Contemporary Computer Science - GPU, Many-core, and Cluster Computing

Tom Robson - hzwr87

Step 1

Before attempts are made to improve the performance of the karman code, we must examine its current characteristics. This has been done using the Intel VTune performance analysis tool. The types of analysis used for this project were General Exploration, Advanced Hotspots and HPC Performance Characterisation.

From running these analysis methods, it has been determined that the key hotspots are in the functions computeP and setPressureBoundaryConditions. These functions are linked, as computeP calls setPressureBoundaryConditions, giving us a clear idea that the main issues for this algorithm lie in its implementation of pressure. From a closer examination of the computeP function, we can see that the main bottleneck in this function comes from the line given below:

```
p[getCellIndex(ix,iy,iz)] += -omega * residual / 6.0 * getH() * getH();
```

This line is so computationally expensive because it performs complex mathematical functions at floating point precision, meaning that it has a high CPI rate. The General Exploration analysis states that the primary bottleneck in this line comes from the Back-End of the pipeline, with the majority of this being as a result of a memory bound, where the Back-End cannot accept new operations due to existing operations still running. This function also has a substantial core bound, meaning that some of the execution ports have become saturated. The results of the Advanced Hotspots analysis reveal that this line also consumes a large proportion of the CPU time. The HPC analysis supports the General Exploration in the fact that there is a substantial memory bound present in the whole computeP function, and this line in particular.

When briefly examining the secondary bottleneck, setPressureBoundaryConditions, the General Exploration analysis tells us that this function is limited by the Front-End of the pipeline. The majority of this is caused by Front-End bandwidth, meaning that not all the slots in the pipeline are filled, rather than Front-End latency, where none of the slots are filled.

To address these issues, the performance model that should be applied is Strong Scaling. To calculate the possible speedup due to parallelisation over p processors, was must apply the equation $S(p) = \frac{t(1)}{t(p)}$, where $t(1)$ is the time taken on one processor, and the $t(p)$ is the time taken over p processors.$t(p)$ can be calculated using Amdahl's law, $t(p) = f.t(1) + \frac{(1-f).t(1)}{p}$, where f is the fraction of the code that is not parallelisable. As computeP is the main hotspot for this code, this will be the target of the parallelisation. Therefore, the non parallelisable part of the code is everything but computeP. These to formulae can be combined and simplified to give $S(p) = \frac{1}{f+\frac{(1-f)}{p}}$. To calculate the maximum possible speedup, we will assume p tends to infinity. The code was run on Hamiton with parameters 10 0 1600, and computeP consumes 66.6% of the runtime of the code, giving us a theoretical value for possible speedup of 2.99. With $t < 20$ as the while loop condition, the code runs in 1588.834 seconds.

Step 2

To break up the computational domain into blocks, or subproblems, we must first determine a suitable block size, set by a global variable. To facilitate the process of computation, it has been decided to restrict the grid dimensions to be multiples of this block size, supplied by the first command line argument. If this is not supplied correctly, the system will error. To allocate cells to blocks, we loop through the cells that would be in the top left corner of each block, and for all cells in the block, check if the obstacle is present in any of these cells. These blocks are enumerated by a counter, and if a block contains the obstacle, this is stored in a separate array of booleans, with the index in the array indicating which block the boolean refers to. This initialisation process is performed in setupScenario.

The benefits of this process can be realised in computeP, the main hotspot of this code, to improve its runtime. We loop through each block, and if the boolean array tells us that this block doesn't contain the obstacle, we can vectorise the loops for the cells in this block using Intel's SIMD pragmas. If one or more of the cells of the block does contain part of the obstacle, then the computation must be done in the same way as before, by looping through all of the cells individually, and checking if they have a part of the obstacle in them, and only performing the update to p if it is not. Due to the small size of the obstacle, the majority of the blocks can have SIMD vectorisation applied to them, so this vectorisation results in a substantial increase in the time taken to run the simulation. This process is necessary as the if statement to determine whether a cell contains the obstacle presents a barrier to vectorisation.

The vectorisation report, generated by the command -qopt-report-phase=vec,loop -qopt-report=5, tells us that the inner most loop of computeP that we told the compiler to vectorise with the SIMD pragma has indeed been vectorised successfully, and the vast majority of the rest of the code cannot be vectorised.

