15-418 Project

Parallel Simulation of Blood Flow through Human Vessel

Final Report

Josh Rong (dingchar), Vincy Zheng (yuxinzhe)

Summary

We implemented a simulation of blood cells traveling through a blood vessel in CPP on CPUs. We parallelized the computation in various ways such as over individual cells and across batches (statically or dynamically assigned) with openMP and parallelization across vessel regions with MPI. The simulation takes in a Python-generated text file containing random cell coordinates and outputs the total simulation time, the order in which the cells exited the simulated section of the vessel, the number of collisions, and cell positions at each time step, all of which are used to create the visualization. Given the speedup of various parallelization strategies, we've demonstrated the tradeoffs between computation workload and synchronization overhead.

1 Background

We developed a parallelized simulation focusing on the calculation and visualization of blood flow within human vessels, which prioritized developing efficient parallel algorithms for calculating blood particle positions and velocities.

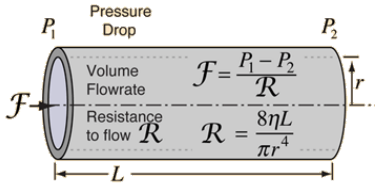
1.1 Algorithms

Two of the most important data structures we've declared are "cell" and "vessel", both of which get their field values from the input file and command line argument. The cells are then assigned to an array and get shared between or passed between processes. At each time step, the

simulation would first check collisions between each pair of cells and between the cell and vessel wall. Upon valid collision, the impact on each cell is modeled through elastic collision with 3D vector calculus. Furthermore, to incorporate more elements into the simulation and have a more accurate model, we've included the impact of Poisuille's Law and the velocity profile into the simulation.

1.1.1 Poiseuille's Law

Poiseuille's Law is fundamental in predicting the laminar flow of an incompressible and Newtonian fluid through a cylindrical pipe. In our context, this law helps in determining the pressure drop across the vessel as a function of the flow rate, fluid viscosity, and vessel dimensions. This relationship is crucial for simulating how changes in vessel diameter or viscosity impact blood flow, influencing the dynamics of cell interactions within the flow.



$$\text{Volume Flowrate} = \frac{\text{Pressure difference} \times \text{radius}^4}{\frac{8}{\pi} \text{ viscosity} \times \text{length}}$$

Figure 1: flow rate equation demonstration

1.1.2 Velocity Profile

The velocity profile of the fluid within the vessel is not uniform; velocity is highest at the center of the vessel and decreases towards the walls due to friction. This parabolic velocity profile is modeled after real-life blood flow dynamics in vessels, allowing the simulation to more accurately reflect the speed at which cells move depending on their position within the vessel.

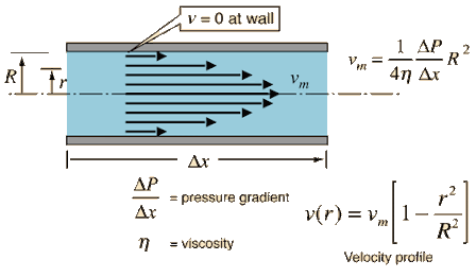


Figure 2: velocity profile equation demonstration

1.1.3 Collision between the Blood Cell and the Vessel Wall

This aspect of the simulation focuses on the interactions between blood cells and the vessel wall. When a collision occurs, the model calculates the elastic response of the cell, incorporating potential deformations and subsequent movement patterns.

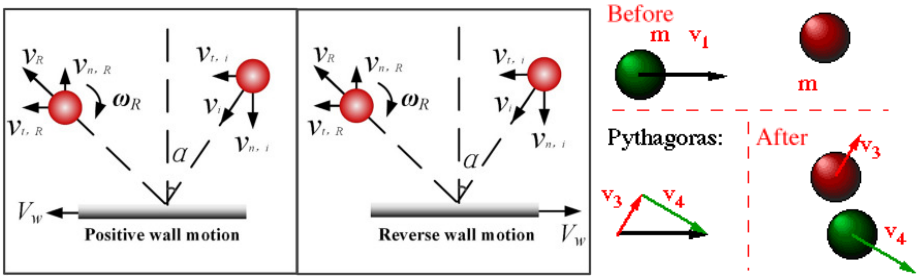


Figure 3: collision with wall and between cells demonstration

1.1.4 Collision between the Blood Cell and the Blood Cell

Similar to wall collisions, cell-cell interactions are modeled using principles of physics that govern elastic collisions. This part of the simulation assesses how cells rebound off each other, simulating realistic scenarios like blood flow in narrow capillaries or high-shear environments. Each collision is computed considering the 3D vectors of velocity.

1.2 Inputs & Outputs

1.2.1 Inputs

The simulation is configured primarily through command-line inputs and file inputs:

Command-line Inputs: These include the number of processing threads, pressure differences across the vessel, fluid viscosity, and the simulation batch size. These parameters allow users to adjust the computational intensity and physical properties of the simulation environment according to specific research needs.

File Inputs: These consist of vessel dimensions (radius and length) and the initial positions and velocities of the blood cells. These inputs are specified in a structured file format, facilitating easy modifications and batch processing.

