1 Introduction

Implementing computational fluid dynamics in a program requires many nested loops. The OpenMP programming interface can easily be used to parallelize these loops via threads and therefore improve performance. The provided *Parallelizing Fluid Simulations* implementation includes loops that require different methods of parallelization. Some loops compute reductions and require different OpenMP declarations, while others have data dependencies that require special attention.

The goal for this project is get a parallel efficiency of at least 50% when running with 16 threads by using the following optimization techniques.

2 Parallelization Techniques

2.1 Collapsing

In total, the program includes seven functions that have computationally intensive loops that can be parallelized. The main technique used to speed them up was loop collapsing, which divides more work over the threads. Some loops only contained one nested loop, but others had two. Most required some minor alterations to change them into perfectly nested loops, which is required by the `collapse` declaration.

2.2 Scheduling

The manner in which threads are scheduled can have a noticeable performance difference. For this project, static scheduling on all the loop is likely to produce the best performance.

Due to the first-touch policy, when data is first accessed by a thread, that data will be allocated where that thread is running. If multiple loops have the same iteration count and are partitioned with the same number of threads, declaring them static scheduled will place the same block of indexes on the same threads for each loop. This prevents threads from having to read and write data from distant memory.

Even if a loop was able to be triple collapsed, all the loops were only collapsed twice to make sure the iteration count was the same across loops.

2.3 Reductions

Loops inside the `computeStableTimestep` and `integrateKineticEnergy` perform reduction operations, so they were quickly parallelized with the `reduction` as min

and max declarations respectively.

2.4 Data Dependencies

Some loops required special care because of data dependencies, specifically those in the `computeResidual` function. The method chosen to handle this was to mark these operations as atomic using the `omp atomic` declaration. This is the easiest method and will certainly perform much better than declaring the entire block `omp critical`, which does not make use of hardware specific instructions.

3 Results

All the measured times presented were taken on the Ptolemy cluster. Tests were conducted with 2, 4, 8, 16, 32, 64 threads via the provided run scripts, and the reported times are the average of three runs to reduce outliers.

The serial timings were done on the original, unaltered implementation via the `run01.js` run script.

Threads	Time (s)	Speedup	Efficiency
serial	129.996		
2	151.604	0.857	
4	85.846	1.514	0.379
8	50.354	2.582	0.322
16	34.473	3.771	0.471
32	19.855	6.547	0.205
64	17.414	7.465	0.117

Figure 1: Measured program run times

From the list of run times, it is obvious that, as expected, there are diminishing returns the after 16 threads, which is also the most efficient thread count, as expected. It also appears that running with less than four threads is worse than running the serial version of the program.

3.1 Efficiency Concerns

The target efficiency of 50% with 16 threads was not hit even with the above optimizations, though 47.1% was close. The slowdown is most likely in the `computeResidual`

function, which is one of the largest functions and has the most complicated loops.

The choice to use `omp atomic` declarations provides an easy way to prevent race conditions but is much less efficient than manually tiling the loops in such a way that there are no data dependencies. However, finding a clever way to do this without decreasing memory access efficiency and creating annoying bugs was very much a challenge which is why it was not chosen.

3.2 Scheduling Comparison

The method described in section 2.2 regarding scheduling was also tested to see how much scheduling effect performance. All the schedules were changed to dynamic to test the effects. Only the 16 thread timings were used.

The result was a 111.976% difference in run times with the static time being 34.473s and the dynamic time being 122.179s, which proves that scheduling definitely effects performance.