Parallelisation of karman

CS922 High-Performance Computing - Assignment 2

Introduction

This report details the parallelisation of karman - a CFD (computational fluid dynamics) program which calculates the velocity and pressure of a 2D flow. To achieve parallelisation, both OpenMP and MPI (Message Passing Interface) were used.

Program Breakdown

The karman code takes an input of a binary file, which contains a 2D array of discretised fluid and obstacle cells. The time-step is then calculated by assessing the amount of since the last time-step. Next, the tentative velocity is computed based on the previous values. The resultant horizontal and vertical arrays are then passed through to a function to compute the RHS of the pressure equation. The resultant RHS matrix, alongside the current pressure and flag matrices, is then used to solve the Poisson equation. An iterative approach is used to converge on the motion for each cell. The iteration terminates when the iterative threshold of 0.001 is met. This iterative method is sped up by using Successive Over-Relaxation (SOR), a variation of the Gauss-Seidel method. After this, the true velocity values are updated based on the pressure matrix and tentative velocities. Boundary cells are then scrutinised, so that any cell next to the boundary is updated accordingly. This sequence is then repeated until time = t_end.

Decomposition

One-dimensional (1D) decomposition was used in order to allow for MPI implementation. Figure 1 illustrates the premise of 1D decomposition.

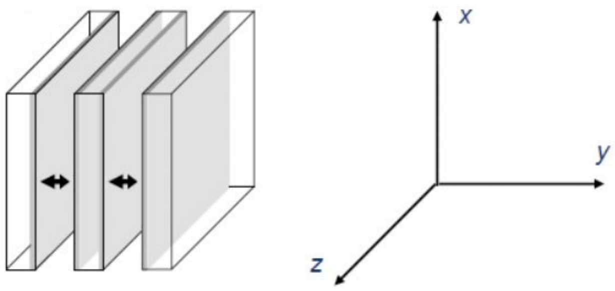


Figure 1: 1D Decomposition

The operations were decomposed vertically. This provides scalability benefits; 1D decomposition is able to handle virtually any number of nodes, compared to 2D decomposition, which would require an even number of nodes in order to run. Where the number of nodes is denoted by *n*, and the size of the original data matrix by *imax* x *jmax*, the number of data transmissions required in 1D decomposition is 2 x (*n* - 2) x (*jmax* + 2) x (*n* - 2) + 2 x (*jmax* + 2), which simplifies to 2(*jmax* + 2)(*n* - 1).

Karman Profile

Running the command `gprof karman > karman_sequential.gprof` gave a profile for the runtimes for each function in the karman code. These runtimes are tabulated in Table 1.

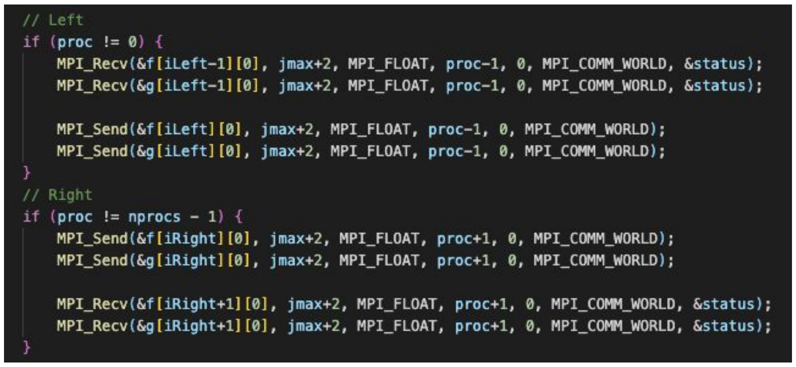
Function	Runtime [s]	Runtime Percentage
`poisson`	22.58	95.03
`compute_tentative_velocity`	0.97	0.97
`compute_rhs`	0.12	0.12
`update_velocity`	0.1	0.10
`set_timestep_interval`	0.01	0.04

Table 1: Runtimes of sequential karman functions.

Inspecting the table, it is clear that the majority of the running time amassed from the Poisson function. This provided an informed starting point for parallelisation.

Data Exchange

To facilitate data exchange, the MPI resource was initialised in karman.c. Simulation.c contains the bulk of the code parellisation. Inspecting the `compute_tentative_velocity` function;



```
// Left
if (proc != 0) {
    MPI_Recv(&f[iLeft-1][0], jmax+2, MPI_FLOAT, proc-1, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&g[iLeft-1][0], jmax+2, MPI_FLOAT, proc-1, 0, MPI_COMM_WORLD, &status);

    MPI_Send(&f[iLeft][0], jmax+2, MPI_FLOAT, proc-1, 0, MPI_COMM_WORLD);
    MPI_Send(&g[iLeft][0], jmax+2, MPI_FLOAT, proc-1, 0, MPI_COMM_WORLD);
}
// Right
if (proc != nprocs - 1) {
    MPI_Send(&f[iRight][0], jmax+2, MPI_FLOAT, proc+1, 0, MPI_COMM_WORLD);
    MPI_Send(&g[iRight][0], jmax+2, MPI_FLOAT, proc+1, 0, MPI_COMM_WORLD);

    MPI_Recv(&f[iRight+1][0], jmax+2, MPI_FLOAT, proc+1, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&g[iRight+1][0], jmax+2, MPI_FLOAT, proc+1, 0, MPI_COMM_WORLD, &status);
}
```

Figure 2: Message passing logic.

only the left and right extremities of the flow need to be unidirectionally passed. As long as the process in question is not the first or the last, then bidirectional passing is permitted. This is seen in Figure 2. All other functions in simulation.c, bar `set_timestep_interval`, feature this code too.

