15418 Final Project: Comparing OpenMP and MPI Parallelization Strategies for 3D Particle Fluid Simulation

Katerina Nikiforova

April 2025

1 Website URL

https://kn1451j.github.io/ParallelFluidSim/

2 Summary

This project entailed developing a 3D particle-based fluid simulator from scratch, and optimizing it using OpenMP and MPI strategies, which were compared by efficiency and communication overhead. The more standard parallelization strategy of CUDA SIMD instructions is discussed in application to the semi-Eulerian component of this implementation of the Fluid-Implicit Particle method. The final implementation landed on using OpenMP for shared-memory parallelization, with near-linear improvements over the sequential implementation with respect to thread count. A more intensive MPI method was evaluated, though the preliminary results for this method were less promising.

3 Background

The FLIP [BKR88] method models fluid dynamics using a hybrid particle-grid approach. It iterates over particles, updating velocities and positions using classical and transfers their velocities to the grid in

- **Advection**: Particles are moved through the velocity field.
- **External Forces**: Gravity and other body forces are applied.
- **Particle-to-Grid Transfer**: Particle data (velocity, mass) is mapped to grid cells.
- **Pressure Solve**: Solves a sparse linear system to enforce fluid incompressibility.
- **Grid-to-Particle Transfer**: Updated grid velocities are transferred back to particles, blending PIC/FLIP methods.

4 Background

The Fluid-Implicit Particle (FLIP) method models fluid dynamics using a hybrid particle-grid approach. It iterates over particles, updating their velocities and positions according to the underlying

velocity field, external forces, and pressure corrections derived from the incompressibility condition of the Navier-Stokes equations [20].

At the particle level, the velocity $\mathbf{v}_p$ and position $\mathbf{x}_p$ of each particle are updated as:

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{v}_p \tag{1}$$

$$\frac{d\mathbf{v}_p}{dt} = \mathbf{g} + \mathbf{f}_{\text{pressure}} \tag{2}$$

where $\mathbf{g}$ is the external force per unit mass (e.g., gravity) and $\mathbf{f}_{\text{pressure}}$ represents pressure forces interpolated from the grid.

At the grid level, to maintain incompressibility, the velocity field $\mathbf{u}$ must satisfy:

$$\nabla \cdot \mathbf{u} = 0 \tag{3}$$

To enforce incompressibility after advection and external forces, we solve a pressure projection step that ensures the total velocity divergence of the update step is 0 and follows the Navier-Stokes equations of fluid motion under pressure. For this, we seek a corrected velocity $\mathbf{u}^{n+1}$ such that:

$$\mathbf{u}^{n+1} = \mathbf{u}^n - \frac{\Delta t}{\rho}\nabla p \tag{4}$$

This is solved using an iterative method called conjugate gradient descent for a sparse divergence operator matrix A, pressure vector p, and velocity divergence vector v.

$$A\mathbf{p} = \mathbf{b}$$

With this, are 4 stages to the pipeline:

- **Advection**: Particles are moved through the velocity field according to (1).
- **External Forces**: Body forces such as gravity $\mathbf{g}$ are applied via (2).
- **Particle-to-Grid Transfer**: Particle velocities and masses are mapped to grid nodes using interpolation (e.g., trilinear weights).
- **Pressure Solve**: A sparse linear system is solved to enforce the incompressibility condition (3).
- **Grid-to-Particle Transfer**: Grid velocities are interpolated back to particles

The initial implementation was based off of the guidelines and derivations in [Bri15]. There were significant challenges in numerical stability, and many bugs due to the size of the codebase. The most significant bugs in the sequential implementation involved mathematical miscalculations - specifically in the pressure solve which were found through extensive formula cross-references. Existing (non-parallelized) open source examples like [Dan22] were helpful in the outline of the initial implementation. The system was first implemented in 2D, and expanded into 3D after debugging.

