Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Applications of Parallel Computing in Chorin's Fractional Step Methods for solving Incompressible 2D NS Equations

Boyuan Lu

December 18, 2022

Abstract

This report presents the comprehensive details of implementing parallel computation on the two-dimensional Navier-Stokes equation. Chorin's fractional step method is chosen as the numerical method for the solver of NS equations. Three different implementations of the solver are delivered and discussed in this report: a non-parallel implementation, an OpenMP-based parallel implementation, and a CUDA-based parallel implementation. The performances of these three implementations are evaluated based on a simple computational fluid dynamics scenario: a lid-driven cavity. Throughout this project, we wish to enhance the understanding of parallelism using two different approaches: fine-grain parallelism for CUDA and coarse-grain parallelism for OpenMP.

Link to Final Project git repo: https://git.doit.wisc.edu/BLU38/repo759

Contents

1 General Information

- Civil and Environmental Engineering, Water Resource and Fluid Mechanics.
- PhD student
- I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party

2 Problem Statement

The Navier-Stokes equations are groups of partial differential equations describing the movement of Newtonian fluid in a given space and time. The formation of NS-equation are two parts: continuity and momentum. Assuming two-dimensional, incompressible, constant viscosity, the controlling equation can be expressed as following equation groups:
Continuity:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{1}$$

Momentum in x and y direction:

$$\frac{\partial u}{\partial t} + \frac{\partial uu}{\partial x} + \frac{\partial uv}{\partial y} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) \tag{2}$$

$$\frac{\partial v}{\partial t} + \frac{\partial uv}{\partial x} + \frac{\partial vv}{\partial y} = -\frac{1}{\rho}\frac{\partial P}{\partial y} + \nu(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}) \tag{3}$$

The equations are really difficult to solve analytically since both u,v and P appears in everywhere of the equations, which makes the equations nonlinear and complex. Thus, people usually solve the NS equations using numerical methods.

This mixed equation can be solved in many efficient and smart ways. In this project, we will choose the fractional step methods (FSM) as a primarily numerical method to solve the 2-dimensional incompressible Navier-Stokes equations. The idea of fractional step methods (FSM) is to split up the equations into their constituent pieces and alternate between advancing each equation in time. The simplest form of FSM with two steps can be expressed as follows:

$$U^* = N_A(U^n, k) \tag{4}$$

$$U^{N+1} = N_b(U^*, k) \tag{5}$$

where $N_A(U^n, k)$ represents some one-step numerical method that solves $u_t = A(u)$ over a time step of length k.

This project is motivated by my personal interest and is an extension of my course project on Math 714. The field of computational fluid mechanics is inextricably linked to high-performance computing.From a broader point of view, numerically solving NS equations

by FSM requires a huge amount of computation, which can be summarized as three layers of nested loops in sequential order. The first layer of loops is a step-forward loop iterating over each time step, using the idea of either explicit Euler or implicit Euler time discretization. This layer of the loop cannot be parallelized because the results of the current moment depend on the results of the last steps if we applied the explicit Euler method. On the contrary, implicit Euler can be parallelized since the step discretization can be formed as a matrix operation. In this project, we will not parallelize this level over loops since we will apply the explicit Euler method for the sake of saving time. The second layer of the loops is the predictor of velocity. This layer is perfect for parallelization because each operation can be computed independently. The third layer of loops is the corrector of poisson pressure discretization. Basically, this layer of loops is equivalent to solving a linear system, which is also suitable for parallel computing. The details of the parallel algorithm for the latter two layers will be stated rigorously in the next sessions. Overall, parallel computing is ideally suited for solving the CFD problem, and this project was initiated based on this perception.

The goal of this project is to implement the three versions of Chorin's fractional step method (no parallel, OpenMP, and CUDA) and build a specific solver for two-dimensional incompressible Navier-Stokes equations with predefined initial and boundary conditions. Beyond the implementation of the algorithm, we expect to evaluate and analyze our work by validating the result with other people's work and comparing the differences in performance among parallelisms.