Python Script Configuration: Within the accompanying Python script, we define the permissible ranges for blood cell sizes. This script is responsible for generating input files that simulate various scenarios by varying blood cell sizes, enabling comprehensive testing and analysis.

1.2.2 Outputs

The outputs of the simulation are recorded for subsequent analysis, validation, and visualization:

Timing Metrics: The simulation logs include initialization time, computation time, and total runtime. These metrics are crucial for performance evaluation and optimization efforts.

Cell Dynamics: We track the sequence in which cells pass through the vessel and the total number of collisions occurring during the simulation. This data is vital for understanding cell interactions and flow dynamics within the vessel.

Positional Data: The position of each blood cell is recorded at every iteration. This high-resolution data is instrumental in detailed trajectory analysis and aids in the visualization of blood flow through the vessel.

Validation and Visualization: The collected output data are used for validation checks to ensure the simulation's accuracy and reliability. Additionally, the data supports advanced visualization

techniques, providing intuitive displays of the simulation results for further analysis and presentation.

1.3 Data & Algorithm Analysis

1.3.1 Data Dependencies

In our simulation, each iteration involves checking for collisions between blood cells, which introduces data dependencies within each iteration. However, between successive iterations, there are no dependencies because all blood cells are updated simultaneously at the end of each iteration. This setup ensures that the state of each cell at the beginning of an iteration is independent of the previous iteration, allowing for more straightforward parallel execution.

1.3.2 Parallelization Potential

The most computationally intensive part of the simulation is the updating of velocities, which involves both collision detection and the application of physics laws to adjust velocities based on interactions. Given that each blood cell's velocity update is calculated independently, this part of the process is highly amenable to parallelization. Implementing parallel processing for this step can significantly enhance the simulation's performance by reducing the time required for each iteration.

1.3.3 Data Locality

The blood cells in the simulation are stored within an array, which improves data locality in the cache. This organization allows for efficient access patterns and cache utilization, as cells that are close in memory are often accessed sequentially during the simulation processes.

1.3.4 SIMD Execution Analysis

Although the simulation benefits from data locality, collision checking, specifically, the requirement to compare each cell with its neighbors, makes SIMD (Single Instruction, Multiple

Data) execution less suitable. SIMD is ideal for operations that can be applied uniformly across data sets. However, the varying attributes of interactions between cells (e.g., differing numbers of neighbors and varying distances) complicate SIMD implementation for collision checks.

2 Approach

2.1 System Overview and Workflow

Our implementation utilizes cpp with both openMP and MP libraries. We've targeted the GHC cluster machine which has 8 core processors. Since there are frequent collision checking and velocity updates, these may create divergent flows of data streams, which might be problematic to run with SIMD. We decided to examine the difference in speedup with different input sizes, batch sizes, and numbers of threads. In particular, we aim to benefit from the concurrency of those machines with varied groups of data structures. Based on the discussion in section 1, each time step is simulated all at once, and consecutive iterations depend on the previous one, in terms of cells' final velocity and position at the end of the last iteration. Therefore, the position-checking algorithm and update to velocity for each time step has to happen sequentially for each cell. Nonetheless, across individual cells, or even batches of them, the computation is independent, as they all depend on the result from the previous iteration, which is available already. Therefore, for each group of elements, we would map that to a thread and schedule each thread to a corresponding core in the cluster machine.

2.2 Sequential Implementation

To start, we have listed out all the elements of the simulation and sequentially called each. To simulate each time step, we have used a while loop. Each iteration checks collision with the cell wall and all other cells first. The final velocity after collisions is calculated with the algorithms

discussed in section 1 and recorded. The following collision is the calculation of the velocity profile and flow speed. As a result, we end up with four different velocity metrics: initial velocity defined in the input text file, velocity profile and flow speed calculated, and a final velocity which is the sum of all other velocity metrics. To keep the impact of initial velocity, we've separated it from the other components in the calculation. After the velocity is updated through collisions, we subtract the velocity profile and flow speed from the last iteration it and add the new velocity profile and flow speed calculated from the current iteration. At the end of each iteration, we check if all the cells have exited out of the vessel, which signals the end of the simulation.

2.3 Optimized-Sequential Implementation

After implementing the sequential version, we had an idea to optimize before moving on to developing parallelization strategies. In particular, in our original code, we brute-forced through collision checking, meaning that we would check the collision condition of a cell with every other cell. This has a cost on the scale of O(n^2), and it is repeatedly called at each time step until the end of the simulation. Though a decent amount of cells would have already exited through the cell in the middle of the simulation, our algorithm would still redundantly check their collisions. As a result, this opens up two areas of improvement. First, in an effort to optimize the O(n^2) cost, we could only be checking the collisions of a cell with its neighbors instead of everybody else. We end up achieving this in one of our parallelized implementations discussed later. The other possibility is to stop checking or updating the position of a cell once it exits out of the vessel. At the moment, all of the cells' information is contained in a global array. As a result, we have simply looped over all cells in the array and added another condition checking at the end of each iteration to see if the cells have already exited. Those exited cells will be added

to a separate array. As a result, at the start of the next iteration, we would loop through all cells and check against their membership in the “exited array”. If yes, we could safely skip over this cell.

