Using MPI to Parallelise Computational Fluid Dynamics

Ed Pasfield 1898594

I. Introduction

Computational fluid dynamics (CFD) is the calculation to find the solution of the equations related to fluid flow, over a specified time and space. Fluid dynamics is a problem domain mainly studied by physicists and is based on three fundamental laws.

1) Mass is conserved
2) Newton's second law
3) Energy is conserved

It is expressed in terms of some governing equations, one being how velocity and pressure are related; these are calculated using partial differential equations. The Governing equations use the variables time and coordinates (independent) and velocity and pressure (dependent). These equations calculate Continuity, energy and momentum in the x,y and z planes.

The CFD code provided calculates the pressure and velocity of a 2D flow which writes to a binary file containing the values of the solution. When provided with a binary file this program calculates the next 2.1s of the flow, meaning if you keep providing it with the binary file that it outputs it will show the flow over a longer period of time. Shown below at the time-steps 1,3,5,7,9 fig(1).

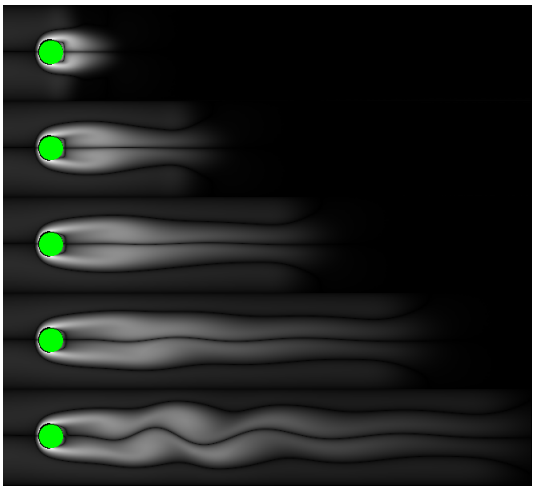

Fig. 1. CFD Flow.

The aim of this report is to explain how to parallelize this code so that it no longer runs sequentially and to make it as computationally efficient as possible. When calculating these values a 2D grid is used to represent the space the computation is being executed within. This 2D grid has cells which communicate with each other and use their neighbouring vertical and horizontal properties to update their own, this is called the communication pattern. There are two types of cell, C_F (fluid) and C_B (obstacle). Each cell has four distinct properties, u (x-velocity), v (y-velocity), p (fluid pressure and flag) and cell type. This program iterates through initializing cell values, checking they satisfy the governing equations and then if not create a new solution based off neighbouring cells.

Successive over-relaxation (SOR) is used as a method to generate new solutions; it is used to speed up convergence. It does this by calculating a current cell from its neighbours, this creates a chain of dependencies for every cell; which is appropriate for a sequential program but not a parallel one due to the work being unbalanced. This unbalaned work stems from using a different number of cells to compute step to step. This method was implemented in solution.c as a Red/Black SOR which splits the cells into two populations (red and black). The red cells are then used to calculate the black cells and vice versa. This means the stencil for a cell is itself and cells of the other population meaning they can be updated at the same time. The two main benefits of this implementation is that work has less communications and can be balanced between processors.

This Red/Black SOR gives the ability for parallelisation to take place as a processor can then have their own section of cells to update. As each partition is updated the parallelisation is going to be within the communication between the bordering cells at the end of each iteration and making sure that the correct data is passed both ways. The synchronisation of the cell values will be vital to keep the whole program on track.
The Red/Black SOR is within a function called poisson, this is were most of the parallelisation will take place due to the information gathered from the profiler (fig 2).