3 Solution Description

3.1 Domain setup and data structure for grid

The u velocity, v velocity, and P pressure fields were stored as a matrix, which can be converted into one-dimensional squash form when using CUDA parallelism. To couple the velocities u, v and pressure P, we applied a staggered grid based on the idea of control volume methods. A staggered grid is one in which the velocities and pressures are located at different positions. Figure 1 presents the schematic layout of staggered grid units for the defined locations of velocities and pressures. Every cell has one u velocity, and one v velocity, which are defined as the surface velocities at the bottom and left boundaries of the cell, as well as the pressures that will be defined in the center of the cell.

3.2 The first parallel implementation: fractional steps for the time discretization

To discretize the partial t terms in two momentum equations in Eq.2 and Eq.3, we first applied the Explicit Euler scheme which can be written as,

$$\frac{u^{n+1}-u^n}{dt} = -u^n\frac{\partial u^n}{\partial x} - v^n\frac{\partial u^n}{\partial y} - \frac{1}{\rho}\frac{\partial P^{n+1}}{\partial x} + \nu(\frac{\partial^2 u^n}{\partial x^2} + \frac{\partial^2 u^n}{\partial y^2}) \tag{6}$$

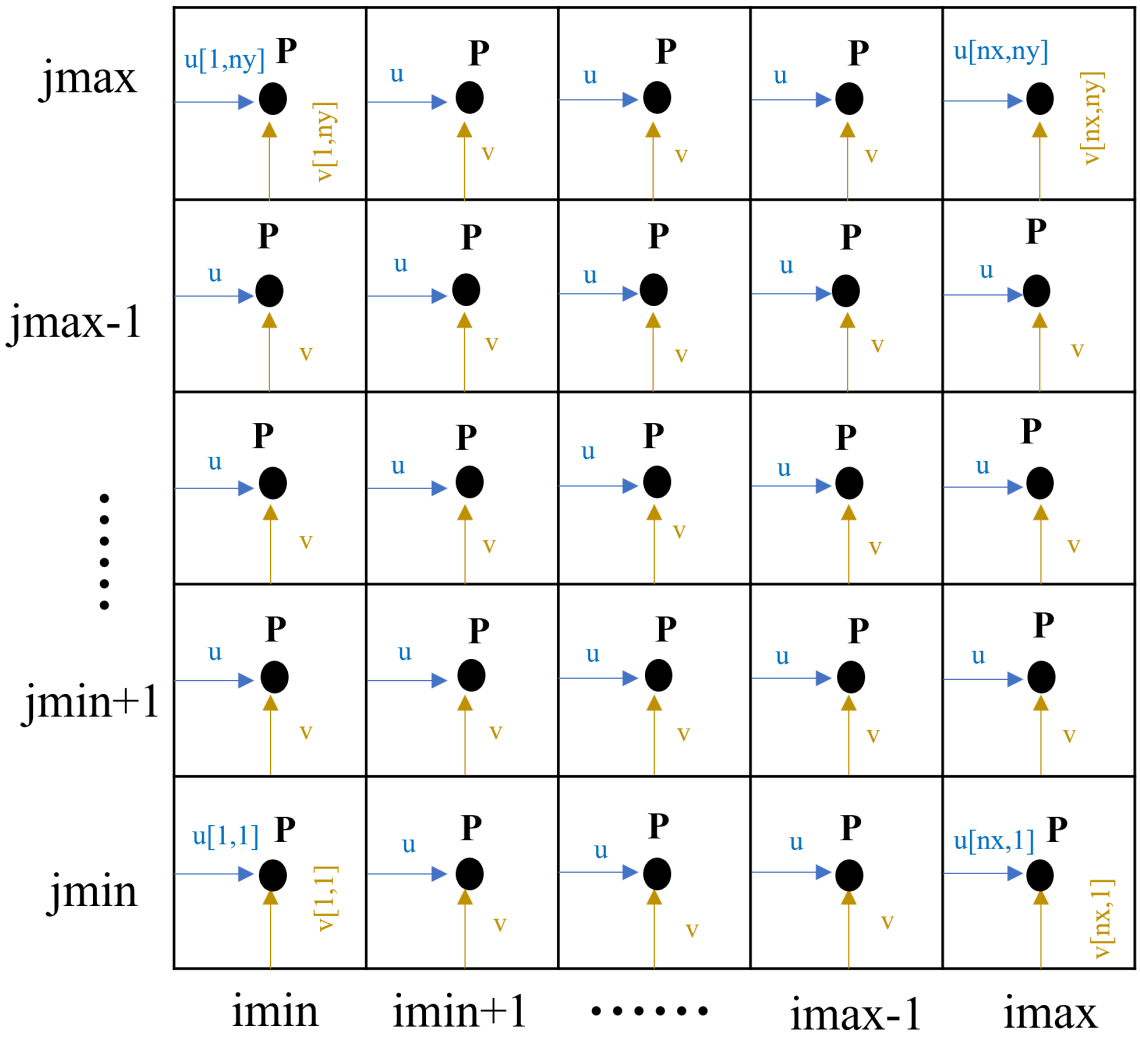


Figure 1: computational staggered grids for our CFD solver. The locations of u, v, P are presented in each cell with indices.

However, ordinary explicit Euler has a weakness in that it cannot be solved directly because P^{n+1} is not known yet, and the solution may not be guaranteed to be computed due to the explicit Euler's instability. To solve the temporal discretization, we must use predictor-corrector schemes that are coupled with Eq. 1 of continuity conservation by splitting the explicit Euler into two steps. The first step, also known as the predictor step, is to compute the intermediate velocity u^*, v^* using the Explicit Euler, but only with convection and diffusion terms included.

$$u^* = u^n - (u^n \frac{\partial u^n}{\partial x} + v^n \frac{\partial u^n}{\partial y})dt + \nu(\frac{\partial^2 u^n}{\partial x^2} + \frac{\partial^2 u^n}{\partial y^2})dt \tag{7}$$

where each derivative term can be interpreted by the central difference scheme