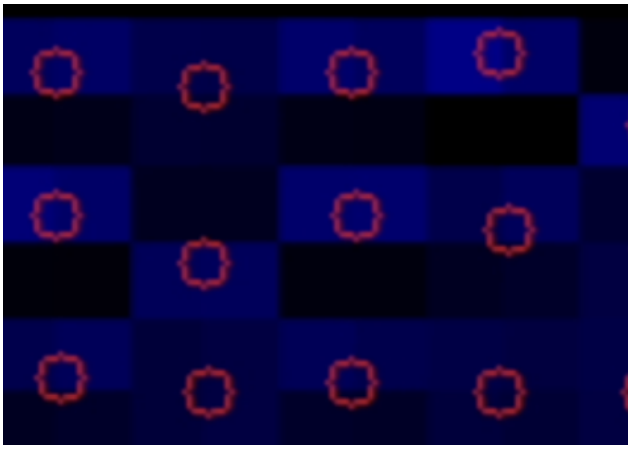

Figure 1: The system consists of particles distributed across a grid, carrying velocity information which is then used to approximate pressure forces using a second-order approximation over the grid.

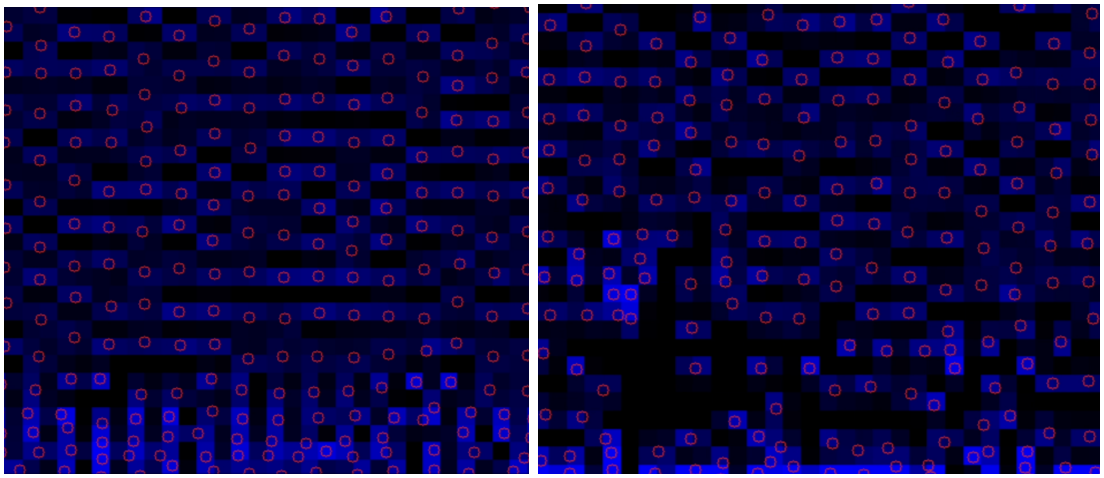

Figure 2: Two views of a single run of the 2D grid simulation visualized, with the intensity of the grid cell denoting its density and the red circles visualizing particle position.

After this expansion, the system was discretized into a $32 \times 32 \times 32$ grid with 2048 particles. The rest of the discussion will be based on this configuration.

The most computationally expensive part is the pressure solve, requiring a sparse linear system solution. The particle-grid transfer and advection are mostly data-parallel with high spatial locality.

5 Approach

5.1 Sequential Implementation

The initial sequential 3D implementation was instrumented with profiling across every step. It was clear that there was necessary speedup, as a single timestep was taking up to 0.12 seconds to evaluate, which made the simulation significantly slower than real-time.

