CSE 6163 - Project 2: Parallelizing Fluid Simulations

Jacob Kutch

April 7, 2023

Introduction

This project involved the parallelization of a serial computational fluid dynamics solver used to solve the Navier-Stokes equations using numerical methods for turbulent flows. In particular, it solves a benchmark problem that simulates the breakup and dissipation of a large eddy structure. The benchmark, the Taylor-Green vortex problem, is commonly used to evaluate the suitability of fluid solvers in the simulation of the dynamic details of turbulent flows.

The code involves computation of total kinetic energy of fluid in all cells in the mesh by Runge-Kutta numerical integration at various time steps. These time steps must be minimized over each cell considering inviscid and viscous terms of the problem to find a stable time step for each iteration of the simulation. This is the starting point of the time integration. Due to the nature of considering the mesh in a grid form, while actually a continuous surface, careful attention is given to enforce periodic boundary conditions on the mesh to correct for "ghost grid values." Error is approximated by the sum of momentum fluxes of each cell's faces. These residuals are intended to be minimized by minimizing total flux through cells and their immediate neighbors. A weighted sum procedure is used to update solutions by using previous solutions and the computed residuals. This iteration repeats until the entire time interval has been traversed.

The project involves parallelization on two fronts: thread-level parallelism using OpenMP and process-level parallelism using OpenMPI. The thread level parallelization was mostly a matter of distributing loop iterations between threads, while the MPI implementation was very complicated. Ultimately, this project fell short in implementing the process-level parallelism in the short amount of time provided.

Methods

The original incarnation of the C++ project was provided by Dr. Edward Luke of Mississippi State University. All utilities were provided save for a parallelized driver file. The original driver file, ***fluid.cc***, was implemented sequentially but written so as to be easily parallelized with OpenMP. The code was run on the Mississippi State High Performance Computing Center's Shadow cluster of the Titan supercomputer, using openMP and slurm batch scheduling. All results are estimated from running on the cluster.

The first function encountered in the solver and the first to be parallelized was `setInitialConditions()`, which initializes each variable for each cell using provided parameters of the canonical Taylor-Green vortex problem. This contains a doubly-nested loop with no dependencies in loop conditions that needed to be amended. Simply declaring the for-loop block to be parallel with the OpenMP compiler directive "`#pragma omp parallel for`" was enough to improve speeds. This was the case for other functions that simply iterated over each cell as well. `zeroResidual()`, which resets residual vectors back to 0 for the next integration step, also contained a doubly-nested loop that only needed the same compiler directive as `setInitialConditions()`. This was also the case for `weightedSum3()`, which computes the weighted sum over each variable to be used in the next step of Runge-Kutta time integration. This same directive was used in `copyPeriodic()` for each periodic face of cells in the system, i.e., the directive was used on three separate single nested loops that copied periodic boundary conditions at the boundary of the faces given by the yz-trace, xy-trace, and xz-trace. Additionally, a convenience function `copy_faces()` was written to improve readability within `copyPeriodic()`.

Similar attempts were made to improve readability of `computeResidual()`, which compute the residue of the computed rate of change for the pressure variable p and the three components of the velocity vector denoted (u, v, w). There were multiple attempts to create these convenience functions, but all resulted in either breaking parallelism or enormous memory requirements in making deep copies. Using reduction clauses were also out of the question because of the specific implementation of the residual vectors. Using the directive "`#pragma omp parallel for`" was sufficient on each doubly-nested loop except for one test case, most likely due to race conditions. Trying to avoid race conditions using atomic clauses and critical sections both showed a greatly diminished performance from using a simple `for` clause alone. It was likely because the 8 updates on 4 shared variables created a large amount of overhead where many threads were waiting to access that shared memory. Additionally, using 40 threads required both changing the file ***run40.js*** to use 20 threads per processor to align with the available hardware of the shadow cluster while still using `OMP_NUM_THREADS=40` to actually use 40 threads via hyperthreading and also required the addition of a new partition scheme within the nested loops. Without the first change, the batch job did not run at all. Without the second change, results were typically incorrect. To fix this, another outer loop that went through 2 iterations was added to each set of doubly nested loops so that the code iterated over only odd iterations or only even iterations of the outermost loop at any given time, thus avoiding race conditions.

Simple reduction clauses were enough to improve the execution time of both `computeStableTimestep()` and `integrateKineticEnergy()`. The former required minimization of time steps within another doubly-nested loop, so a reduction clause specifying *min* as the operation was used with the directive

```
#pragma omp parallel for shared(u,v,w) reduction(min: minDt)
```

where `minDt` accumulated the minimum over all threads. The latter function used a reduction clause centered around the accumulator variable `sum`, which summed the fluid kinetic energy over the whole domain. In this case, the reduction clause was `reduction(+: sum)`.

The additional requirement given to graduate students was much more difficult addition and required hybrid parallelism using multiprocessing through MPI as well as the above implementation of OpenMP. Many attempts were made to distribute data among different processes and allow for the multithreaded loops to run over slices of the original arrays.