$$\frac{\partial^2 u^n}{\partial x^2} = \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{dx^2} \tag{8}$$

$$\frac{\partial^2 u^n}{\partial y^2} = \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{dy^2} \tag{9}$$

$$u^n \frac{\partial u^n}{\partial x} = u(i,j)(\frac{u(i+1,j) - u(i-1,j}{2dx}) \tag{10}$$

$$v^n \frac{\partial u^n}{\partial y} = \frac{1}{4}(v(i,j) + v(i-1,j+1) + v(i,j+1) + v(i-1,j))(\frac{u(i+1,j) - u(i-1,j)}{2dx}) \tag{11}$$

Calculating any arbitrary velocity (except boundary) at a given node will involve nearby nodes. If we parallelize the computation directly, the race condition will occur because there is a high chance that another thread will access the neighbor node when the current node is being calculated by the nearby neighbor node. In order to avoid this race condition, we use a buffer matrix to store the updated u^*.Therefore, an extra space of $O(n^2)$ will cost by such approach. Here are the pseudocodes for OpenMP and CUDA.
For OpenMP, the "for clause" can be a really good way to do parallel computing. Here is the pseudocode for OpenMP.

```
#pragma omp parallel for simd num_threads(threads_num)
  loop the j index
    loop the i index
        calcuate u_star use Eq.7 and store in buffer matrix
...
overwrite the old matrix with buffer matrix
```

For the CUDA, we use the "thread block" to implement parallelism. Here is the pseudocode for the CUDA.

```
launch cuda kernel for (n-block_dim+1)/block_dim grids,
each grid has block_dim*block_dim threads.
...
# kernel function
```

```
obtain the id of thread
    calculate u_star use Eq.7
...
overwrite the old matrix with buffer matrix
```

3.3 The second parallel implementation: momentum discretization for predictor step

the second step, also known as corrector step, is to solve velocities u^{n+1}, v^{n+1} at the next moment, but includes the pressure terms exclusively,

$$\frac{u^{n+1} - u^*}{dt} = -\frac{1}{\rho}\frac{\partial P^{n+1}}{\partial x} \tag{12}$$

Taking the derivative of the Eq.12, and combine with Eq.1, we can derive the gradient of pressure projects u, v onto a divergence free field that satisfies continuity:

$$\frac{\partial^2 P^{n+1}}{\partial x^2} + \frac{\partial^2 P^{n+1}}{\partial y^2} = \frac{\rho}{dt}(\frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y}) \tag{13}$$