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 94.79     22.47    22.47                             poisson
  4.47     23.53     1.06                             compute_tentative_velocity
  0.51     23.65     0.12                             compute_rhs
  0.17     23.69     0.04                             set_timestep_interval
  0.13     23.72     0.03                             update_velocity
```

Fig. 2. Original Loop Processing Percentages.

MPI is a parallel programming model and will be implemented to try and make this cfd program more efficient. It primarily works by creating a certain amount of processes which communicate with each other by sending and receiving messages. MPI is a specification meaning it tells the processes what to share with each other instead of telling them how. MPI uses distributed-memory and is low level control allowing the data specifics (partition, communication and allocation) to be implemented by programmers. Firstly an initial communicator needs to be defined by using MPI_COMM_WORLD, when initialising MPI size and rank at the start of each file.

After the initialisation of the communicator MPI has many functions that can be implemented to carry out different types of message passing using different parameters. The main ones used in this parallelisation of the cfd code are send/recieve, reduce, all reduce and all gather. These four functions between them are used to parallelise the code and synchronise it at each time step making certain cfd is run correctly and more efficiently.

The main couple of variants which will cause a big difference in performance are the amount of processes and the size of the flow being outputted. This effects performance because of the amount of partitions and how much each one has to compute. In this report a test will be carried out to examine the trend between processes and the speed of execution with a normal sized flow; and a scaled up version which should show the trend more clearly. An introduction of OpenMP (Open specification for Multi-processing) will also hopefully make the parallelisation even more efficient within this code. OpenMP assists the compiler in understanding the serial program, it does this by dividing a program into individual tasks each are executed concurrently by different threads. These threads communicate through shared global data. OpenMP will be explained and examined more thoroughly at a later point.

II. Methods

The steps taken to implement this parallelisation had to be very methodical, starting from understanding how the cfd code works all the way through to data analysis this section will explain the steps take to complete the methodology of parallelisation using MPI and OpenMP.

A. Preperation

Firstly, a profile of the cfd code was taken after running karman.c in the tinis cluster (allowing the request of a certain number of nodes and tasks). The profiler used was gprof which gathers and orders data on your programs by timing each of your programs functions [2] and gave the results shown in figure 2. Proving that the loop with by far the most computation time was poisson. This was the loop that was needing to be paralleised to speed up this program dramatically seeming as it took up 94.79% of the total time. The next logical step was implementing timers to output the time taken to complete certain loops which would indicate if the program was improving or not. The decision was made to use the MPI timers which were placed around the poisson loop and the entire main loop in karman.c to give the times of each poisson loop and each main loop which could be used in concordance with a counter to figure out averages of these same variables. To correctly implement these timers first MPI had to be included and the MPI communicator needed to be initiated using the code snippets below respectively.

```
#include <omp.h>

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&proc);
```

Once these two pieces of code had been put into the correct places, being the top for #include and just before the start of the main loop for the initialisation; the timers and the other MPI functions could then be implemented. The timers were implemented using:

```
startt = MPI_Wtime();
endt = MPI_Wtime();
```

which when placed around a loop would provide start and end time-stamps to allow the calculation of the total time taken; further statistics using this information. Once timers were in appropriate places the parallelisation could be attempted.

B. MPI Parallelisation

There are four main MPI functions used in this parallelisation, send/receive, reduce, all reduce and all gather. This section will provide a breakdown of where and why these were implemented.
Firstly, send/receive. This was used at the end of Red/Black in order to send and receive the values from the borders of the other partitions. Carried out by the code shown in fig(3).

```
//send /receive ileft and iright respectively to their neighbouring sections. As defined datatype ftype to...
if(ileft > 1){

   MPI_Send(p[ileft] + 1 + (((ileft+1) % 2) ^ rb), 1, ftype, proc-1, 666, MPI_COMM_WORLD);
   MPI_Recv(p[ileft - 1] + 1 + (((ileft+1) % 2 ) ^ rb ^ 1), 1, ftype,  proc-1, 666, MPI_COMM_WORLD,&status);
}
if(iright < imax){

   MPI_Send(p[iright] + 1 + (((iright +1) % 2) ^ rb), 1, ftype, proc+1, 666, MPI_COMM_WORLD);
   MPI_Recv(p[iright + 1] +  1 + (((iright+1) % 2 ) ^ rb ^ 1), 1, ftype,  proc+1, 666, MPI_COMM_WORLD,&status);
}
      } /* end of rb */
```

Fig. 3. Send/Receive.

This was done using a derived datatype, ftype; which sends and receives the data as every other value. This matches up with the way Red/Black works updating values based off every other piece of data for the opposing colour. This in turn allows the Red/Black SOR to be carried out efficiently instead of sending all the data from the border, this send/receive with the ftype defined in the code halves the data transfer. The

code to implement this data type is in fig(4). Showing how it is constructed.