In `poisson`, the pressure computation is parallelised, therefore MPI_Allreduce was used to sum the separate pressure calculations. The Allreduce function was also used to compute umax and vmax in `set_timestep_interval`, and the residual computation in `poisson`.

Runtime Analysis

Effects of Parallelisation

The runtimes of code with various levels of parallelisation is documented in Table 2. The runtimes for the two parallel implementations were obtained by running the program with a node count of 2.

Function	**Runtime [s]**		
	Sequential	**MPI**	**MPI + OpenMP**
`poisson`	22.58	12.20	12.15
`compute_tentative_velocity`	0.97	12.67	12.66
`compute_rhs`	0.12	12.75	12.75
`update_velocity`	0.10	12.79	12.78
`set_timestep_interval`	0.01	12.80	12.79

Table 2: karman function runtimes.

Inspecting Table 2 shows that parallelising the code makes a sizable speedup of almost 100%. It is worthy to note that the runtimes between the MPI-only and the hybrid MPI-OpenMP implementations result in minimal runtime difference, and it is not yet conclusive whether a hybrid implementation makes a notable improvement. There are overheads associated with adding OpenMP directives to the code, therefore the runtime difference should be noticable when karman is supplied with a problem size bigger than the default imax = 660, jmax = 120.

Increased Problem-size

To test this hypothesis, the problem-size was increased - imax and jmax were doubled (to 1320 and 240, respectively). The `poisson` runtimes across the various implementations are recorded in Table 3.

Implementation	Runtime [s]
Sequential	262.24
MPI	270.22
MPI + OpenMP	267.07

Table 3: `poisson` function runtimes for the larger problem-size.

It is clear that adding parallelisation at this larger problem size does not necessarily better the performance of karman. However, it is interesting to note that between the two parallelised implementations, the hybrid solution boasts a 3.15 second improvement. This determines that when the program operation time is sufficiently long, the hybrid solution is a better candidate than the MPI-only solution.

Node Count

The runtimes of `poisson` for different node counts are illustrated in Figure 3.

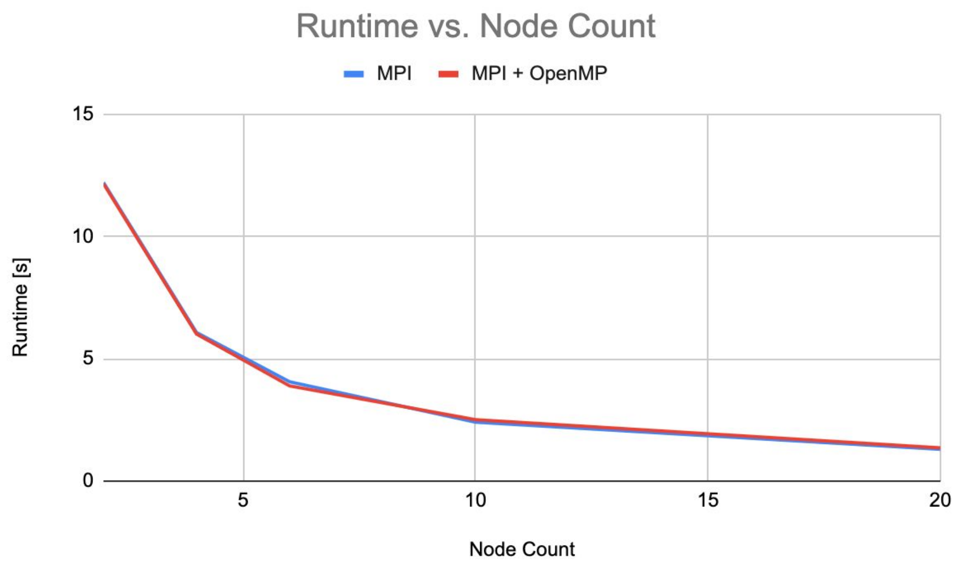


Figure 3: Runtime of `poisson`, for karman instances with various node counts

Inspecting Figure 3, it is clear that there is a logarithmic increase in performance as the node count increases. A strong tradeoff is presented when using between 4 and 10 nodes - node counts larger than that yield minor improvements on the runtime.

Conclusion

This paper detailed the parallelisation of karman. MPI and MPI-OpenMP hybrid implementations were developed, and the runtimes of functions were scrutinised under different settings. It was found that parallel implementations generally offer a positive speedup, however the option to include OpenMP parallelisation should be determined by the problem size, as the directive overheads may result in a lower performance speedup.