Nonetheless, the result wasn’t promising. For the inputs of 10, 100, 1000, and 10000 cells, this implementation has exhibited a significant increase in computation time from the sequential implementation. The possible reason may have derived from the cost of checking existing conditions. Since we would loop over all current cells at the start and end of every time step, this checking has a very high cost in combination with the fact that checking for membership of cells in an array is expensive. More detailed execution time will be shown in section 3.

2.4 Overcell Implementation

Recognizing the poor performance of the sequential implementations, we have moved forward to designing the most straightforward parallel implementation first. Similar to the class assignments we have done, we recognized the power of a shared address model and the use of threads to split the total workload. In our simulation, we thought that the most obvious parallelizable element of simulation would be the individual cells. Therefore, we have utilized the power of openMP and tried to spawn a new thread at each time step for every cell. Specifically, the processes of checking collision, updating velocity, and checking whether the cell has exited are all independent of each other. We’ve parallelized the simulation of each element in the global cell array and dynamically scheduled them across all cores available. In evaluating this over-cell parallelized implementation, we’ve identified another potential area for improvement: when we increase the cell number, the speedup starts to decrease, which may have resulted from the communication overhead in scheduling the significant number of threads to each core. More details are shown in section 3.

2.5 Batch-Static Implementation

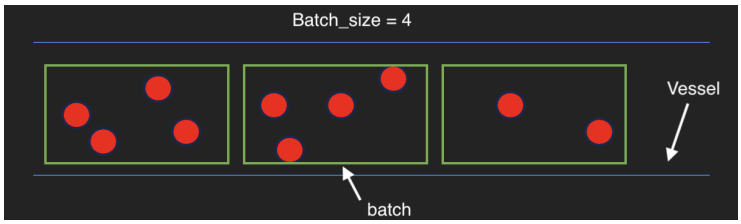


Figure 4: batch-static implementation visualization

In response to the first area of improvement discussed in section 2.3 and similar discussions from section 3.4, we've come up with a new implementation where we could limit the frequency of collision checking and the number of total threads by assigning groups of cells to each thread and only check their mutual impact within the group. In here, we have decided to sacrifice correctness by a little bit to examine the improvement in speedup. In particular, we have split the cells into batches based on the input argument "batch_size" and the initial position of each cell along the z-axis, which is the main direction of flow. We assumed that those cells that are nearby at the simulation would remain close enough to each other throughout the simulation since the components of travel along other directions are minimal. In comparison to the over-cell implementation, our best performance design so far, the speedup is overall comparable. This implementation even has a higher speedup at a thread count of 8. However, noticing that we've sacrificed correctness, we are trying to keep the basic framework while dynamically modifying the batches to guarantee accuracy.

2.6 Batch Implementation

To resolve the aforementioned problem with correctness, we modified the code so the batches could be reassembled at the end of each time step. In particular, with the same assumption that only cells nearby would collide with each other at any given moment, we've looped over all cells and grouped them based on the z-positions of each cell. To our surprise, the overall performance

is very poor, with more detail in section 3. We've thought about the potential cost of regrouping cells into batches at the iteration but never imagined it being so huge.

2.7 MPI Implementation

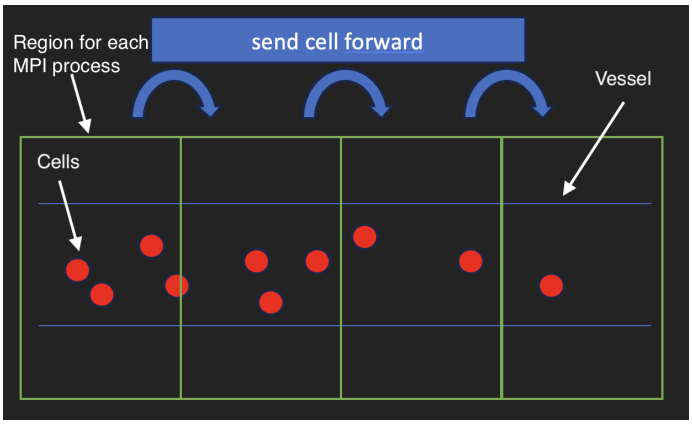


Figure 5: Message-Passing implementation visualization

In the end, we thought about an approach to minimize the cost of regrouping cells. With MPI, we could divide the entire vessel into separate regions based on the number of processors available each having its distinct address space. This would behave similarly to grouping cells into batches, but each processor would have a varied amount of cells wor compute. The most important optimization focuses on the amount of information being transferred at each time step. Each processor would only need to identify the cells that are no longer in their designated region. At the start of the simulation, the thread with pid 0 will read the input file and construct the "cells array" as we've added for all previous implementations. All other threads will blockingly receive from thread 0 and construct their own cell array. All the threads will enter the simulation phrase with their own cells. At each time step, each processor will only compute the collisions and update the velocity of all cells it has information about. The message-passing only happens at the end of each iteration where each thread will forward cell information of those whose z-position exceeded their bounds to the next thread (pid+1), and remove those cells from their array. Also,

each processor will have a variable denoting whether the region is completely empty or not. Thread 0 will gather such information and broadcast it after it processes it. If all regions are empty, the simulation will end. From this, we see a very promising computation speedup, whereas the overhead is much more than what we imagined. More details in section 3.