Solving the above pde is equivalent to solve a linear system of $Ax = B$ as following,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{-1}{dx^2} & D & \frac{-1}{dx^2} & \cdots & \cdots & \cdots & \frac{-1}{dy^2} & \cdots & 0 \\ \vdots & \cdots & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots & \vdots \\ \vdots & \frac{-1}{dy^2} & \ddots & \frac{-1}{dx^2} & D & \frac{-1}{dx^2} & \ddots & \frac{-1}{dy^2} & 0 \\ \vdots & \cdots & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots & \vdots \\ 0 & \frac{-1}{dy^2} & \cdots & \cdots & \cdots & 0 & \frac{-1}{dx^2} & D & \frac{-1}{dx^2} \\ 0 & 0 & 0 & \cdots & \frac{-1}{dy^2} & \cdots & 0 & \frac{-1}{dx^2} & D \end{bmatrix} \begin{bmatrix} P(1,1) \\ P(2,1) \\ \vdots \\ P(i,j) \\ \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} R(1,1) \\ R(2,1) \\ \vdots \\ R(i,j) \\ \vdots \\ R(nx*ny, nx*ny-1) \\ R(nx*ny, nx*ny) \end{bmatrix} \tag{14}$$

Here, we propose two different approaches for solving a linear system, as well as their parallel versions in OpenMP and CUDA: the directed method (LU decomposition) and the iterative method (conjugate gradient descent). It should be noted that other methods for solving the Poisson Pressure equations, such as Gausse-Sedel or Successive Over-Relaxation, are preferable. However, due to time constraints, we are not going to include these approaches.

- Direct approach: LU decomposition LU decomposition is the most common but inefficient method to solve linear systems. By decomposing the matrix A into a lower triangular matrix and an upper triangular matrix, variables can be solved by forward substitution. Here are the algorithm for LU decomposition by OpenMP and CUDA.

  OpenMP:

```
#pragma omp parallel for simd num_thread(threads_num)
    loop the i index
        loop the j index
            update the lower triangular matrix
        loop the j index
            update the upper triangular matrix
```

CUDA:

```
launch kernel for (n - threads_num + 1)/threads_num grids,
each grid has threads_num threads.
...
# kernel function
obtain the id of thread
loop the j index
    update the lower triangular matrix
loop the i index
    update the upper triangular matrix
```

We only parallelize the outmost loops for both OpenMP and CUDA implementation, because inner loops have dependent relationships with each other.

- Iterative approach: conjugate gradient descent Besides the direct method, the iterative approach is another group of algorithms that it approximates the solution. Among these algorithms, conjugate gradient descent is well known as its $O(1)$ convergence speed. Here is the typical implementation of conjugate gradient descent.

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$

if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result

$\mathbf{p}_0 := \mathbf{r}_0$

$k := 0$

repeat

$\alpha_k := \dfrac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$

$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$

$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$

if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop

$\beta_k := \dfrac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$

$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$

$k := k + 1$

end repeat

return $\mathbf{x}_{k+1}$ as the result

Notice that $r_k^T r_k$ and Ap_k can be parallelized with the idea of reduction we learned in the lecture. Here are the OpenMP and CUDA implementations for $r_k^T r_k$.
OpenMP:

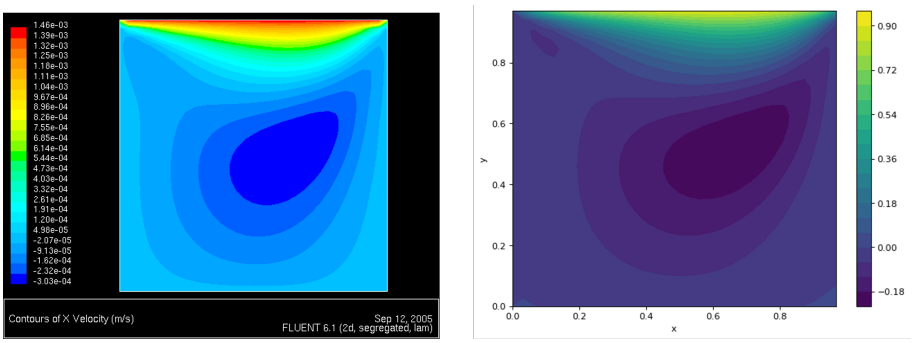
```
    int sum = 0;
#pragma omp parallel for simd reduction( +: sum)
    for(int i=0; i<n; i++){
        sum += r[i]*r[i];
    }
```

CUDA: Sorry, I haven't implemented this part yet.