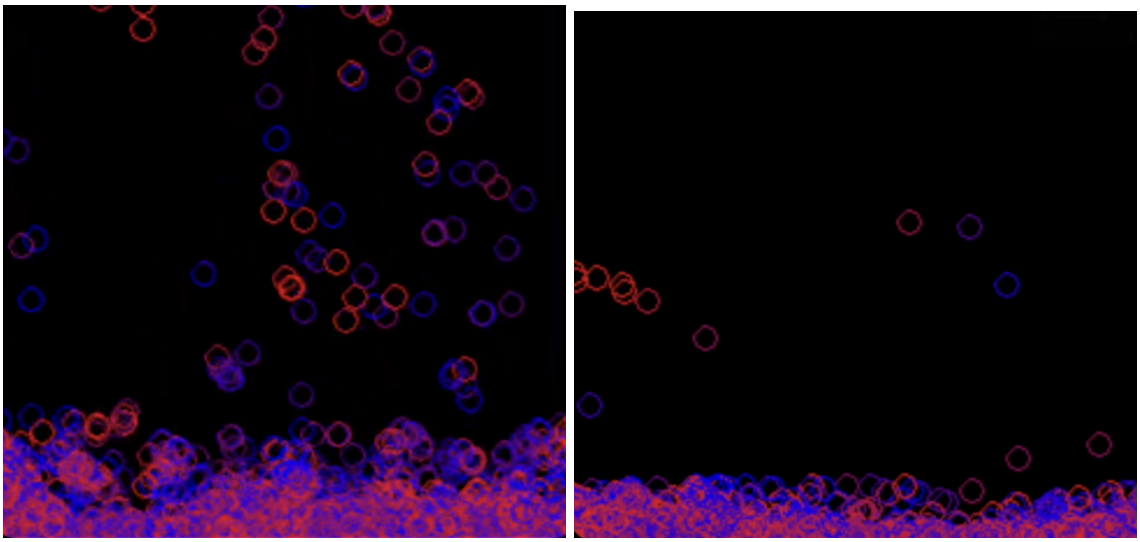

Figure 3: Two sliced views of a single run of the 3D grid simulation visualized using particle color to denote depth.

5.1.1 Initial Profiling

Initial profiling results (tables 1 and 2) revealed that the pressure solve step dominated the runtime, which can be seen in figure 4.

Function	**Time (sec)**
advect	0.0205
external_forces	0.0049
solve_pressure	33.0411
transfer_from_grid	0.4405
transfer_to_grid	0.6244

Table 1: Profiling results over 500 timesteps using preconditioned conjugate gradient descent.

Preconditioning is a strategy used to reduce the number of times for the pressure solve, an iterative descent method, to converge. By "pre-conditioning" the vector using Cholesky-decomposition, it is possible to improve the descent and converge quicker. This difference can be seen in tables 1 and 2.

5.2 Parallelization Strategy

The entirety of our parallelization strategy focused on the conjugate gradient descent algorithm, purely because of its runtime dominance (4).

The initial observation that made parallelization possible was noted in [Wu+18], which observes that the only computational dependencies in a simple conjugate gradient solve are in the preconditioner algorithm, as described in [Bri15].

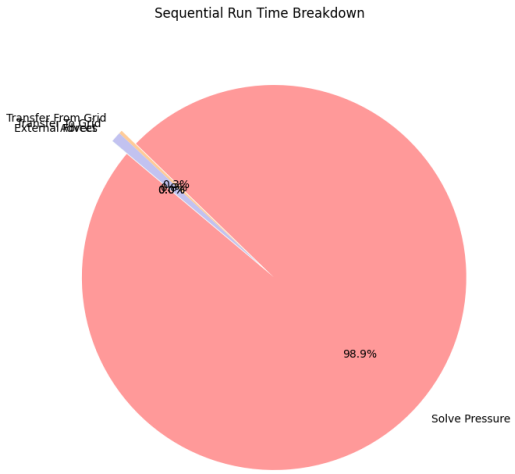

Figure 4: The time breakdown on a sequential pressure solve of the fluid simulator, demonstrating the computational domination of the pressure solver.