2.8 Validity Checker

To ensure the accuracy of our algorithm, we have implemented a validity checker that performs a comprehensive evaluation of the simulation results as detailed in the output file. This validity checker focuses on several critical aspects:

Complete Passage of Blood Cells: It verifies that all blood cells successfully exit the vessel, ensuring that no cell is left behind or incorrectly accounted for in the simulation process.

Correct Order of Exit: The checker confirms that the sequence in which blood cells exit the vessel aligns with the expected order. This is crucial for simulations where the order of events may influence subsequent outcomes.

Validation of Positions During Execution: Throughout the simulation, all positions of the blood cells are within the permissible boundaries of the vessel. This ensures that all simulated movements adhere to the defined physical constraints of the model.

These checks are essential for confirming that the simulation behaves as intended and provides reliable data for further analysis and decision-making processes.

2.9 Visualization

To enhance the presentation and clarity of our simulation results, we implemented a visualization feature that dynamically represents the movement of blood cells through a vessel. Utilizing Matplotlib's 3D plotting capabilities, we plot the trajectory of each cell based on its position and radius data at every iteration, rendering these trajectories in three dimensions to provide a clear

visual context. This visualization is then converted into a GIF animation, sequentially displaying the progression of cells through the vessel.

This approach not only makes the results more accessible and easier to understand but also allows for a visually engaging way to demonstrate the simulation's accuracy and the dynamic behavior of the cells within the vessel. Error handling is integrated to ensure reliability, making this visualization a valuable tool for analyzing and presenting complex simulation data.

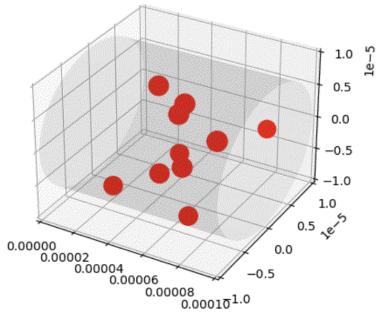


Figure 6: 3-D visualization of sequential implementation with 10 number of cells

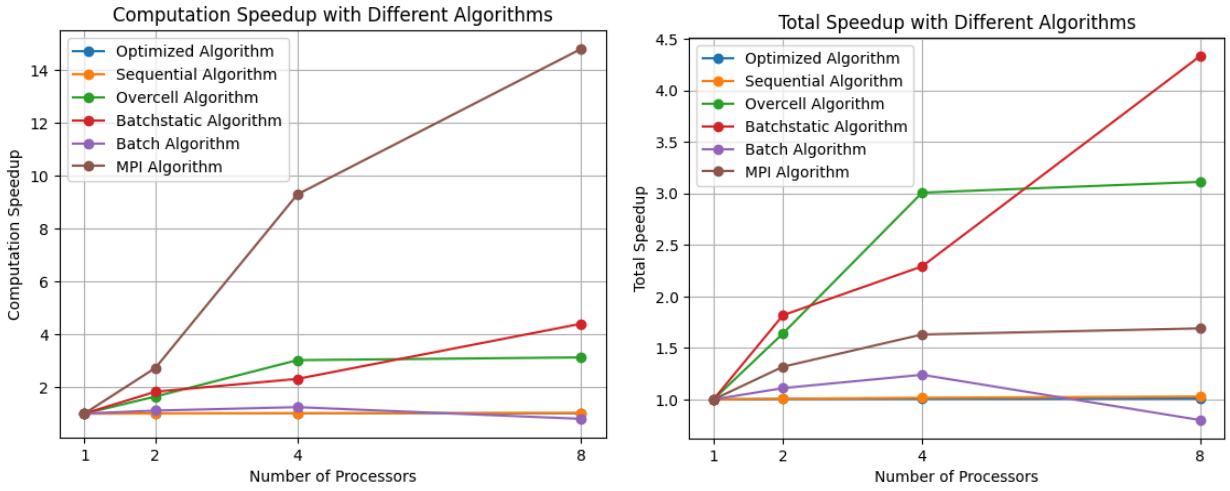
3 Results & Analysis

3.1 Environment Overview

Our simulation takes a text file as input which has a size count of 10, 100, 1000, and 10000 cells. The simulation takes 5000 as the pressure difference between the two ends of the vessel and 100 micrometers as the vessel length, but the length is abstracted as a small section within a very long vessel so that the impact of the pressure difference is controllable. It also restricts the diameter of the vessel to 10 micrometers and that of the blood cells to 2-3 micrometers. Our baseline model is the computation and total execution time of the sequential implementation. The performance we utilized is execution time in the wall-clock time of each implementation and the speedup relative to the baseline model.

3.2 Performance with Number of Processors

(complete speedup graphs for each implementation in the Appendix)



Figures 7 & 8: Computation and total speedup with all implementations with input size of 1000 and batch_size = 8

Although some of the above analyses are hypothetical, the execution profile will be discussed in the following section. As can be seen from Figure 1, the computation speedup of the MPI algorithm is significantly better than all other implementations. This could have resulted from the fact that with private address space, there won't be contention to cell updates and collisions. At the same time, each region is designated to only a particular processor. This eliminates the need to do context switching and fetching for each processor. The speedup increases continuously on the graph, exceeding 14 times the performance of the baseline model. In general, this implementation scales very well with the number of processors we use, which meets our expectations. The communication is limited to the end of each iteration and there are very few messages being transferred - only between neighboring threads and to thread 0.