```
//Define own datatype ftype to which halfs the data transfer
MPI_Status status;
MPI_Datatype ftype;
MPI_Type_vector(jmax/2, 1, 2, MPI_FLOAT, &ftype);
MPI_Type_commit(&ftype);
```

Fig. 4. Derived Data Type.

Secondly, All_reduce. This is a function that is used in a couple of places within the code which in essence, reduces sets of sets of numbers into a value based on the operation carried out. In both cases in simulation.c where it is used a sum is carried out taking a specified number of sets of numbers and summing them so they can be used, this is used to synchronise the p0 and res values, respectively. These have been used instead of broadcast and gather; another two functions which in this situation are not worth implementing over all reduce due to complexity reasons fig(5).

```
MPI_Allreduce(&p0, &tot, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);

MPI_Allreduce(res, &tot, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

Fig. 5. All Reduce.

These two values, p0 and res are then used in calculations to help the program determine its states; for example res is used to help determine whether the program has converged or not.

MPI_AllGather takes in values from many processes and can distribute these across all processes synchronising the current state of the grid of cells that is being calculated. This is done through all gathers' many-to-many communication that it implements. In this specific case in karman.c it takes in all the p[ileft] values and passes them to all other processes so that all partitions know exactly what is going on. This makes sure that all data is the same at certain time steps keeping all processes on track and making certain they can stay in tune with each other. This overcame a problem which occurred being one process would finish and terminate and therefore not pass required data to the other meaning it never finished. This was done simply by using the line of code shown fig(6).

```
//gather the ileft values and populate p2 with them
MPI_Allgather(p[ileft], (imax/nprocs) * (jmax + 2), MPI_FLOAT, p2[1], (imax/nprocs) * (jmax + 2), MPI_FLOAT, MPI_COMM_WORLD);
```

Fig. 6. All Gather.

The last MPI function that is used is reduce which is a simpler version of reduce all where it only takes in one set of numbers. This is used in karman.c within the main loop.

These four functions worked in unison to parallelise and synchronise the program well enough so when the scale of the cfd was increased by x2 it still carried out the program efficiently; this made the dimensions 1320x240. At this point batch scripts were made to speed up the use of tinis, allowing the user to define an amount of nodes and tasks per node and to run a specific time-step. All of the time-steps prior to 9 had to be gathered so that 9 could be implemented each time.

C. OpenMP Parallelisation

This section will explain how OpenMP was implemented to hopefully improve the efficiency of karman to be able to handle the bigger scale to a better standard. This implementation created an MPI-OpenMP hybrid which should be useful.

OpenMP can be implemneted on a sequential loop and automatically partitions the data and allocates it to a certain number of threads defined by the programmer. This could only be implemented in one logical position within the cfd code; this position was in simulation.c. The Red/Black SOR calculations could be split into different threads which should make the computation faster, especially on a larger scale as each partition has more of these calculations to carry out. The most logical way of implementing this was with the line in fig(7).

```
//OpenMP code for static parallelisation of the for loop
        #pragma omp parallel for schedule(static)
```

Fig. 7. OpenMP Static.

To implement this correctly an OpenMP package must be included as such:

<omp . h>

Once all of this had been debugged and was working well, the data collection and analysis could then begin taking timings of the poisson/main loops with different amounts of nodes, tasks per node and when OpenMP was involved CPUs. These would be tested for big and small scalings and will be profiled at each stage to show the percentages of the loops improving. The overhead will be analysed as well, hopefully to reason why certain techniques work well or badly based off the data analytics which will take place. All averages in the data below were calculated through 5 sets of the data to give accurate and reliable data. The time-step that will be tested is 10 as this is the one of the more interesting steps graphically. Figure 8 shows the scaling difference between big and small for this specific time-step.

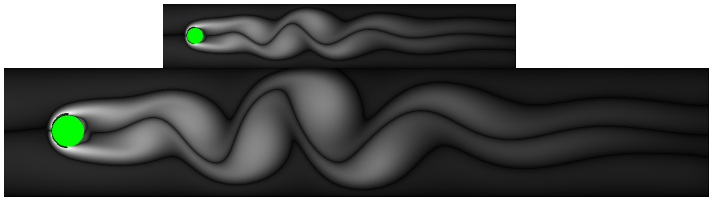

Fig. 8. Timestep 9: Big Vs Small.

III. Results

This section will describe the results in each of the figures.