Function	**Time (sec)**
advect	0.0056
external_forces	0.0010
solve_pressure	44.9276
transfer_from_grid	0.1164
transfer_to_grid	0.2545

Table 2: Profiling results on 500 timesteps without preconditioning.

$$L_{ij} = \sqrt{A_{ii} - \sum_{k<i} L_{ik}^2} \text{if } i = j$$

To eliminate this dependence, we completely removed the preconditioner, sacrificing poorer convergence for increased parallelization. Thus, our strategy involved parallelizing the simple conjugate gradient descent, where within a single iteration, each index of the pressure update is calculated independently, except for a shared constant factor σ that modifies the step size of the update and is calculated through an inner product of the entire residual vector:

Conjugate Gradient Pseudocode:

```
Initialize: x = 0, r = b - A * x, p = r, k = 0
While not converged and k < max_iterations:
    alpha = (r^T * r) / (p^T * A * p)  # Step size
    x = x + alpha * p                  # Update solution
    r_new = r - alpha * A * p           # Update residual
    If norm(r_new) < epsilon: break     # Convergence check
```

```
    beta = (r_newˆT * r_new) / (rˆT * r)  # Direction adjustment
    p = r_new + beta * p                  # New search direction
    r = r_new                             # Update residual
    k = k + 1

Return: x
```

We noted that interactions with the sparse A matrix were READ only, requiring no synchronization across threads, and each cell's pressure solve with respect to other cells is independent within a single iteration, conditioned on this A. Thus, within an iteration, there is significant parallelism to be taken advantage of for large grids. We further point out there is significant locality structure to the parallelism, where adjacent cells access adjacent A values, and modify adjacent pressure cells. We take advantage of this by ensuring that the ordering of grid iteration is in contiguous order to maximize cache locality and throughout parallelization, processor memory coherence.

While the initial plan for the parallelization was to use CUDA to independently calculate each index of each pressure solve by launching a thread for every index of the pressure solve. However, there were multiple concerns with this method of parallelization for this context:

1. Each index of the pressure solve has wildly different computation cost due to its dependence on the density and "fluidity" of its neighbors. A fluid cell will only have pressure interactions with its neighbor in the solver if its neighbor is another fluid cell. Thus, each cell has very different amounts of computation.
2. The necessity to globally redistribute data between every thread after every iteration, and relaunch all threads every iteration.

However, we noted that between iterations, although there is communication required between every cell in the beta and alpha calculation 5.2, the amount of communication was low - requiring only a single floating-point reduction for these values. The larger communication cost was in the communication required from neighboring cells of each cell in the block for the calculation of Ap, since the value of p changed every iteration. HOWEVER, observing the problem leads to the insight that A is a sparse, close-to-diagonal matrix by nature. This means that every cell has 6 neighbors, but any neighboring cell of that cell will be a neighbor of at least 8 of the surrounding cells of that cell. More loosely, a cell will share many of its neighbors with nearby cells. This idea is visualized in 5.

Thus, for every $M \times N \times K$ block, there are $2KM + 2KN + 2MN$ elements required to be communicated 6. As the block size increases, this approaches an arithmetic intensity of O(N) for cubic blocks. This need for iterative communication created an interesting challenge for the CUDA implementation due to the block-block GPU-CPU communication every iteration. Although the CUDA implementation would likely be an efficient way to parallelize, this relationship made message-passing an interesting parallel topic to explore. Since every block just needed its perimeter of neighbors, the question of how message passing compares to the OMP shared-memory model becomes interesting.

5.3 MPI Exploration

We implemented MPI parallelization by

- Creating cubic blocks of work over the 3D grid and updating each independently each iteration

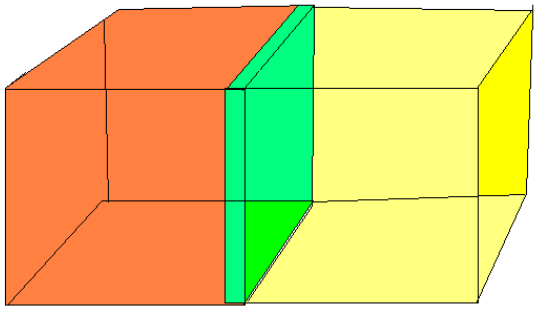

Figure 5: Every W^3 block (orange and yellow) of neighbors only requires W^2 neighboring (green) elements from each side of the block to compute Ap