Next, we see a comparable performance from the over-cell and batch-static implementation. Both of them have around two to four times the speedup as the baseline implementation. The speedup comes from doing the independent work of each cell or group of cells at every iteration. Both implementations exhibit an overall strictly increasing trend in terms of computation

speedup as the number of processors increases. Nonetheless, the performance of the over-cell implementation at a processor count of 8 seems to plateau. This could have originated from the fact that the overhead of dynamically scheduling threads to each individual processor accumulates. With more threads, there would be, overall, more overheads of fetching threads from the work queue or task stealing. This could also be the reason for all those openMP-implemented versions to perform far worse than the MPI implementation. Meanwhile, the batch-static implementation seems to have scaled well with the number of threads. Since the cells are grouped into batches and assigned to threads at the start of the execution, we wouldn't suffer as much from the overheads of fetching at every iteration. Nonetheless, though this implementation is able to pass our checker, when we consider the accuracy of a larger and more complicated model, we might want to stay away from this approach.

Lastly, we have the batch algorithm struggling to have a better performance than the two sequential implementations. The reason originates from the large synchronization overhead, which is also discussed in section 3.4 with the execution profiles. At the end of each iteration, we are asking the program to go through all the cells again and group them into new batches. The cost of doing it significantly outweighs the benefit of parallelization. At a thread count of 8, we even see the performance of it being lower than that of the sequential version. Thus, we speculate that, with increasingly more threads, the performance of this algorithm may continue to get worse beyond 8 threads.

Almost all the trends we've discussed about computation speedup apply to total speedup besides the message passing implementation. The total speed-up dropped significantly to below 2 times across all processor counts. This is the result of the huge initialization setup cost for the implementation, which is also discussed and shown in section 3.4. With a consistent but

enormous cost to set up the point-to-point messaging channel, programming model as well as the synchronization overhead in communicating to share the cell information, the total execution time of this implementation varies not as significantly across all thread counts. While almost all the implementations with the shared address space model have very good load balance, we could also have stuffed from workload imbalance for message passing implementation. At the start of the simulation, there would for sure be more cells at the beginning of the vessel, which are mostly assigned to a particular processor (i.e. pid = 0). In the end, most cells would be around the end of the vessel to exit, which leaves processors such as the one with pid = nproc - 1 very busy. During that time, the other processors all have less work to do or simply sit idle. This imbalance wastes computation resources and further hurts our total speedup.

3.3 Performance with Input Size

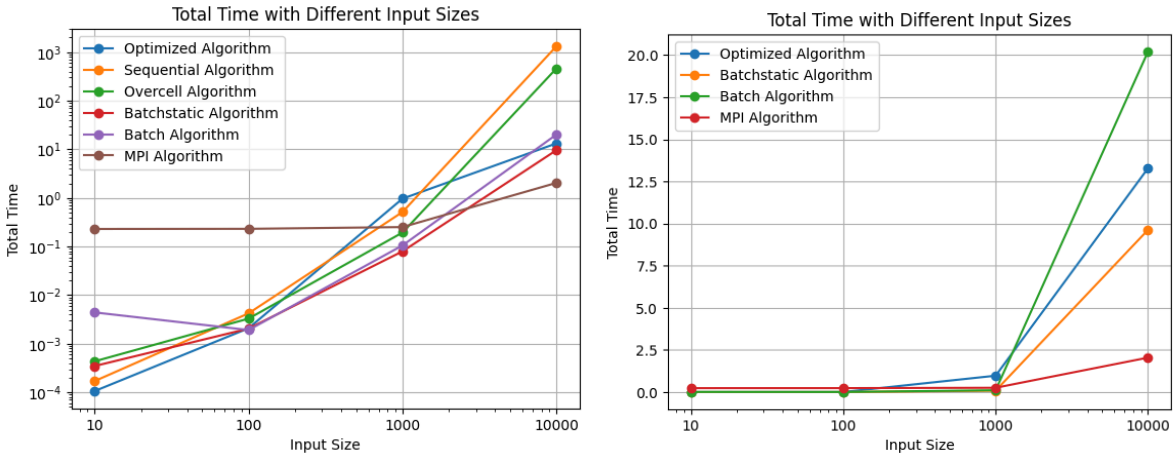


Figure 9 & 10: Total Time with all implementations with 4 threads and batch_size = 8

According to the graph, as the input size increases, the total time increases. At the smallest input size tested, the Sequential, Optimized, Overcell, and Batchstatic algorithms demonstrate similar total times. This convergence is likely attributed to the minimal computational demand at this scale, where overhead and initialization times dominate the total execution time. Unlike these implementations, the Batch algorithm is slower, possibly due to its dependency on dynamic work

assignments based on the configuration of blood cells, introducing additional overhead. The MPI (Message Passing Interface) algorithm shows notably poor performance for small input sizes due to the significant initialization overhead involved in setting up MPI processes.

