CS402 High Performance Computing
Assignment 2: MPI with karman code

Michael Boyle
1403116

March 2018

Introduction

This coursework deals with implementing Memory Passing Interface commands for the fluid dynamic karman code. Due to the lack of shared memory between MPI 'nodes' it is possible to drastically improve spacial locality for the program (due to lesser data storage) as well as utilize processor cache more efficiently.

Code Breakdown

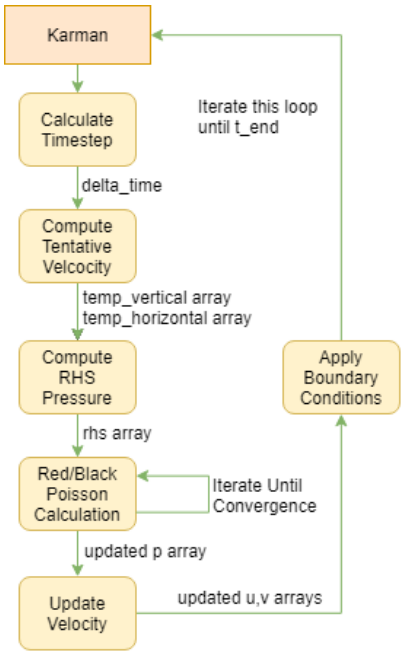

Figure 1: The karman code performs this loop, with dependencies between each method that is executed

The karman code takes a .bin file representing a 2D cell array of moving fluids and steps forward in computed time-steps, taking an iterative approach to converge the motion of each fluid cell. Figure 1 shows how each of methods that are executed in the time loop are dependent on the previous loop and the most recently processed function. As such, it is not possible to parallelize the code between time steps, so each MPI node will instead need to be allocated a section of the 2 dimensional cell array. In order to maximise throughput, these sections should be as close to equal as possible so that each node has a similar workload and will finish each task at around the same time. There will need to be communication between the nodes performed at regular points, as many of the functions require information on the position of nearby cells.

By gprofiling the base code, a distribution of the timings for the various functions shows that 94.98% of the total time is spent in the Poisson function to calculate the pressure in each cell. As is shown in Figure 1, this function iterates on pressure values until the net pressure on each cell (squared) summed together is below a small epsilon value. This function is an example of MPI communication being required internally since there needs to be communication during every iteration towards convergence, as well as the summed pressure being shared across all nodes datasets.

The decomposition strategy used for this MPI implementation was one dimensional, dividing the program into vertical section as is shown in Figure 2. Vertical decomposition is useful as a scalable solution, allowing for n-way MPI processing with similar workloads for each processor, unlike 2D decomposition which only works explicitly well for factorisable values (will not be able to utilize 3, 5, 7... etc nodes in a balanced way). Vertical 1D decomposition also benefits from how the cells are stored (rather oddly) in columns instead of rows. This reduces transmission times over 2D as data for a given row does not need to be stored in a new separate array before transmission.

Explicitly for the example bin we are working with, this also results in fewer transfers of data. Vertical 1D decomposition requires $2 \times (jmax+2) \times (nodes-2) + (jmax+2) \times (2) = 2 \times (jmax+2) \times (nodes-1)$, whereas a 4 way decomposition would require $4 \times (\frac{jmax+2}{2} + \frac{imax+2}{2}) = 2(jmax + imax + 4)$. With imax

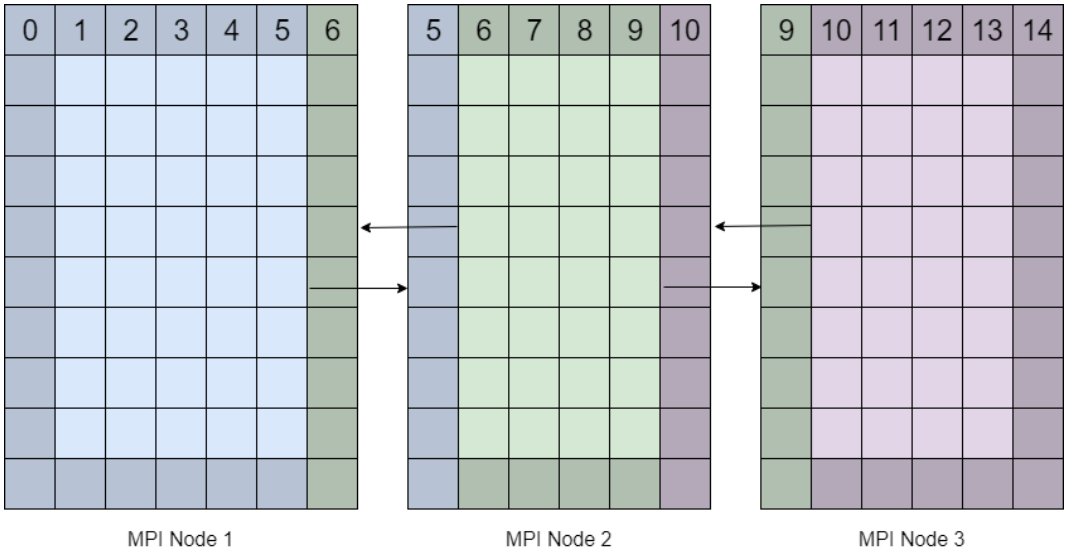

Figure 2: A three way decomposition for an imax value of 13 produces a slightly heavier load for node 1, but balances the overall task quite well. Only have to communicate the columns between nodes.

being somewhere in the range of 5 times larger than jmax, this means 2D decomposition would result in nearly twice as much communication being required for 2D decomposition (without even factoring in the cost of collating rows for transmission).