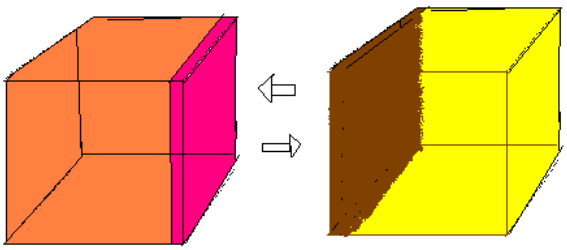

Figure 6: Left block (orange) and right block (yellow) can fully communicate the information needed by each others computation of Ap for the pressure indices in p relating to the block by sending to its neighbors the perimeter cells bordering that neighbor (e.g. orange block sends its pink cells to the yellow block).

based on pressure estimates from prior iterations

- Boundary communicating with neighboring blocks each iteration

Our profiling runs saw poor performance, experiencing 120% - 135% slowdown for 8 threads relative to the sequential implementation. Despite significant debugging efforts, the reason behind this slowdown was not discovered.

It is possible that this is due to a bug with numerical instability, causing long convergence times due to incorrect communication. It is also possible that the message-passing protocol is experience communication overheads that outweigh gains. For a 32 x 32 x 32 grid and 8 threads, the communication cost is:

$$\frac{2MN + 2KM + 2KN}{MNK} = \frac{6 * 16 * 16}{16 * 16 * 16} \approx 0.375$$

which is more than a third of the computation. It is likely that this method would only scale for extremely large grids, which was beyond the experimentation bounds of this project.

Therefore, we opted to switch to a less communication-heavy method using the shared-memory model of OpenMP. The particular observation was that any element sharing between blocks 5 was read-only. That is, any element needed by neighboring blocks is not modified by any cell but by the processor that cell is assigned to. This became a crucial part of the successful parallelization, as this prompted us to switch to the shared-memory space model of OpenMP.

The parallelization followed three major lines of thought:

- Exploiting spatial locality by aligning memory access patterns by processor (chunking)
- Parallelizing the conjugate gradient solver with `reduction` clauses over the beta and alpha step-size variables, and using parallel for loops to independently update every cell.
- Taking advantage of the read-only shared memory to read and update current processor memory in chunks at the end of every iteration

Thus, we blocked the assigned cells by memory location, reordering all loops to ensure the flat index alignment of the pressure vector was contiguous with respect to loop traversal, and parallelized with respect to cell index, computing each cell independently in parallel loops.

5.4 OpenMP Results

We also explored dynamic and static chunking scheduling types 7, confirming that dynamic scheduling was more efficient. This is likely due to the varying per-cell computational complexity based on active-neighbor count discussed previously. This uneven distribution of work may also be exasperated by the alignment of blocks, since corner blocks will have significantly more non-fluid neighbors than inner blocks. However, our grid blocking was large enough that all blocks were corner blocks.

- **8 Threads (Dynamic Scheduling)**: solve_pressure $\approx$ 4.3747 sec
- **8 Threads (Static Scheduling)**: solve_pressure $\approx$ 4.4769 sec

Our implementation was able to achieve near linear speedup for 8 threads, falling just under 8x 8 speedup over the sequential, preconditioned implementation. However, thread contention slowed performance beyond 8 threads - indicating that the efficiency of hyperthreading was not sufficient

to mask its overhead, possibly due to the extreme spatial locality of the processed data leading to relatively few data stalls 9.