Analysis

The following analysis will only focus on the multithreaded performance analysis since increased speedup was not attained through multiprocessing. It is also worth noting that the sequential runtime increased a fair amount due to the addition of helper functions and other small additions. Given the mostly independent and uniform nature of tasks within each nested loop, we may consider the total amount of sequential work W to be done as being distributed among the number of threads. The ideal speedup S_{ideal} can be found using the ratio of sequential run time T_1 on one thread (assumed in this case to run at a fixed rate of k operations per second and require amount of work W) to the parallel run time T_p on p threads given by

$$S_{ideal} = \frac{T_1}{T_p} = \frac{W/k}{W/(kp)} = p$$

First consider the case in which the simulator performs some total amount of serial work. The sum total of operations involved in instantiating simulation parameters, allocating vectors, recording results, saving output, and any other operations that would perform work on a single thread will be denoted W_s. Task k will require W_k operations to complete. The time required to solve the problem on a single thread with no communication overhead is then

$$T_1 = W_s + \sum_{k \in Tasks} W_k$$

The time required for the completion of each task depends only on the specific function that the task belongs to. Any smaller task within the simulator is likely to be found deep within a nested for-loop. The actual number of these tasks is on the order of $O(n^3)$ in most cases with a doubly-nested loop and $O(n^2)$ in the single-nested loop, where n is the largest number of cells in any direction. Because the number of atomized tasks is much larger than the number of threads for any given p, time to distribute and collect results between threads is not likely to be a negligible amount. For the sake of developing the formula, consider it negligible to start with. Thus distributing the tasks (assumed to be completed in constant time) between threads is given by the sum of sequential work plus some amount of work done in parallel by

$$T_p = W_s + \frac{1}{p} \sum_{k \in Tasks} W_k$$

Since $lim_{p \to \infty} = 0$, we see that no matter the theoretical speedup by the number of additional threads, we are limited by the work required in serial W_s, and in reality also by the overhead of the time required to communicate between threads that we were ignoring for the sake of this analysis. The sequential fraction may be written as the ratio of sequential work required to total work required as

$$F_s = \frac{W_s}{W_s + \sum_{k \in Tasks} W_k}$$

Then parallel execution time T_p may be written using Amdahl's Law as

$$T_p = \frac{T_1}{p}(F_s(p-1)+1)$$

When considering the task distribution to be non-negligible as previously assumed, let C_s be the time required for OpenMP to initially deal with identifying the number of threads and other operations needed when preparing to add multithreading. The work needed to partition a loop among threads and to combine results upon completion of the tasks within a fork-join model result in some overhead per task C_k. We must consider the communication time C_s in the serial task time that cannot be reduced as a sum with W_s.

$$T_p = (W_s + C_s) + \frac{1}{p} \sum_{k \in Tasks} (C_k + W_k)$$

As the number of threads increase, so too do the copies of private variables that need to be made, the care with which shared variables are used, and the order in which threads are joined. As the number of threads increase, the distribution of work between threads and resultant cache conflicts, etc could potentially become a bottleneck, and the speedup may plateau or decrease, as the cost C_k grows with the number of threads p.

Amdahl's Law with communication overhead gives the T_p previously written, but T_1 would not require this communication time overhead and simply stay the same. We can finally approximate the speedup to be

$$S_p = \frac{T_1}{T_p} = F_s(1-p) + p = \frac{W_s + C_s}{W_s + \sum_{k \in Tasks} W_k}(1-p) + p$$

Taken alone, the simpler form of Amdahl's Law shows that the speedup was limited by the sequential fraction, but it is now limited by the the communication overhead and this sequential fraction.

We also know the efficiency of this algorithm to be

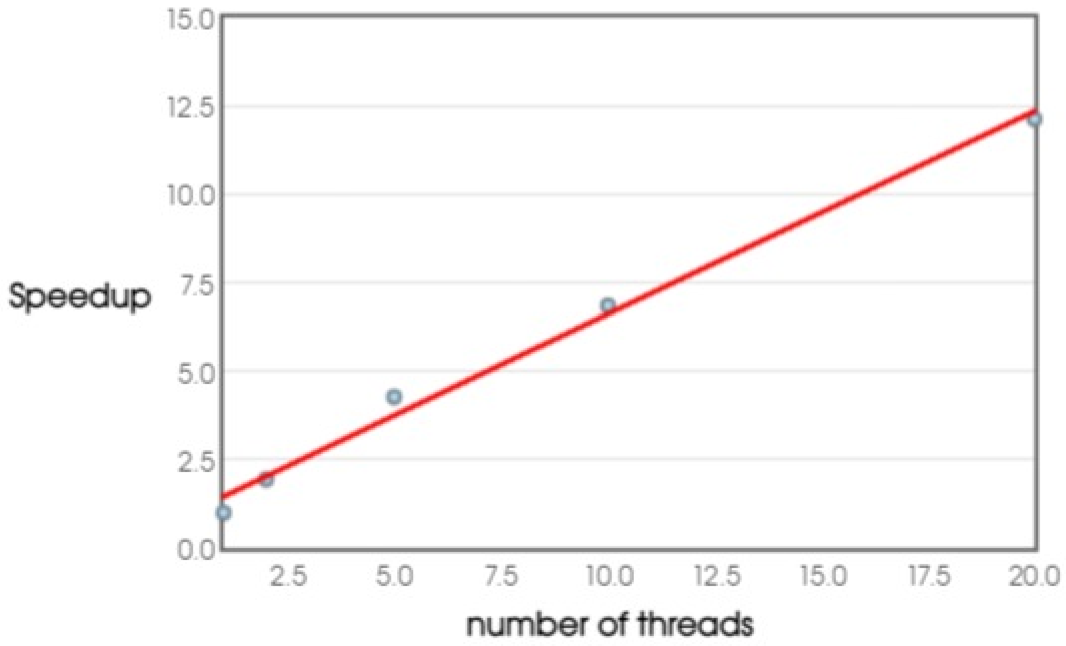
$$E_p = \frac{S_p}{p} = \frac{W_s + C_s}{W_s + \sum_{k \in Tasks} W_k}(\frac{1}{p} - 1) + 1$$