At the final step of the corrector, we can update Eq.11 and Eq.12 for the u, v using the downward difference of pressure,

$$u^{n+1}(i,j) = u^*(i,j) - \frac{dt}{\rho}\frac{P(i,j) - P(i-1,j)}{dx} \tag{15}$$

This step can also be parallelized by the buffer approach since it is similar to what we have done in 3.1. Therefore, I will not illustrate the details here.



Fluent results for u velocity

Our results for u velocity

Figure 2: Comparison between Fluent results (left figure) with our results (right figure) in terms of u velocities for Re = 100 at T = 5s

4 Overview of results

4.1 Validation of implementations

To test if three CFD solvers work or not, we chose to set up a fluid scenario: the lid-driven cavity. The lid-driven cavity problem is a simple two-dimensional case in which fluid is contained in a square domain with Dirichlet boundary conditions on all sides and three stationary walls. This case is suitable for testing for several reasons. First, as mentioned above, there is a great deal of research literature to compare with. Second, the laminar solution is steady. Third and most important, this case is easy to build and solve, which eases my workload for this project. Note that this is not necessarily the case for finite element methods, in which difficulties may arise at the corner intersections of the moving wall and the stationary wall. Figure 2 shows the results of u velocity fields. Notice that the results are sampled at t = 5s at step size $dt = 0.001$ with Reynolds number $Re = 100$ (Reynolds the number defines the degree of fluids turbulence) under $32 * 32$ grids for $L = 1$ spatial domain. Our results roughly are the similar with the results from the Fluent software. Unfortunately, we do not have the exact time domain of the Fluent setting. so that we cannot compare two results from grid to grid. The good thing is, as we mentioned previously, there are lots of literature we can somehow use to check the validity of our results. With the help from the previous research[1], we can match out results in a reasonably good agreement. Figure 7 shows the u velocities varied in the center line of domain between our results and Ghia (1982) results, and our results match with Ghia's results, which means that our solver is accurate and valid. Figure 3 shown below is the comparison between the 3 results of u velocities from the center of the vortex with the results post in Ghia's work[1]. This plot shows that sequential, OpenMP, and CUDA implementations work correctly.

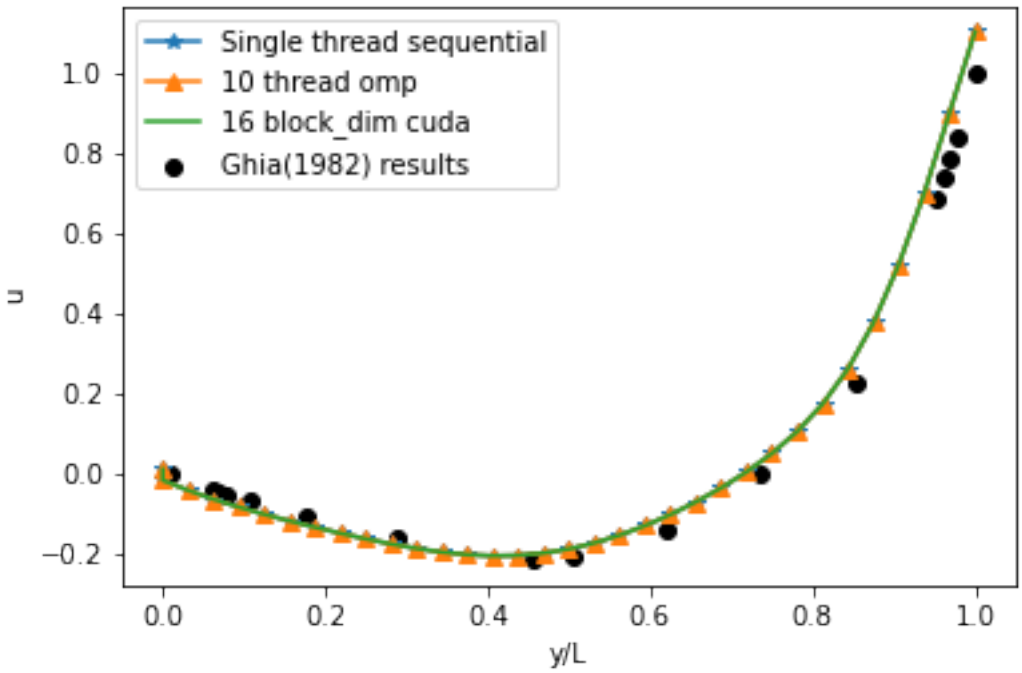


Figure 3: results of sequential, OpenMP, CUDA compare with the results from the other literature.