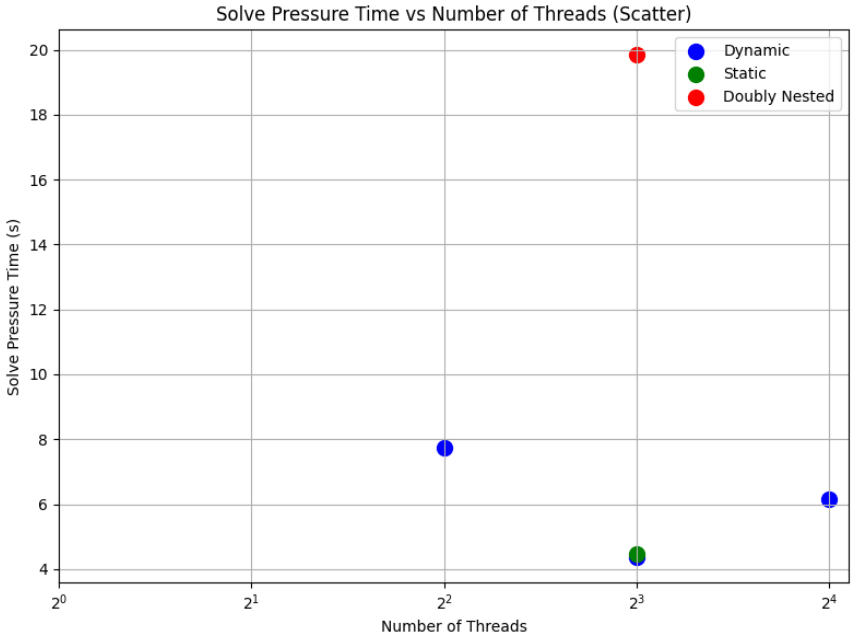

Figure 7: Scatter plot of dynamic vs static vs nested parallel loop scheduling.

6 Results

Table 3: Profiling Results with OpenMP (Dynamic Scheduling)

Operation	**4 Threads (s)**	**8 Threads (s)**	**16 Threads (s)**
Advect	0.002320	0.002184	0.002271
External Forces	0.001202	0.001212	0.001239
Solve Pressure	7.754984	4.374669	6.156234
Transfer From Grid	0.071533	0.070665	0.074848
Transfer To Grid	0.201497	0.204107	0.217347

System:

- Machine: 8-core CPU (x86_64), GHC Shark Machines
- Fluid domain: 32^3 grid, $\sim$ 2000 particles.

Metrics:

- Wall-clock time of pressure solve.