Figure 9 shows the base timings for the original karman file with just the timers in allowing a comparison of the improvement MPI brought the 'small' program.

Base Average Poisson Loop Time	0.113361
Base Average Main Loop	0.119819
Base Total Poisson Time	23.9191
Base Total Time	25.2798

Fig. 9. Base Timings.

The original timings for testing Nodes against Tasks in figure 10 shows the worst (red) and the best (green) for each node set. As you can see the original scale; which from here on out will be refered to as small, slowly gets worse the more nodes and tasks there are. Tasks and nodes multiply together to make the amount of processes being used on the current environment.

ORIGINAL					
Nodes	Poission Loop Average	Main Loop Average	Program Execution Total	Tasks	Processes
1	0.110115	0.116334	24.5464	1	1
	0.0584073	0.0646815	13.6478	2	2
	0.0445224	0.0515013	10.8668	3	3
	0.0335287	0.0403031	8.50396	4	4
	0.0281238	0.035455	7.481	5	5
	0.0227649	0.0295904	6.24357	6	6
	0.0208498	0.0281906	5.94822	10	10
	0.0165094	0.0239094	5.04489	11	11
	0.016097	0.0235188	4.96246	12	12
	0.0148296	0.0224385	4.73452	15	15
2	0.0581669	0.0644918	13.6078	1	2
	0.0337757	0.0401909	8.48028	2	4
	0.027053	0.0338592	7.14429	3	6
	0.0217394	0.0279323	5.47473	4	8
	0.0243644	0.0314925	6.64491	5	10
	0.0185049	0.0258875	5.46437	6	12
	0.0570016	0.0644093	13.5904	10	20
	0.0569419	0.0643889	13.5861	11	22
	0.0166261	0.0232161	4.68966	12	24
	0.0323492	0.0399267	8.42454	15	30
3	0.0449624	0.0513473	10.8343	1	3
	0.0255215	0.0320957	6.7722	2	6
	0.023063	0.0294551	5.83211	3	9
	0.0183719	0.0256875	5.42005	4	12
	0.0189772	0.0262838	5.54587	5	15
	0.0699757	0.0764353	15.5164	6	18
	0.0235113	0.0309927	6.53946	10	30
	0.0230218	0.030497	6.43487	11	33
	0.0219222	0.0283131	5.71924	12	36
	0.0291814	0.0358209	7.0209	15	45
4	0.0368975	0.0431783	9.11063	1	4
	0.025735	0.0315916	6.19195	2	8
	0.0584842	0.0653918	13.7977	3	12
	0.019547	0.0255976	5.01713	4	16
	0.0777412	0.0852046	17.9782	5	20
	0.0952444	0.101409	20.4845	6	24
	0.0521074	0.0586145	11.5471	10	40
	0.17371	0.181167	38.2263	11	44
	0.0308461	0.0374275	7.37321	12	48
	0.0294369	0.0369518	7.79683	15	60

Fig. 10. Original cfd Data.

The small average poisson loop times in figure 11 show the clear decrease in performance due to the increase of processes, meaning there is a negative correlation at this stage.

SMALL				
1	2	3	4	
0.110115	0.0581669	0.0449624	0.0368975	1
0.0584073	0.0337757	0.0255215	0.025735	2
0.0445224	0.027053	0.023063	0.0584842	3
0.0335287	0.0217394	0.0183719	0.019547	4
0.0281238	0.0243644	0.0189772	0.0777412	5
0.0227649	0.0185049	0.0699757	0.0952444	6
0.0208498	0.0570016	0.0235113	0.0521074	10
0.0165094	0.0569419	0.0230218	0.17371	11
0.016097	0.0166261	0.0219222	0.0308461	12
0.0148296	0.0323492	0.0291814	0.0294369	15

Fig. 11. Original cfd Average Poisson Times: Nodes Vs Tasks.

The Big timings for Nodes against Tasks in figure 12 show that the amount of processes does help the efficiency but starts to decline when the number of processes exceeds 30.