As the input size increases, the Sequential and Overcell algorithms exhibit drastically longer execution times. The Sequential algorithm, lacking parallelism, struggles with the large volume of computations, resulting in prolonged execution times. In contrast, the Overcell algorithm, while utilizing parallel processing, suffers from high overhead possibly due to extensive context switching among threads, each handling different blood cell calculations. To emphasize the performance of more scalable algorithms, the right-hand graph omits the Sequential and Overcell algorithms. Here, the MPI algorithm stands out, displaying superior performance as it scales efficiently with larger input sizes. This scalability is indicative of optimal resource usage and reduced contention typical of supercomputing environments. The MPI implementation initially underperforms with small input sizes due to its setup costs but excels with larger data sets, showcasing its scalability. This derives from the fact that message passing has a large setup cost at the start of the simulation due to resource allocation, discussed further in the section below. Nonetheless, such a huge cost does not depend on input and simulation time. Thus, this one-time cost could be amortized with a larger workload and exhibit better speedup as input increases.

3.4 Execution profile of each implementation

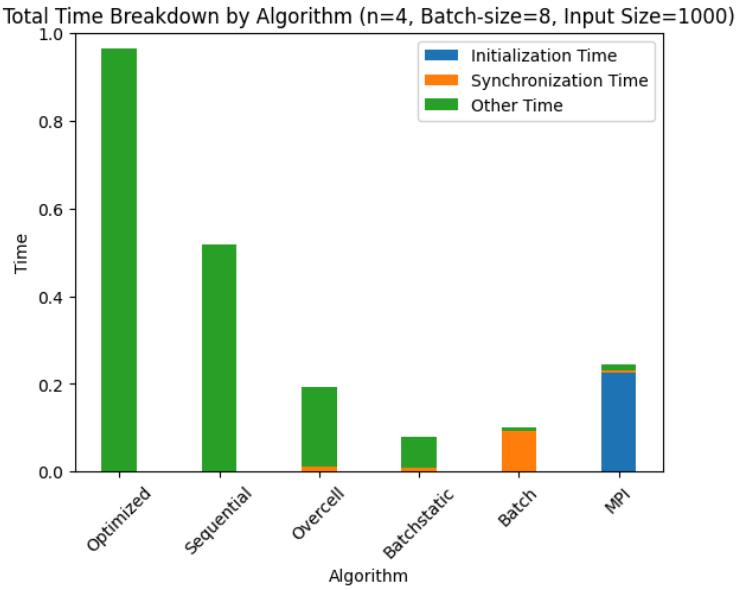


Figure 11: execution profile of each implementation

We further gathered information about the different costs in terms of execution time in all of the implementations. As can be seen from the figure above, we have recorded the overall execution time and divided it into three sections: synchronization overhead, initialization, and the remaining computation time. On the left, we see that the two sequential designs have very minimal initialization and synchronization time compared to the actual computation time, which results in a mostly green-colored column. In the middle, we have over-cell and batch-static implementations that have relatively the same distribution of costs. With both of the implementations having similar overall execution times, both of them have similar synchronization times as can be seen from the yellow portion of the columns. Both of them would suffer from the overheads of dynamically scheduling the threads to each processor in trading off with work balancing and parallelization. Similarly, batch implementation also suffers from the aforementioned overhead. Nonetheless, this implementation also suffers the cost of dynamically regenerating each batch at the end of each iteration, which is an increasingly more

crucial contribution to the total execution time as the input size increases. As a result, the column for batch implementation is mostly dominated by synchronization overhead. Lastly, we have the profile of message-passing implementation on the very right. As can be seen from the figure, the synchronization time is non-trivial and is almost comparable to the computation time, which indicates the overhead and stalling for sending and receiving messages between each processor. The synchronization time could also suggest a large network latency or memory bandwidth constraint in communication. Nonetheless, the entire column is dominated by the initialization time, which is unprecedented in all the previous implementations. Since each processor has its own private address space, it would require initial resource allocation to all processors. In addition, to enable message passing between processors, we would need to set up communication channels between each processor.

Overall, this would be a very high cost, but a one-time cost that could be amortized with large input and thread count as seen in previous sections.

4 Conclusion

From this project, we have examined the differences in performance across all the implementation strategies. In particular, we see that most of the implementations with shared address space model and with openMP exhibit good work balancing but require scheduling overhead for each processor. However, they tend to not scale as well as the message-passing implementation. Although it suffers a huge initialization cost, such cost is amortized over large input workloads, which shows evidence of the scalability of the message-passing programming model and explains why it would be adopted for many large-scale supercomputers. Overall, we still believe that the use of CPU is sound as the control flow of our program isn’t fixed until

execution time, which may create control flow divergence if it were to be run on a GPU. If we had more time, we would have tried combining message passing and shared address space programming model. Seeing the promising computation speedup of message passing implementation, we could further optimize it by parallelizing the cells within each region enclosed with openMP individually or even in groups.

