Parallel Fluid Dynamics Simulation: Technical Report

Sydney A. Christenson

sac771

Mississippi State University

CSE 4163: Design of Parallel Algorithms

Dr. Ed Luke

04/14/2024

Methods and Techniques

Initial Implementation

For the initial implementation of this project, each function was parallelized before testing the results. Once the final function was parallelized, great speedup was achieved. However, the outputs differed between running on 32 thread and 64 threads, indicating a race condition. By narrowing down the code once again, this time by each loop, it was found the race condition appeared within the *computeResidual* function. This is due to a dependency that arises when adding to the *presid, uresid, and wresid* arrays. Due to this, a different approach needed to be implemented.

Final Implementation

In the final implementation of this project, the strategy remained largely the same, focusing on parallelizing individual loops within each function. The key improvement was in reordering the loops within the computeResidual function, which effectively resolved the race condition that arose from concurrent modifications to the residual arrays by multiple threads found in the initial implementation. This targeted restructuring of loops was critical in managing data dependencies and eliminating bottlenecks, leading to a more reliable and efficient simulation process.

OpenMP Techniques Used

Several OpenMP techniques were utilized to optimize performance and ensure the correctness of the parallel outputs:

1. #pragma omp parallel for: This directive was used to distribute loop iterations among available threads, which leads to greater optimization. It was applied to loops across all functions where data independence between iterations could be guaranteed.
2. Collapse directive: To maximize parallel efficiency, nested loops were collapsed into a single loop iteration space, allowing for more uniform distribution of iterations across threads. This technique was particularly useful in the computeResidual function, where three nested loops were combined to ensure that each thread could perform a substantial amount of work.
3. Reduction directive: Used in functions like computeStableTimestep and integrateKineticEnergy, the reduction operation was crucial for correctly performing operations like finding the minimum timestep and summing up kinetic energy across all iterations, respectively.
4. Loop restructuring: To mitigate the race conditions observed in the initial implementation, loops within the computeResidual function were restructured. Instead of processing based on the original data layout, loops were organized to process data in blocks that reduced dependency conflicts, ensuring that each thread worked on data segments that did not overlap with those of others in critical sections.

Theoretical Analysis

Amdahl's Law

By utilizing Amdahl's Law, the expected speedup of this project can be calculated based on the number of threads. Within the project, the jobs could be run on 2, 4, 8, 16, 32, or 64 threads. Considering the typical overheads of threading such as managing thread life cycles and the non-uniform memory access, P (the proportion of the code that is able to be parallelized) is pragmatically set to 91%. The following will calculate the theoretical speedup:

$$\textit{2 processors} = S = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-0.91)+\frac{0.91}{02}} = \textit{1.83x speedup}$$

$$\textit{4 processors} = S = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-0.91)+\frac{0.91}{04}} = \textit{3.15x speedup}$$

$$\textit{8 processors} = S = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-0.91)+\frac{0.91}{08}} = \textit{4.91x speedup}$$

$$\textit{16 processors} = S = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-0.91)+\frac{0.91}{16}} = \textit{6.81x speedup}$$

$$\textit{32 processors} = S = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-0.91)+\frac{0.91}{32}} = \textit{8.44x speedup}$$

$$\textit{64 processors} = S = \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-0.91)+\frac{0.91}{64}} = \textit{9.60x speedup}$$

S being the theoretical speedup

P being the parallelized proportion

N being the number of processors

According to this, there should be a greater speedup as the number of processes increases.