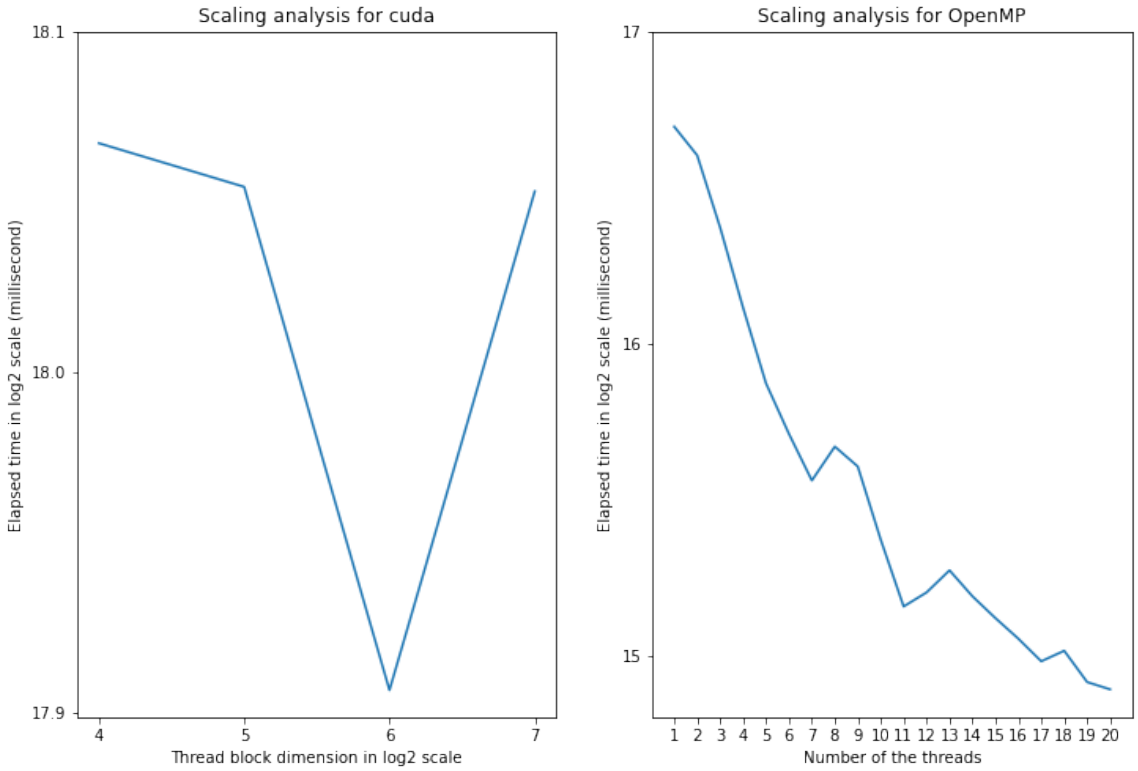


Figure 4: results of sequential, OpenMP, CUDA compare with the results from the other literature.

4.2 Comparison between different implementations

Although parallel computing utilizes threads so that operations can be performed simultaneously, in reality the computation might not be accelerated due to the overhead of the thread's communication, data racing, etc. Therefore, it is necessary to analyze the behavior of parallelism by conducting a scale analysis. Figure 4 shows the different elapsed durations under different configurations of threads for CUDA and OpenMP. Notice that the thread dimension for the CUDA configuration represents the size of the thread block. For example, $threadblock = 16$ represents the tile block was initiated as a $16 * 16 = 256$ CUDA threads in a cuda Streaming Multiprocessing. For the CUDA, the performance is optimal when the thread block is equal to 64; in other words, if 4096 threads were launched at the same time, the performance would be optimal. The CUDA we use is the RTX2080, and the maximum number of threads that can fly for an SM is 4086. Therefore, 100% occupation for SM will be reached at this point. Otherwise, the performance drops if we keep increasing the block size. For OpenMP, the elapsed time decreases as we keep increasing the number of threads up to 20. We were running this part on the Euler cluster with 20 CPUs allocated for our task. Therefore, we achieve the highest performance when 20 CPUs are exhausted.

