on performance



(a) Average time taken to sync per timestep

(b) Proportion of time spent syncing

Figure 6: Effects of node number on time spent syncing

These effects can be clearly seen in Figure 5a with the 6 thread configuration on small meshes actually regressing in performance after 2 nodes while configurations with fewer threads scale poorly. This indicates multi-node configurations should be avoided for small meshes as the effect will likely be detrimental. For medium meshes the speed up gradient decreases notably with increasing node count indicating that while more performance can still be achieved, overhead is becoming a noticeable factor. Large meshes don't scale perfectly but no slowing of scaling is visible.

Looking at the parallel efficiency in Figure 5b shows the expected pattern with small meshes resulting in poor efficiency. More threads again scale worse due to more and proportionally larger overhead.

The time spent synchronising per timestep was recorded to provide the ability to determine how much of the processing time was spend syncing. Measuring the time taken to sync is slightly tricky as each process may have to wait some time for other processes to complete. This technically measures the total synchronisation time required but not the time taken to transfer.

Time spent communicating is small compared to the time taken for an entire timestep for large meshes with a proportion around 5%. For medium to smaller meshes the time spent communicating increases as more nodes are added but the rate of increase drops off as the latency of a single packet becomes less important.