Granularity

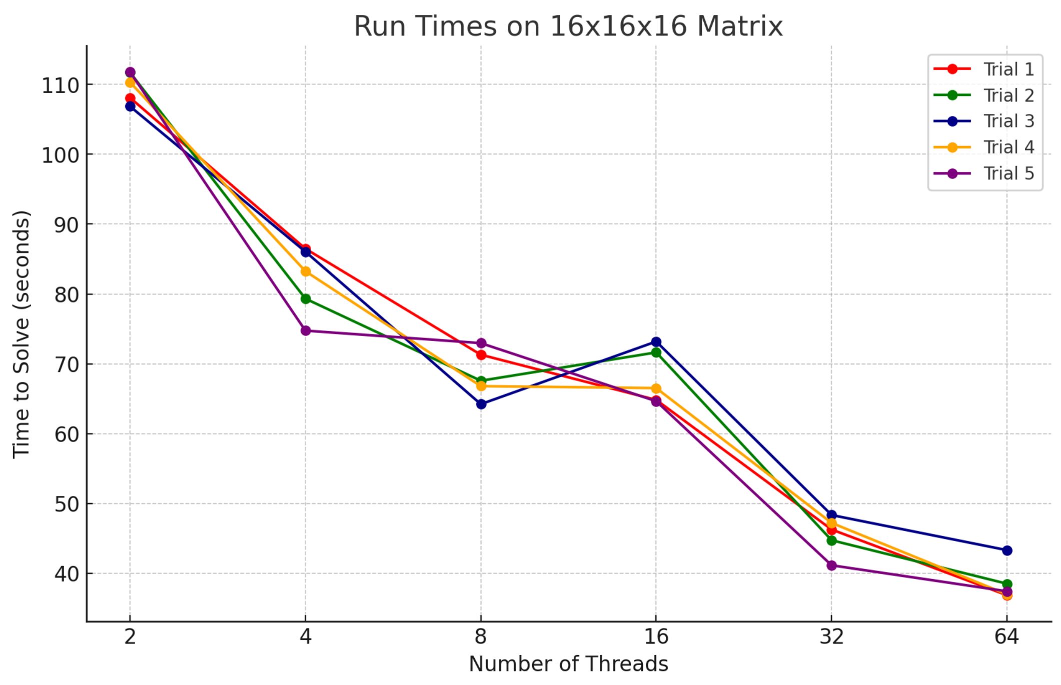
The matrix size significantly influences the effectiveness of parallelization. As matrix size increases, each thread can work on a larger chunk of the matrix, effectively increasing the workload per thread and reducing the overhead of thread management and synchronization. This enhanced granularity can lead to better utilization of the system's computing resources, such as cache and memory bandwidth, by each thread. This allows the program to better leverage the available hardware, potentially approaching the theoretical maximum speedup predicted by Amdahl's Law.

Race Conditions

Race conditions in parallel computing can significantly impact the effectiveness of parallelization, leading to incorrect results or unexpected behavior. These conditions occur when multiple threads attempt to read and write shared data simultaneously without proper synchronization, causing inconsistencies and errors. For instance, in an OpenMP project, if multiple threads modify the same memory location without coordination, it can lead to data being overwritten unexpectedly, resulting in incorrect calculations and outcomes.

Results

In analyzing my approach to this project, the performance data greatly aligns with the theoretical analysis of Amdahl's law. From the graph, it is observed that as we move from 2 to 64 threads, the improvement in run times starts to plateau, especially beyond 32 threads, which aligns with Amdahl's Law indicating diminishing speedup as we add more processors.



The speedup from this data set can be calculated by taking the average serial time, found to be 385.207 seconds, and dividing it by the average time of each thread:

2 Threads: Average Time: 110.54 seconds = 385.207/110.54 = 3.48

4 Threads: Average Time: 82.89 second = 385.207/82.29 = 4.65

8 Threads: Average Time: 70.69 seconds = 385.207/70.69 = 5.45

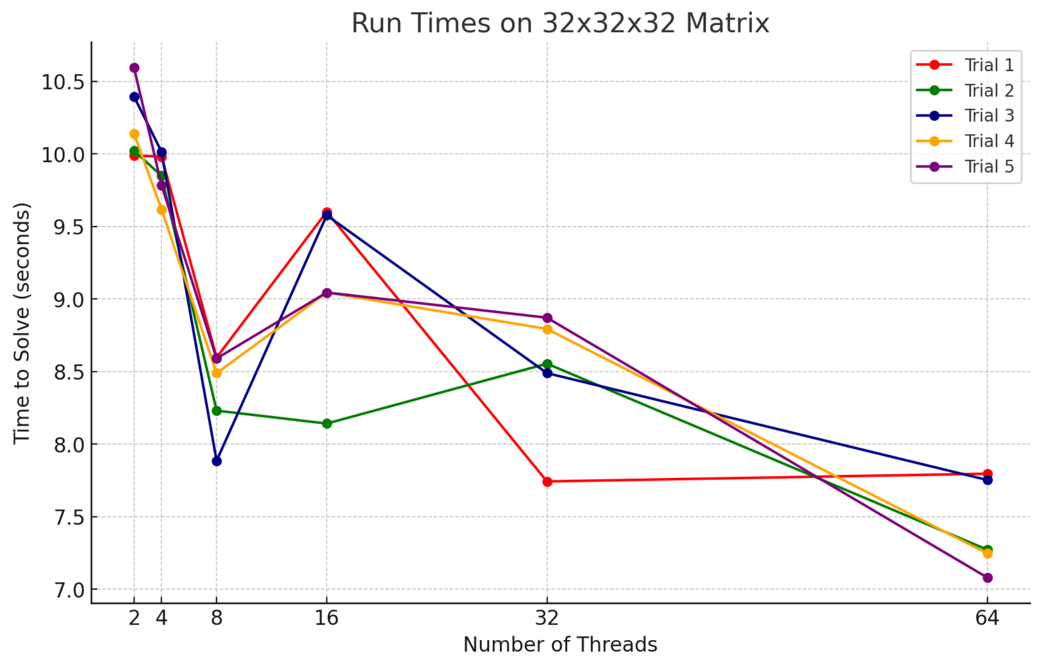
16 Threads: Average Time: 65.60 seconds = 385.207/65.60 = 5.87

32 Threads: Average Time: 44.63 seconds = 385.207/44.63 = 8.63

64 Threads: Average Time: 37.63 seconds = 385.207/27.63 = 10.24

The calculated speedups are greater than those predicted by Amdahl's Law with a P value of 0.91, suggesting that either the system has an effectively higher P due to additional optimizations or that the system is capable of superlinear speedups that have not been accounted for.

To create a more fine grained environment for this project, I upscaled the dimensions of the matrix to 32x32x32. The larger matrix size suggests a finer granularity, where a greater proportion of the computation can be effectively divided across multiple threads. As the problem size increases, the overhead of managing parallel processes is offset by the more substantial amount of work each thread can perform, leading to improved efficiency and performance. This efficiency gain is especially evident in the steep decline in run times with an increasing number of threads.



With this data, speed up is found to be:

2 Threads: Average Time: 10.23 seconds = 385.207/10.23 = 37.66

4 Threads: Average Time: 9.85 seconds = 385.207/9.85 = 39.11

8 Threads: Average Time: 8.36 seconds = 385.207/8.36 = 46.09

16 Threads: Average Time: 9.08 seconds = 385.207/9.08 = 42.42

32 Threads: Average Time: 8.49 seconds = 385.207/8.49 = 45.37

64 Threads: Average Time: 7.43 seconds = 385.207/7.43 = 51.85

Synthesis

In this project, I kept the parallelization at the loop level and reconstructed the loop structure within the computeResidual function to avoid race conditions. This loop reordering helped manage the data correctly, maintaining good speed while ensuring accurate outputs. I utilized several OpenMP techniques such as: 'parallel for', 'collapse', and 'reduction' to make sure loops ran efficiently and safely across multiple threads.

Theoretical predictions using Amdahl's Law expected a certain speedup based on a 91% parallelizable code, however, the calculated speedups were even better, suggesting my code might be more parallel-friendly or that there are extra speedups happening that Amdahl's Law doesn't cover. The finer granularity by upscaling the matrix size proved better results, allowing threads to do more work with less overhead, showing that a well-structured parallel program could indeed outperform the theoretical limits.