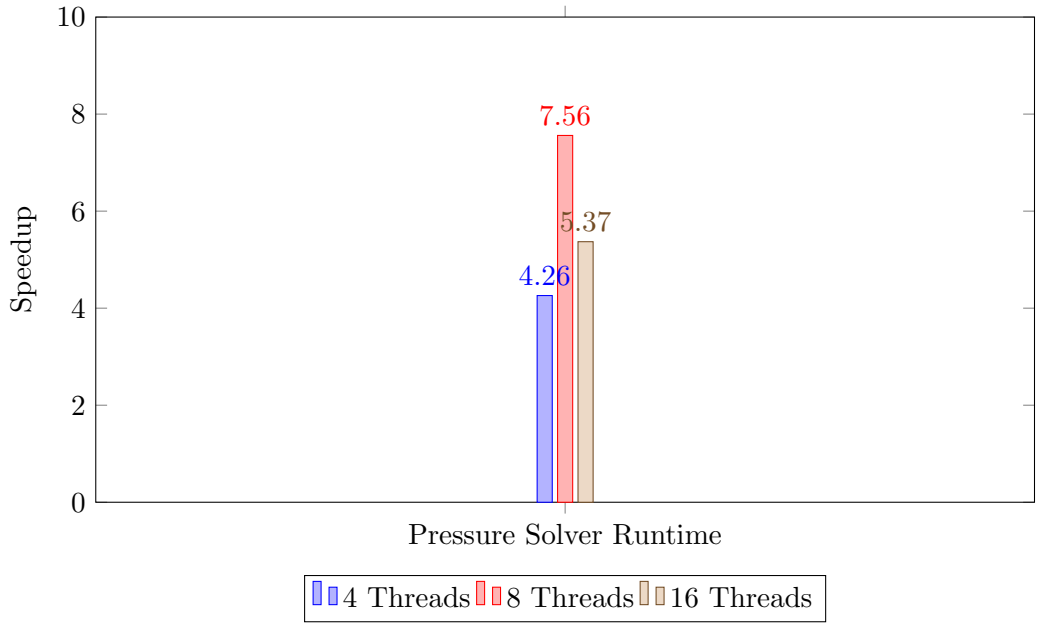

Figure 8: Speedup of each simulation stage relative to sequential implementation

7 References

References

[BKR88] J.U. Brackbill, D.B. Kothe, and H.M. Ruppel. "Flip: A low-dissipation, particle-in-cell method for fluid flow". In: *Computer Physics Communications* 48.1 (1988), pp. 25–38. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(88)90020-3. URL: https://www.sciencedirect.com/science/article/pii/0010465588900203.

[Bri15] Robert Bridson. *Fluid Simulation for Computer Graphics*. Second Edition. 2015. URL: https://github.com/mordak42/fluid-simulation/blob/master/doc/Fluid%20Simulation%20for%20Computer%20Graphics%2C%20Second%20Edition.pdf.

[Wu+18] Kui Wu et al. "Fast Fluid Simulations with Sparse Volumes on the GPU". In: *Computer Graphics Forum* 37.2 (2018), pp. 157–167. DOI: 10.1111/cgf.13350. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13350. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13350.

[20] . *FLIP Fluid Simulation Video*. YouTube Video. 2020. URL: https://www.youtube.com/watch?v=VddQZH_Ppd0.

[Dan22] Luke Dan. *libfluid: A Lightweight FLIP Fluid Simulation Library*. GitHub repository. 2022. URL: https://github.com/lukedan/libfluid/tree/master.

8 Work Division

My partner and I ended up going in pretty different directions almost immediately after our first meeting. We shared ideas, and had good conversations, but all the work (both code and analysis) described in this paper and on the website is entirely my own.

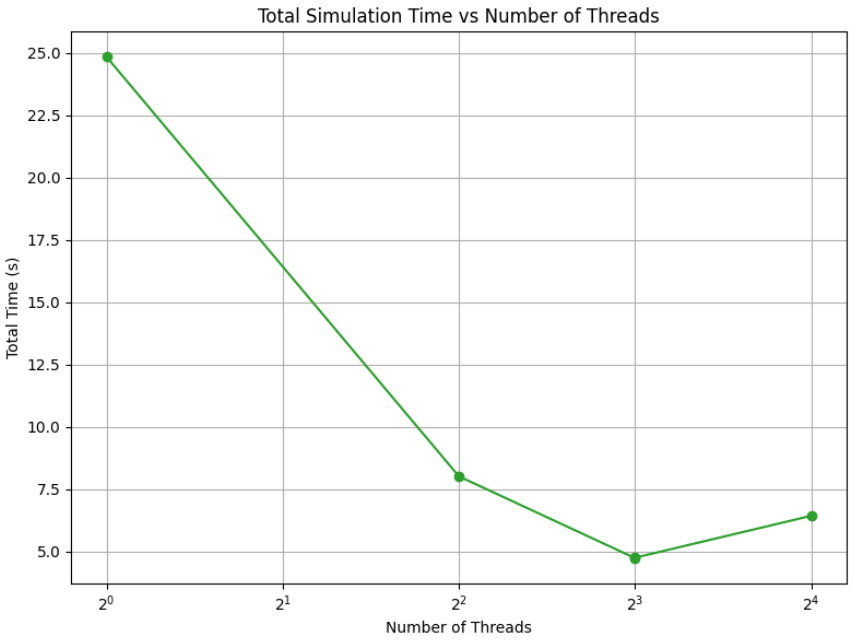

Figure 9: Speedup vs Number of Threads (OpenMP)