Results

The code went through many iterations of implementations, but the one that satisfied all multithreading requirements (where accuracy is concerned) achieved the results below.

Threads (p)	1	2	5	10	20	40
Iterations	3482	3482	3482	3482	3482	3482
Execution Time (s)	310.788	159.486	72.655	45.352	25.622	52.533
Speedup S_p	1	1.949	4.278	6.853	12.130	5.916
Efficiency E_p	1	0.9745	0.8556	0.6853	0.6065	0.1479

Note that the speedup using just 2 threads is nearly the ideal speedup of $p = 2$ and where the efficiency is nearly 100%. The relative improvement as the number of threads increased decreases very quickly, however. If more care is taken in distributing work among threads, I believe that this would be mitigated. Also note that the improvement with 40 threads is much worse than one would expect and displays an even worse speedup than seen when using just a quarter as many threads. I believe that this is mainly due to reasons related to the parallel cluster that it was run on, as we were later informed that the cluster only accounted for up to 20 physical cores. Given that relatively simple usage of OpenMP was employed, I believe this time could definitely be improved on by a more adept treatment of the parallelized sections discussed. Below is a linear regression plot of the speedup witnessed, with the exception of the outlier of 40 threads.



Conclusion

Overall, the undergraduate portion of the project was a success and all features and implementation details were realized. While there is room for improvement in the specific parallel implementation of OpenMP and definitely much improvement to be made using multiprocessing, the code performs admirably and reaches an appreciable speedup. Future work is needed in meaningfully optimizing the parallelism of `computeResidual()` and in atomizing the code enough that so that more complex task parallelism may be employed. For example, a divide and conquer fork-join thread model could even be used.

Additionally, if MPI were effectively introduced to include data parallelism by distributing data between processors in addition to the task parallelism given by OpenMP, I imagine an appreciable speedup could be observed. Time limitations prevented this from coming to fruition, but a clear goal is apparent for future work.

References

[1] Dlepow, “MPI functions - message passing interface,” Message Passing Interface — Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/message-passing-interface/mpi-functions. [Accessed: 07-Mar-2023].

[2] W. Kendall, D. Nath, and W. Bland, “Tutorials,” Tutorials · MPI Tutorial. [Online]. Available: https://mpitutorial.com/tutorials/. [Accessed: 07-Mar-2023].

[3] “Web pages for MPI and MPE,” Web pages for MPI. [Online]. Available: https://www.mpich.org/static/docs/v3.2/. [Accessed: 07-Mar-2023].

[4] L. Tychnoviech and C. Reiss, “1 task,” CS3130: OpenMP – parallelism made easy. [Online]. Available: https://www.cs.virginia.edu/ cr4bd/3130/S2023/labhw/openmp.html. [Accessed: 3-Apr-2023].

[5] J. Yliluoma, “Guide into OpenMP: Easy multithreading programming for C++,” Guide into openmp: Easy multithreading programming for C++, Jun-2016. [Online]. Available: https://bisqwit.iki.fi/story/howto/openmp/. [Accessed: 05-Apr-2023].

[6] “OpenMP Application Programming Interface,” OpenMP. [Online]. Available: https://www.openmp.org/spec-html/5.1/openmp.html. [Accessed: 10-Apr-2023].

[7] J. Hückelheim and J. Doerfert, ”Spray: Sparse Reductions of Arrays in OPENMP,” 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, 2021, pp. 475-484, doi: 10.1109/IPDPS49936.2021.00056.

[8] “OpenMP: Beyond the basics,” YouTube, 02-Nov-2016. [Online]. Available: https://www.youtube.com/watch?v=czfpVTVkiyc. [Accessed: 05-Apr-2023].

[9] VICTOR EIJKHOUT, Parallel Programming in MPI and OPENMP, vol. 2.