# Parallelized Fluid Dynamics Simulation

Alexander Kellough

**Overview:**

The Navier-Stokes equations are a set of equations for fluid dynamics that can be likened to Newton's laws of Motion for discrete objects. The simulation in question was originally written in serial to solve these equations for a set of fluid parameters in a 3-dimensional mesh. Students were asked to parallelize this program using OpenMP as a basis for multithreading.

**Methods and Techniques:**

The main method of parallelization was the OpenMP for work-sharing construct., takes a standard for loop and divides the iterations between threads, The implementation details were as follows:

**copyPeriodic:**

This function uses a surrounding parallel section, with inner for constructs separating the three inner for loops. No alterations are made to the for constructs, except for the final one, to which a no-wait clause is added. This is because the parallel section ends directly following this function, and forces synchronization, making the barrier at the end of this construct redundant.

**zeroResidual:**

This function was parallelized using an omp_parallel_for construct with no alterations.

**computeResidual:**

This function follows the same pattern as the copyPeriodic function.

**computeStableTimestep:**

This function uses the parallel for construct with a min reduction on the variable minDt.

**integrateKineticEnergy:**

For this function, a parallel for construct was used with a collapse(2) clause and an addition reduction on the sum variable.

**WeightedSum3:**

This function uses a parallel for construct with a collapse(2) clause.

**Notes:**

- Only the weighted sum and integration steps were found to increase in execution speed by collapsing the neighboring loops. This is because in other functions, there is no guarantee that the loops will be square(of equal size) making this take longer to resolve.
- The reason that parallel sections were used for the functions with multiple for loops is to remove the redundancy of constantly joining and forking the same number of threads.

**Analysis**

To approximate ideal speedup, the main bottleneck is assumed to be the work done per processor. This is due to the nature of parallelizing for loops (splitting iterations). Using the BSP model's definition of a superstep, it can be shown that there are 35 supersteps per timestep, equal to the number of for loops in the subroutines that occur within the main while loop (this isn't perfectly accurate to the actual implementation, as some barriers are left out for the sake of time optimization).. Taking all this into account, and putting it into Amdahl's Law we get:

$$T_p = c_o + f + 35i(\frac{w}{p})$$

The Tp is time parallel, Co is startup time/overhead, f is sequential fraction, and the last term is the product of supersteps(35), while loop iterations(i), and work per processor or for loop iterations(w/p). Assuming that each processor has to have at least one for loop iteration to affect the speed of the program, this gives the ideal running time of:

$T_p = c_o + f + 35i$ (W is used instead of p for clarity. They must be equal for the above fractional section to reduce).

This means that the ideal speedup reduces the overall time to a linear relationship between simulated time, and a constant (offset and serial fraction). Assuming time is static, this should reduce to some constant (Offset + serial fraction + 35 * time * thread_overhead).

**Data:**

*Tests were done on a dataset of 1000 puzzles, and timed using a timer around the server process, which outputs the total number of solutions.*

| # of Processors | Average Execution Time (in secs) |
|---|---|
| 1 | 255.069 |
| 2 | 138.951 |
| 5 | 60.2136 |
| 10 | 35.6481 |
| 20 | 23.332 |

)Note: Results are rounded to the nearest half to aid in visualizing trends).

Speedup(10): $S = \frac{255.069}{35.6481} \approx 7.16$

Speedup(20): $S = \frac{255.069}{23.332} \approx 10.93$

Efficiency(10): $E = \frac{7.16}{10} \approx 0.716$

Efficiency(20): $E = \frac{10.93}{20} \approx .547$

**Final Thoughts**

As demonstrated by the above analysis, the total time follows the previous prediction closely. Approaching a constant overhead. One interesting experiment would be to vary the time and complexity of the simulation to see how that scales compared to earlier numbers. One thing that affects this constant is the thread overhead, which was only briefly mentioned. Another factor is small sections of serial code and/or blocking sections within the parallel region of code.