BIG					
Nodes	Poission Loop Average	Main Loop Average	Program Execution Total	Tasks	Processes
1	0.439775	0.464875	214.307	1	1
	0.238018	0.264577	121.97	2	2
	0.160963	0.187568	86.4689	3	3
	0.126061	0.153569	70.7952	4	4
	0.106725	0.135321	62.3832	5	5
	0.098571	0.129155	59.5403	6	6
	0.071378	0.10218	47.1049	10	10
	0.0697851	0.0998986	46.0532	11	11
	0.0571307	0.0871096	40.1575	12	12
	0.0487449	0.0793517	36.5811	15	15
2	0.236489	0.262309	120.924	1	2
	0.12518	0.151653	69.9122	2	4
	0.091906	0.121743	56.1235	3	6
	0.0923941	0.1195	55.0897	4	8
	0.0792853	0.109098	50.2943	5	10
	0.0734178	0.103572	47.7465	6	12
	0.0668519	0.0974798	44.9382	10	20
	0.0886412	0.119297	54.996	11	22
	0.0375998	0.0676508	31.187	12	24
	0.034642	0.0653105	30.1081	15	30
3	0.170149	0.196542	90.606	1	3
	0.0886951	0.114839	52.9408	2	6
	0.0707385	0.09318	42.8628	3	9
	0.0953789	0.12302	56.7123	4	12
	0.070611	0.099481	45.8607	5	15
	0.058392	0.0822003	37.8122	6	18
	0.0388998	0.0693162	31.9548	10	30
	0.048092	0.0782025	36.0514	11	33
	0.0408071	0.0659667	30.0149	12	36
	0.0471281	0.0730213	33.2247	15	45
4	0.130452	0.158337	72.0712	1	4
	0.0775673	0.104609	48.2249	2	8
	0.0610154	0.0888638	40.9662	3	12
	0.0530053	0.0772035	35.668	4	16
	0.0468322	0.0758762	34.9789	5	20
	0.0399084	0.0699191	32.2327	6	24
	0.0424955	0.072623	33.4792	10	40
	0.0542031	0.0841108	38.7751	11	44
	0.0558152	0.0810568	36.8808	12	48
	0.0424892	0.0731318	33.7138	15	60

Fig. 12. Big cfd Data.

Figure 13 shows that the fastest combination of Nodes and tasks at this stage is 2 nodes 15 tasks, this totals up to 30 processes and in general this data shows a positive correlation.

BIG				
1	2	3	4	
0.439775	0.236489	0.170149	0.130452	1
0.238018	0.12518	0.0886951	0.0775673	2
0.160963	0.091906	0.0707385	0.0610154	3
0.126061	0.0923941	0.0953789	0.0530053	4
0.106725	0.0792853	0.070611	0.0468322	5
0.098571	0.0734178	0.058392	0.0399084	6
0.071378	0.0668519	0.0388998	0.0424955	10
0.0697851	0.0886412	0.048092	0.0542031	11
0.0571307	0.0375998	0.0408071	0.0558152	12
0.0487449	0.034642	0.0471281	0.0424892	15

Fig. 13. Big cfd Average Poisson Times: Nodes Vs Tasks.

This scatter graph (figure 14) clearly shows the improvement across the 4 nodes in time especially in the lower amount of tasks whereas in the higher number of tasks it becomes much harder to interpret. It also shows a plateau in performance.

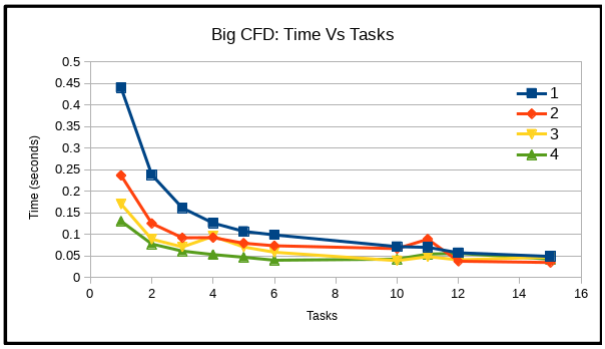

Fig. 14. Big cfd: Time Vs Tasks.