Based on the results of the scaling analysis in figure 4, we have conducted a sensitivity analysis among the non-parallel, OpenMP (20 threads), and CUDA (32 thread block) with different fluid fields (grids) with one iteration (running the simulation for 5s would take too much time). The results are presented in figure 5. When the size of the grid is not too large, the CUDA implementation is actually pretty slow. However, as the size of grids becomes larger and larger, CUDA gains superiority over OpenMP and sequential computing. There are two possible reasons leading to this. First of all, the LU decomposition might not work for grain-level parallelism when the grid size is small. According to our parallel implementations in 3.3, we only call the parallel block for the outermost loop (equal to the size of the grid) when we conduct three-level loops to factorize the matrix. When the size is too small, there might be a few "lazy" CUDA threads that are not flying in the computations. These "lazy" threads fly only when the grid size is large. In other words, the outermost part of the loop is larger than the number of threads, such that only threads will participate in the loop. Another possible reason is that the cost of memory transactions between CPU memory and GPU memory is not economical when the grid size is too small. To conclude, we believe that CUDA parallelism should achieve better performance than other implementations since we usually have a really dense and large grid size (usually over one million) for an occasional CFD problem.

5 Deliverables

The whole project was written in cpp/cu files and built with the assistance of CMAKE building system. The program was running the Ubuntu OS 20.04, GCC 9.4, and NVCC 10.1. The source codes were saved under "./src" folder, while the corresponding header files