There is the potential for the first node to be larger due to all but the root nodes sizes being calculated as $\lfloor \frac{imax}{size} \rfloor$ whereas the root node is calculated as $imax - ((size - 1) \times \lfloor \frac{imax}{size} \rfloor)$. While this may result in a disproportionate amount of nodes for the root node to deal with, this is balanced by the root node only having to communicate along a single boundary. An extension to this implementation would be to balance the load more effectively with the final node, which also only has a single boundary to communicate.

Optimisations

The Poisson Function

The poisson function takes up the majority of the running time within this program due to iterating until a convergence point is reached. Fortunately, it's Red/Black implementation means there is little enough dependency that impressive speedups can be achieved. With alternating p values being changed between Red/Black steps [1], an additional iStartPos parameter can be passed in so that the Red/Black assignment of each cell is correct across each of the nodes.

The newly calculated p values at the boundary can be transferred at each iteration of the Red/Black calculation. The decision to implement this in a non-blocking way helps to avoid the need for communication to cascade along the nodes, although the messages must be waited on for receiving as the p values that are calculated are needed for either the second phase of the Red/Black SOR iteration or the residual calculation. If the messages were exchanged in a blocking way, the information would need to be communicated 'left' to 'right' and then back the other direction (or vice versa) however this could mean that any of the nodes was held up on a single node finishing it's execution.

The residual calculation also serves as a demonstration of how the MPI_Reduce function [2,3] can be used, as shown in Listing 1. Since this *res* calculation decides whether the SOR iteration continues, it is essential that consensus is reached by all nodes simultaneously, which the redistribution of the res value helps with.

```
MPI_Reduce(res, &resTot, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
        *res = resTot;
        *res = sqrt((*res)/ifull)/p0;
```

```
        int node;
        // Send back out to the rest of the nodes
        for (node = 1; node < size; node++) {
                MPI_Send(res, 1, MPI_FLOAT, node, tag, MPI_COMM_WORLD);
        }
} else {
        MPI_Recv(res, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &stat);
}
```

Listing 1: MPI reduction of the residual pressure and redistribution to the other nodes

Boundary Conditions

Applying the boundary conditions is one of the few functions that can not be sped up by a large amount. One of the drawbacks of Vertical 1D decomposition is that the dependency on previously calculated values can not be avoided. Two dimensional decomposition would have the benefit of branching after the first, North Western node had completed it's boundary conditions, allowing for the southern and eastern nodes to continue in parallel. Unfortunately, this function must execute linearly, requiring each of the previous i values to have been calculated at a previous point, however it is one of the quickest functions to executes since it executes minimal amounts of instructions.

Figure 3: Shows how the boundary information for a 4 way MPI implementation can become erroneous if the dependency is not met.

As mentioned in the Code Breakdown section, a 1D decomposition has the benefit of needing less pre-processing before messages can be transmitted. For the most part across the karman code, only a single array (of u, v, f, g or p) needs to be passed between the different nodes, however the boundary conditions code requires the transfer of u, v and p arrays, which correspond to what is written to the bin file at the end of execution and what is needed for the start of the next loop (if it occurs). Since these arrays needed to be sent to the previous node asynchronously to stop the program halting, these values needed to be stored in a single array to avoid holding up the application buffer [4] and to reduce the amount of messages that are actually sent. This also demonstrates how receiving can be set up on the previous node in a non-blocking fashion.

The boundary code does demonstrate how a blocking receive before the start of the loop can be used to enforce serial execution across the seperate MPI nodes. Figure 3 shows what happens when these linear dependencies are not properly met resulting in inconsistencies with the u and v values stored at the boundary.

Testing

With MPI implemented, Figure 4 shows that a near linear speedup can be achieved over the original sequential code.

Testing the range of functionality with the MPI implementation was simpler with an extensive number of *msub* scripts which could be submitted to the computing cluster [5]. While the poisson function still takes up the majority of the running time at 93.44%, the total running time can be drastically reduced, even when MPI is used on other processors in the same node.