The profiler at this stage measures the performance without OpenMP with the big karman file. It clearly shows that the poisson loop now is no longer the main issue with efficiency and has been reduced by more that 50%. compute_tentative_velocity on the other hand is now the new most computationaly costly.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 50.35     11.61    11.61                             compute_tentative_velocity
 37.50     20.25     8.64                             poisson
  5.43     21.50     1.25                             compute_rhs
  4.25     22.48     0.98                             update_velocity
  1.74     22.88     0.40                             set_timestep_interval
  0.78     23.06     0.18                             apply_boundary_conditions
```

Fig. 15. Big cfd Performance Statistics.

With OpenMP implemented figure 16's data shows how the amount of CPUs affects the runtime of the program, with it being faster at 1 CPU for 1 and 2 nodes implying that it is faster without it. It also shows that 4 CPUs is the slowest in both cases for the hybrid approach.

1 NODES				
CPUs	Poission Loop Average	Main Loop Average	Program Execution Total	Processes
1	0.302346	0.329815	152.045	2
2	0.607188	0.635281	292.864	2
3	0.836709	0.866631	399.517	2
4	1.26267	1.28909	594.27	2
5	1.20761	1.23622	569.897	2
6	0.731237	0.761279	350.95	2
8	1.08256	1.11253	512.877	2

2 NODES				
CPUs	Poission Loop Average	Main Loop Average	Program Execution Total	Processes
1	0.157408	0.182972	84.3502	4
2	0.390567	0.419531	193.404	4
3	0.657951	0.68702	316.716	4
4	0.808151	0.837606	386.137	4
5	0.361967	0.390893	180.202	4
6	0.707842	0.737138	339.821	4
8	0.781953	0.81195	374.309	4

Fig. 16. OpenMP cfd: CPUs Vs Nodes .

The profiler in figure 17 takes performance statistics on the big karman file with the hybrid MPI-OpenMP implemented and does show an even bigger reduction in the poisson loop, with an unexpected spike in red_bin for reasons unknown.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 47.12     11.55    11.55                             compute_tentative_velocity
 29.58     18.80     7.25                             read_bin
 12.04     21.75     2.95                             poisson
  5.10     23.00     1.25                             compute_rhs
  3.88     23.95     0.95                             update_velocity
  1.35     24.28     0.33                             set_timestep_interval
  0.94     24.51     0.23                             apply_boundary_conditions
```

Fig. 17. Big OMP cfd Performance Statistics.

Figure 18 depicts the difference between the fastest combinations between 1 and 2 nodes with OpenMP and the big karman.c showing that OpenMP actually slows down the program massively at this size.

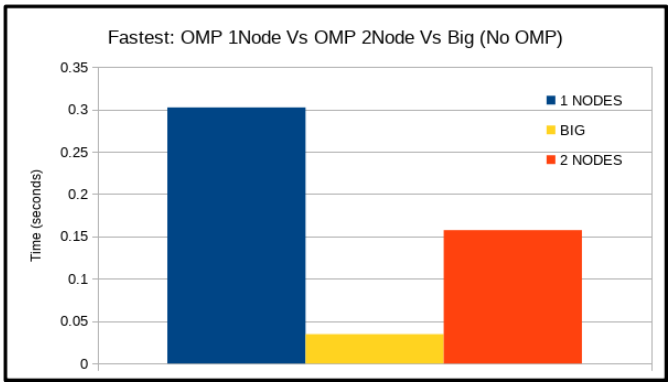

Fig. 18. OMP cfd 2 Nodes.

This data describes that MPI has improved the overall performance even with a larger scaled project. However, a hybrid MPI-OpenMP slows it down at this size. The data in this section does show some interesting trends within the data in between stages of performance measurements.

IV. Discussion

The discussion will explain the material provided in the results section and will explain the reasoning behind the individual figures. It will make some conclusions to help with the understanding and reasoning, which will bring further insight. The discussion path will follow the methodology and go through the figures one by one.

Before we dive into the deeper discussions, it is quite clear to see between figure 9 and 10/11 that the MPI did improve the efficiency of the program dramatically. The base poisson loop took an average of 0.113361s compared to the parallelised average at 0.0148296. Firstly figure 10 shows the small scaled karman results and the worst and best results from each set of Nodes against tasks shows that the performance decreases overtime. This is likely due to overhead. Overhead is the excess time required to coordinate parallel tasks and communicate information between processors, degrades scalability [2]. The overhead on such a small scaled project becomes a bigger problem which is the reason why the best and worst amount of nodes swaps over after 2 nodes meaning the more nodes the more overhead which in turn is slowing down the total run-time of

the program. This is further proved due to figure 11 where you can see the swap even more clearly. Secondly, figure 12 shows the same data except this is for the big karman where the trend of more processes, the slower the program runs does still occur; in contrast it is shown much less. This is almost certainly to do with overhead again and is much less due to the scale of the program. At a bigger scale the overhead has less of an effect on the time taken to perform each loop, this leads to the belief that at an even bigger scale if one was attempted the more overhead would have even less of an affect proportionally. This again is easier to see in figure 13 due to only showing the average poisson run times.