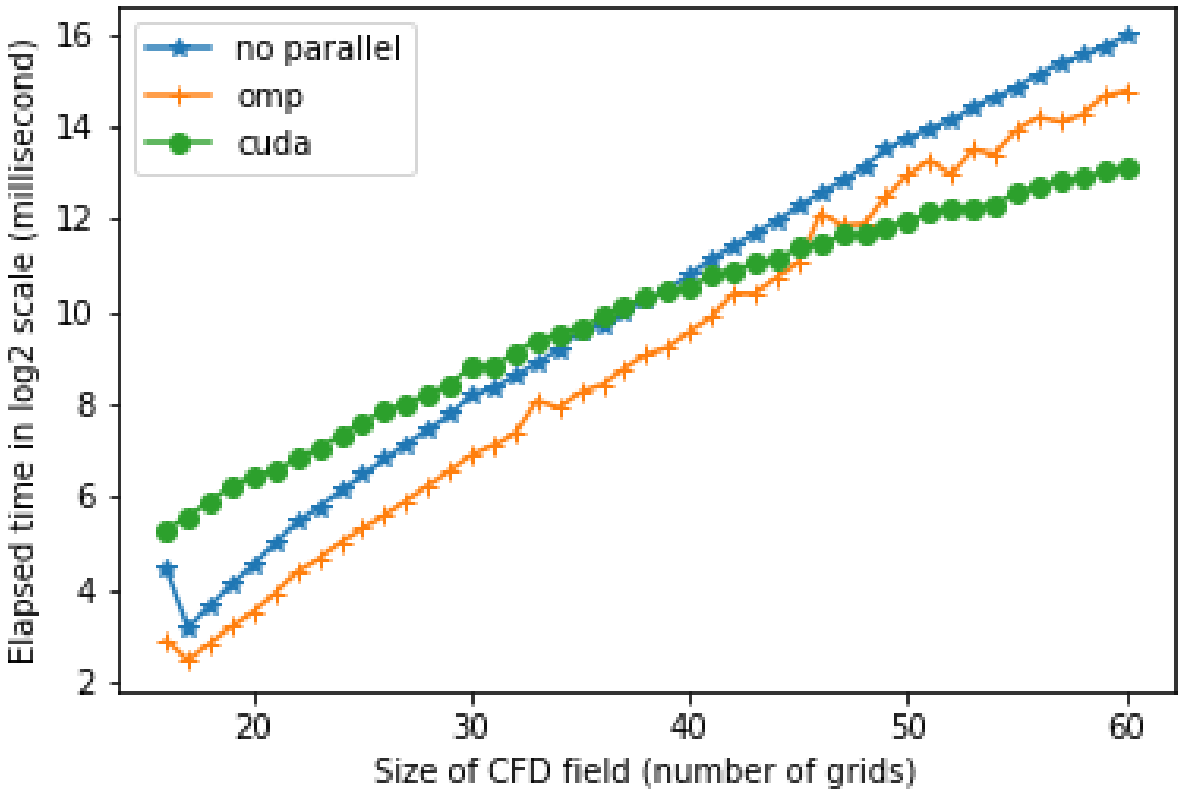


Figure 5: elapsed time under the different sizes of grids.

were saved under "./include" folder. A "CMakeLists.txt" was created under the root folder for our building system files. Generally, the programs can be divided into four submodules.

- parallel implementation including
  cfd.cpp, cfd.hpp: module for creating the mesh fields, calling the Fractional Step Algorithm, output the results
  splinear.cpp, splinear.hpp: module for solving the sparse linear system, called by the cfd.cpp as a subroute.
  A dynamic/shared library will be created in the end: libcfd.so

- CUDA implementation including
  cfd_cuda.cu, cfd_cuda.cuh: CUDA implementations for cfd solver
  splinear_cuda.cu, splinear_cuda.cuh: module for solving the sparse linear system using CUDA parallelism. Called by the cfd_cuda.cu as a subroutine.
  A dynamic/shared library will be created in the end: libcfd_cuda.so

- OpenMP implementation including
  cfd_omp.cpp, cfd_omp.hpp: module for creating the mesh fields, calling the Fractional Step Algorithm, output the results
  splinear_omp.cpp, splinear_omp.hpp: module for solving the sparse linear system using OpenMP parallelism. Called by the cfd_omp.cpp as a subroutine.
  A dynamic/shared library will be created in the end: libcfd_omp.so

- cfd controllers and setup including
  main.cpp: main entrance of program, configuration for grid and threads
  field.cpp, field.hpp: create a class for defining mesh and matrix.

To compile and run the code, we choose to use the CMAKE build system as it is easy to link multiple subroutines. Here is the procedure of building the program.You can find further details in our README file in the project repo.(https://git.doit.wisc.edu/BLU38/repo759).

```
cd FinalProject759
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

The results and plotting codes were saved under the folder "./results". You can find more details on how to generate the results by going through the instruction in the GitLab repository.

6 Conclusions and Future Work

In this project, we implemented three different kinds of CFD solvers (two parallel and one sequential) under the framework of the Fractional Step Method. The outputs of three implementations were compared with commercial software and previous research, which shows all three implementations are correct. The performances were evaluated and compared between each implementation and it shows that OpenMP runs 40% to 120% faster than sequential computing with 20 threads allocated no matter the sizes of grids. As for CUDA implementations, it achieves a better performance of 2-3 times faster under the 64*64 thread block when the grid size is larger than 64.

The three CFD solvers calculate the pressure-poisson matrix by solving a huge linear system defined by the boundary condition and the finite difference scheme. We choose to apply the LU decomposition as the linear solver algorithm. Meanwhile, we also implemented another algorithm, conjugate gradient descent, for solving linear systems. In this report, conjugate gradient descent is not fully analyzed and discussed as we found out that it is hard to converge and compute the correct answer, mainly due to the asymmetric linear system matrix of the CFD condition. However, we still regard it as a good trial for implementing various linear solvers. We may consider better approaches for CFD solvers in the future, such as conjugate gradient squared or successive-over-relaxation under parallel computing.

7 References

[1] U GHIA, KN GHIA, and CT SHIN. HIGH-RE SOLUTIONS FOR INCOMPRESSIBLE-FLOW USING THE NAVIER STOKES EQUATIONS AND A MULTIGRID METHOD. *JOURNAL OF COMPUTATIONAL PHYSICS*, 48(3):387–411, 1982.