MPI_Barrier commands [6] were also an essential element to make sure that all MPI nodes reach the same point before executing subsequent methods, although it was rare that they were a necessity. They

MPI	Timestep	Ten-Velocity	RHS	Poisson	Upd-Velocity	Boundaries	Runtime	Speedup
0	0.000197	0.005320	0.000246	0.112266	0.000246	N/A	24.01	N/A
1	0.000156	0.005163	0.000528	0.109738	0.000366	0.000079	23.567032	1.87%
2	0.000084	0.002623	0.000264	0.056296	0.000184	0.000077	12.094268	98.52%
4	0.000050	0.001162	0.000132	0.029487	0.000092	0.000092	6.315580	280.17%
20	0.000041	0.000277	0.000032	0.008630	0.000023	0.000136	1.874967	1180.56%

Figure 4: Breakdown of Average Running Time for the initial.bin across the main loops which run. The row with 0 MPI levels is for the base serial code, calculated with gprofiling, without any MPI directives included in the code.

instead functioned primarily as debugging points during development, making sure that all code had been correctly executed up until specific breakpoints.

Version control [7] also helped to test the speedup between separate iterations of development and compare the complexity between the original code and the updated MPI implementation.

Single Read versus Parallel Read

In testing, it was found that having each node read the initial bin file instead of only the root node resulting in a slightly longer running time. While this may be possible to optimize further (the read on non-root nodes occurs later in the handshake) there are a few reasons why a single read may be preferred over a read on every node.

1. Reads from a disk can take longer than transmitting messages, especially on a cluster of machines designed to message quickly
2. A single read means the data only needs to be stored in a single location, allowing for some nodes on the architecture to be purely computation and communication based
3. Making the read an atomic action improves data integrity and security, as there is no way various nodes can have different data
4. Produces a symmetric effect of reading out to nodes at startup and reading into the root node at the end
5. Only root node ever needs enough space for the entire file, other nodes do not consume as much in terms of memory resources
6. Only have to read the data once, program will not halt function if a node can't read the file

The code for parallel reads can be found in *kamanSeperateReads.c.unused*.

OpenMP Hybrid Approach

Incorporating OpenMP directives for the MPI code was relatively simple, as many of the existing cases of communication were good indicators for when particular directives were needed. Static scheduling could be used across the board as there was never a time when execution was different within a loop (outside of a few small special cases). The main methods which required special consideration were poisson and timestep calculations, since they both made use of reduction in both MPI and OMP.

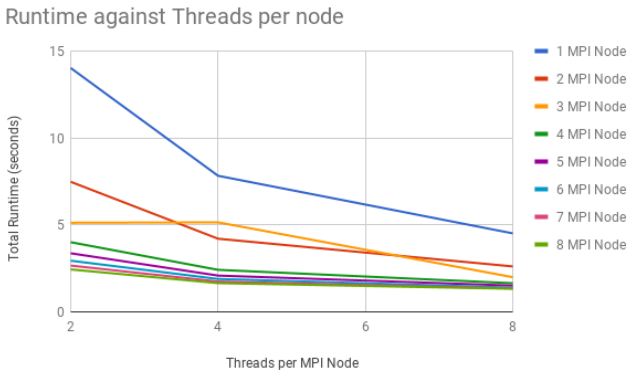

Figure 5: Shows how the program speeds up given a higher thread count across the various levels of MPI

Fortunately both reductions [8] have a similar function in OMP directives.

As with the MPI implementation, the boundary conditions dependency has resulted in a lack of parallelism when applying these conditions. As in Figure 4, the boundaries function takes a larger percentage of the total runtime of the code.

As Figure 5 shows, the speedup when increasing the number of threads is far less than the improvement gained via MPI. Partnered together, there is still improvement, but given the choice between a higher degree of MPI or OMP, MPI has the better performance increase.

If the MPI tasks are running on a single thread is it preferable to use the base MPI code in the *karman.c.MPIOnly*, *simulation.c.MPIOnly* and *boundary.c.MPIOnly* code, as these do not contain the overheads for starting the OpenMP threads.

Conclusion

The MPI implementation of the karman code has some excellent performance increase and allows for significantly less memory requirements on the majority of the MPI nodes. By avoiding heavily dependent code and exploiting spatial and temporal locality more often on the individual nodes, near linear speedup can be achieved. While the OpenMP/MPI Hybrid approach produces less overall performance speedup, the implementation is still worth the investment as the execution is faster still.

References

[1] Ligang He. Assignment background and sor iteration. https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs402/assignment2-background.pdf.

[2] Mpi reductions. http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/.

[3] Mpi reduce operations. http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node78.html.

[4] Mpi isend non-blocking send. http://www.mpich.org/static/docs/v3.1/www3/MPI_Isend.html.

[5] Warwick cluster user guide. https://wiki.csc.warwick.ac.uk/twiki/bin/view/Main/ClusterUserGuide#MPI_jobs.

[6] Mpi barrier. https://www.mpich.org/static/docs/v3.2/www3/MPI_Barrier.html.

[7] Assignment version control. https://github.com/Wellwick/cs402.

[8] Openmp reductions. http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-reduction.html.