Figure 14 on the other hand shows that in general the more tasks the better the performance is. This data is useful for showing the plateau of performance that occurs when the processes reaches a certain level, and does show that 2 is faster than 4. This data is hard to interpret and can lead the reader to the wrong conclusion. The plateau is the interesting bit of information to take from this, the confusion between which is faster is clarified in other figures. The profiler in figure 15 does show a massive improvement which is good and shows that the MPI parallelisation was a success, the next problem that occurs is that other areas of the program are now slowing it down. This could have been fixed if parallelisation had been implemented in all areas of the program however for a program thats scale is this small that would have meant an even bigger overhead slowing it even further. This balance was a hard decision of which path to take. If the problem domain would have been bigger the next logical step would have been to parallelise all other loops in the program so that they share data constantly instead of synchronising at small time steps.

OpenMP was alot easier to implement than MPI, it is a very different method of parallelisation and comes with its own challenges. The main issue to discuss in this situation is it slowed up the program dramatically. Figure 16 shows the best and worst times for each set of CPU's against 1 and 2 nodes. The one interesting trend between this data is with 2 nodes the time halves. This makes sense as there is half as many resources for each to compute. This also leads to the conclusion that with 4 Nodes it will half again, and 8 again. This is a bigger problem and the exploration into whether this was the case was stunted due to the limitations of the cluster tinis which only allowed 4 nodes to be requested. The only other parameter that could have been changed to improve the OpenMP performance within this program would be the scale, as at a much higher scale the statically divided partitions of the Red/Black loop would be sped up. At this scale it is not worth the implementation due to the overhead of creating the partitions and passing the data in and out of the threads to be calculated. Figure 17 is the profiler showing the performance percentages of the loops whilst OpenMP is implemented and althought it does seem to improve poisson, this is not the case. For an unknown reason read_bin's execution time is dramatically increased and as it should not be calculating that much there is an error here that has an undetermined source. Finally, figure 18 reiterates just how slow OpenMP makes the program by comparing the fastest time for 1 and 2 Nodes against the fastest big strictly MPI execution. The big MPI execution is nearly three times faster. The exact calculations to why this is is hard to determine.

V. Conclusion

In Conclusion the parallelisation of the cfd code was successful. The implementation of MPI functions sent the right data between processes and synchronised this at appropriate times. A few simple timers gave accurate data to be able to test just how parallelised the data was. MPI sped up the program in general, however the number of nodes and tasks had a big impact on its performance. More obscure MPI functions were used to try and improve the efficiency of the data transfer as much as possible. The more processes used when running the small karman file meant a massive amount of overhead proportionally. The fastest parameter composition for small karman was 1 node and 15 tasks per node. Once this problem domain was scaled up by x2 the overhead became smaller; the data showed this by having the amount of overhead being proportionally smaller. The fastest parameter composition for big karman was 2 nodes and 15 tasks per node which took the poisson on average 0.034642s to complete. The profilers did prove that the poisson loop was being sped up by the code that was implemented, and gve us enough information to be able to spur on interest into what to work on next. The performance of karman was hindered when OpenMP was introduced due to the nature of the overhead it brought with it on such a small scale program, this hybrid approach was not as efficient as predicted.

If there was more time provided for this project an investigation would have taken place into the parallelistation of all data at all times rather than small incremented time step synchronisations would have occurred. This, in turn would have allowed the program to run at an even bigger scale most likely making OpenMP more and more useful in terms of speeding up the execution time. Another area to investigate would be into a different cluster to provide more nodes to allow the testing to be much more thorough.

References

[1] S. Wolfman, "Profiling with GProf", Users.cs.duke.edu, 2019. [Online]. Available: https://users.cs.duke.edu/ ola/courses/programming/gprof.html. [Accessed: 04- Feb- 2019].

[2] G. G. Howes, Introduction to High Performance Computing. University of Iowa, 2019.