5 Reference & Credits

5.1 Reference

Halliday, D., Resnick, R., & Walker, J. (n.d.). *Poiseuille's Law*. Retrieved from https://www.lon-capa.org/~mmp/kap6/cd155.4.htm

Nave, R. (n.d.). *Pipe friction calculation*. HyperPhysics. Retrieved from http://hyperphysics.phy-astr.gsu.edu/hbase/pfric2.html

Wikipedia contributors. (n.d.). *Hagen–Poiseuille equation*. In *Wikipedia, the free encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Hagen%E2%80%93Poiseuille_equation

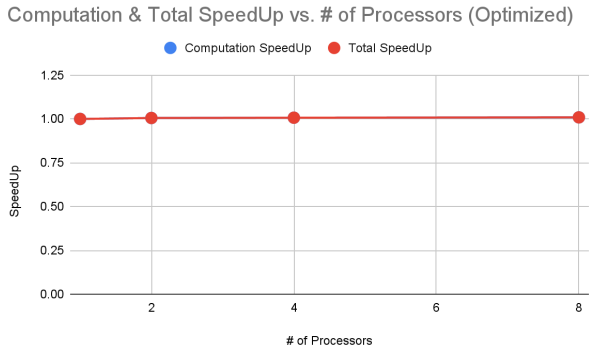
Author, A. A., Author, B. B., & Author, C. C. (2023). Title of the article. *Journal Name, Volume*(Issue), pages. https://doi.org/10.1016/j.sciencedirect.2023.009403

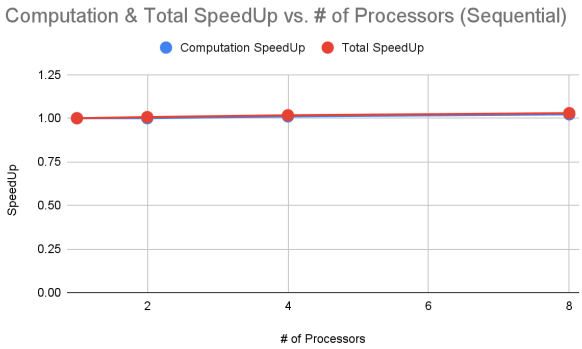
5.2 Credits

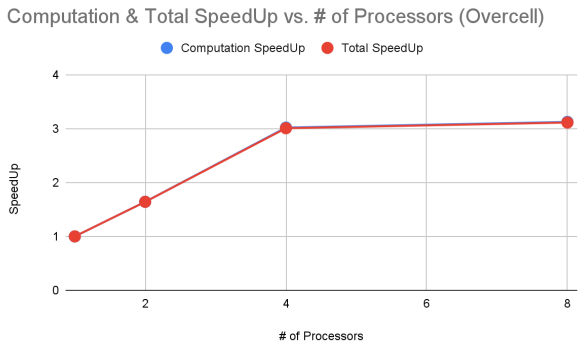
Josh Rong (50%): Implementation of sequential version; Implement parallelization algorithm over batches of cells using shared address space model (dynamic vs static assignment); Implement parallelization algorithm over individual sections of the vessel using shared address space within each section and message passing between sections; Analyze differences in performance gain between parallelization strategies; Write the report and make the poster.

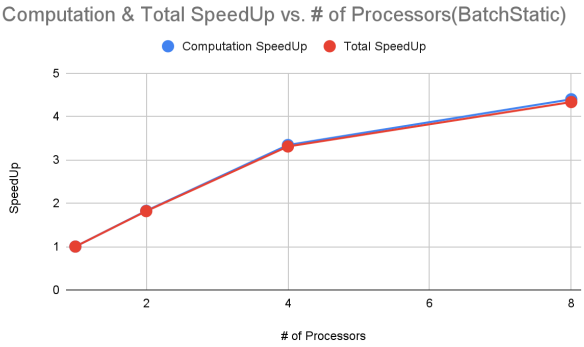
Vincy Zheng (50%): Implementation of sequential version; Implement parallelization algorithm over individual cells using shared address space model (dynamic vs static assignment); Implementation of a checker to verify the validity of the program with information extracted at each timestep; Generate speedup graphs with the performance of sequential version and all parallelized implementations; Implementation of visualization; Analyze differences in performance gain between parallelization strategies; Write the report and make the poster.