The effect of this has been measured below, running the code on hamilton over 100 timesteps with the grid size parameter varying between 8 and 32. As we can see, step 2 represents a significant speedup over step 1. The execution of step 2 is approximately 2.4 times faster than step 1. Changing the grid size also clearly has a large impact on the runtime. Changing the Reynolds number has an effect as well, but this only affects number of timesteps, and this has been fixed to 100 for the purposes of this experiment.

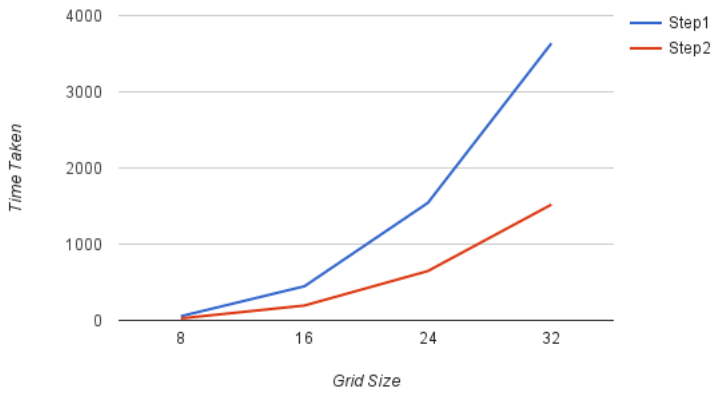

Figure 1: Comparison of Steps 1 and 2

Step 3

To eliminate the data races and facilitate parallelisation, we must implement halo layers for the blocks. In order to achieve this, we calculate the increase in number of cells in each dimension, by adding 2 cells per block in each dimension, and increase the size of p to accommodate this. In computeP, we loop through the blocks in a similar fashion to step 2, and loop through the cells containing data, ignoring the halos. The the end of each computeP iteration, the halos are updated using the updateHalo function so that they contain the correct neighbour data to be used for the calculations in the next iteration.

As halos are not necessary in any of the other arrays, and to facilitate accessing p from any of the functions other than computeP, we must implement some additional getCellIndex functions. When we access p from any other function, the loops have not been expanded to include halos, so we must map the value given by the loops to the newly expanded array. This is done by the function getCellIndexFromOriginals. We also need the inverse of this to access the other arrays from the loops intended for halos, done by the function getCellIndexFromHalos. The final function is for accessing the correct value in p with the halos embedded, used in the computeP loops, done using the getCellIndexHalo function.

When these functions have been implemented, we can then move on to the parallelisation. The OpenMP pragmas will be used to accomplish this, #pragma omp parallel for to be specific. As the global residual value is appended to in each section, and these sections must be brought together. This is done using the reduction option from OpenMP, following the pragma with the command 'reduction (+:globalResidual)'. The SIMD pragma from step 2 has also been changed to the OpenMP equivalent, and also uses the reduction for the global residuals. When comparing the two pragmas however, we found that the intel pragma with the reduction added produced superior performance.

When this new implementation is run, it appears to not be as fast as step 2. Tests have been run in the same way as in step 2, by varying the grid size over 100 timesteps. This is due to the overhead of parallelisation, leading to a large amount of spin time on the set of grid sizes that have been experimented with. We predict that this overhead will be reduced on larger grid sizes, but this has not been shown experimentally yet, and grid sizes up to 200 have been experimented with, and step 3 is still substantially slower than step 3.

Experimentation has also been done with the scheduling options for the parallelisation. This was done by including the statement 'schedule(runtime)' in the for loop pragma, and changing the OMP_SCHEDULE environment variable between static, dynamic and guided. The best runtime was achieved using the guided scheduling method.

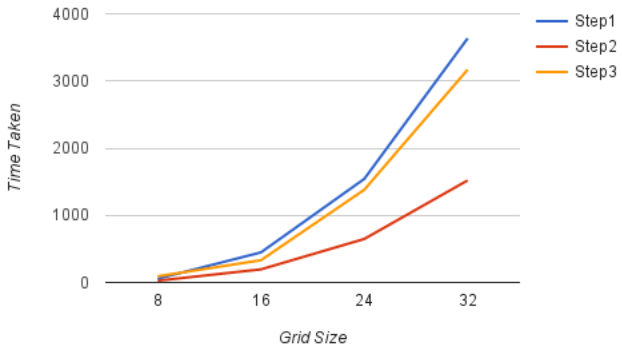

Figure 2: Analysis of step 3