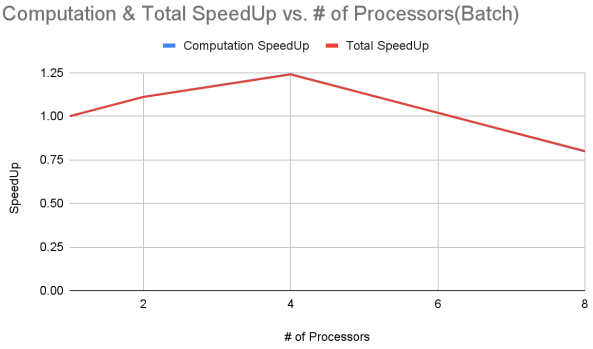
Appendix: SpeedUp Graphs

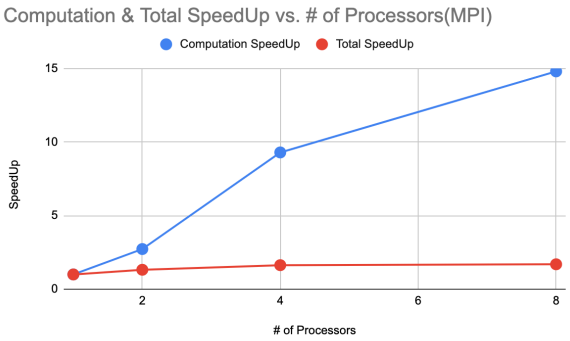


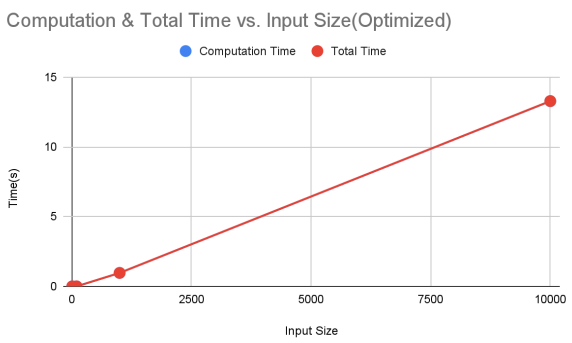


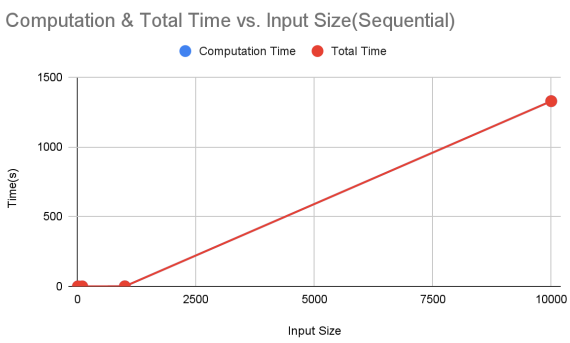


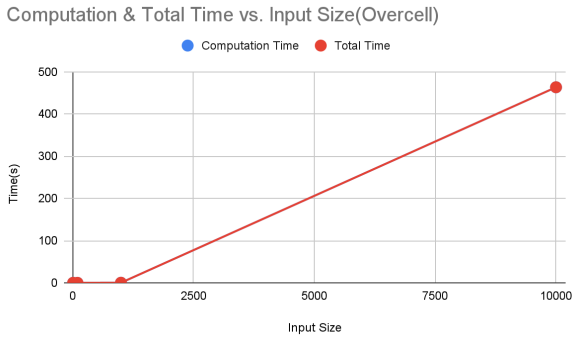


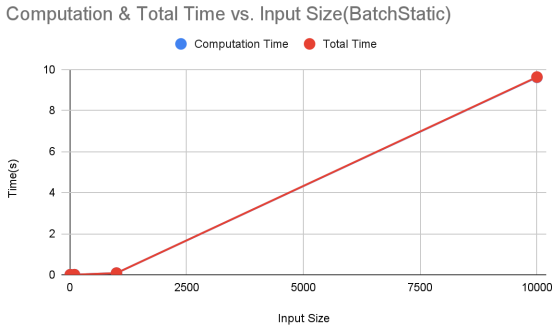


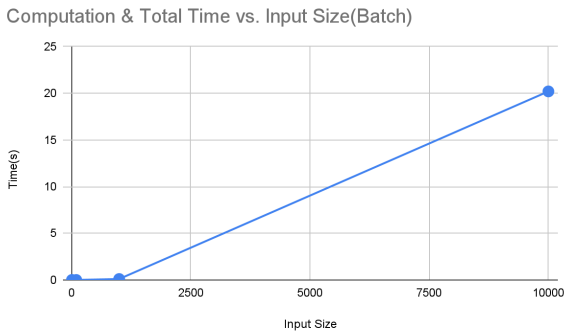


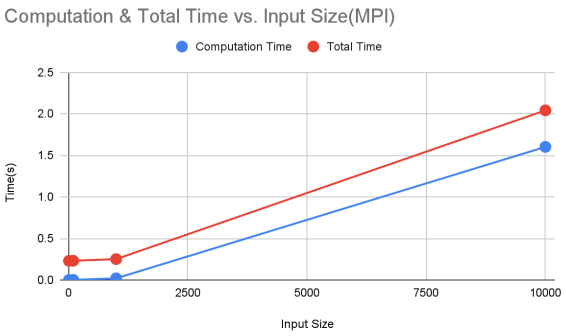

Computation & Total Time vs. Input Size(Optimized)
Computation Time
Total Time
Time(s)
15
10
5
0
0
2500
5000
7500
10000
Input Size


Computation & Total Time vs. Input Size(Sequential)
Computation Time
Total Time
Time(s)
1500
1000
500
0
0
2500
5000
7500
10000
Input Size


Computation & Total Time vs. Input Size(Overcell)
Computation Time
Total Time
Time(s)
500
400
300
200
100
0
0
2500
5000
7500
10000
Input Size


Computation & Total Time vs. Input Size(BatchStatic)
Computation Time
Total Time
Time(s)
10
8
6
4
2
0
0
2500
5000
7500
10000
Input Size


Computation & Total Time vs. Input Size(Batch)
Time(s)
25
20
15
10
5
0
0
2500
5000
7500
10000
Input Size


Computation & Total Time vs. Input Size(MPI)
Computation Time
Total Time
Time(s)
2.5
2.0
1.5
1.0
0.5
0.0
0
2500
5000
7500
10000
Input Size