IMMERSED BOUNDARY METHOD FOR SHARED-MEMORY ARCHITECTURES

JEFFREY K. WIENS.

ABSTRACT. In this report, we propose a novel, massively parallelizable algorithm for the immersed boundary method based on the fluid solver of Guermond and Minev [11]. This solver employs a directional-splitting technique that allows the incompressible Navier–Stokes equation to be efficiently parallelized on both shared and distributed memory architectures. An implementation of the numerical scheme was constructed in *Python* and *C/C++*, where the bottlenecks were parallelized using *OpenMP*. The parallelization techniques used in our implementation are discussed. Furthermore, results for a fluid-structure interaction problem are shown along with a corresponding performance study.

1. INTRODUCTION

The immersed boundary method is a mathematical framework for studying fluid-structure interaction problems created by Charles Peskin to simulate the motion of a heart valve [17]. Since the development of this framework, the immersed boundary method has found a wide variety of applications in biofluid dynamics including: blood flow in heart valves [8, 17, 19], aerodynamics of the vocal cords [5], sperm mobility [3, 4], flight of insects [14], and the motion of jellyfish [2].

Because of the complexity of these problems, numerical methods are essential when approximating the underlying governing equations. This requires designing algorithms [9, 13, 15, 17, 20] that overcome the inherent difficulties in solving a tightly coupled, nonlinear system. Furthermore, for many applications the problem size is far beyond the computational capabilities of a single processor. Therefore, in many instances, parallel algorithms are required. This led to the development of projects such as IBAMR [7] and Titanium [6] that run on distributed-memory architectures. Despite the success of these projects, the design of parallel immersed boundary methods still in its infancy. Currently, the algorithms employed are based around fluid solvers developed originally to run serially. Therefore, by designing an immersed boundary method around a massively parallelizable fluid solver, we expect a substantial improvement in the method's scalability.

In this report, we propose a new class of parallel immersed boundary methods that attempts to minimize communication between processes/threads. The algorithm uses Guermond and Minev's approximate projection method [11] that employs a novel direction-splitting technique. Because the fluid solver is weakly scalable, the proposed immersed boundary method is expected to be efficient for both distributed and shared memory architectures.

Date: April 23, 2012.

The implementation of our numerical scheme is written in *Python* and *C/C++*. Because of time constraints and the algorithm's complexity, we parallelized only the main bottlenecks using *OpenMP*.

To test our implementation, we simulate an oscillating elliptical membrane. The approximate solution to this problem is consistent with prior computations by Griffith[10]. Furthermore, we observe a speedup from our parallelism. However, because only a portion of the algorithm is parallelized, the efficiency of the algorithm could still be improved dramatically.

2. Mathematical Formulation

We begin by establishing the mathematical formulation of the problem. In the immersed boundary method, the fluid flow and immersed fiber evolution are described using two different grids. The fluid flow is described on a simple Eulerian grid whereas the immersed fiber is described on a more flexible Lagrangian grid.

For simplicity, we consider the two-dimensional domain

$$\Omega = [0,1] \times [0,1]$$

with periodic boundary conditions containing a single closed immersed fiber Γ. The domain Ω is parameterized using a Eulerian grid $\boldsymbol{x} = (x, y)$. Similarly, the fiber Γ is parameterized by $s \in [0, 1)$ on a Lagrangian grid whose position is defined by the curve

$$\boldsymbol{X}(s,t) = (X_1(s,t), X_2(s,t)).$$

The equations of motion are described by the system

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u}\right) = \mu \Delta \boldsymbol{u} - \nabla p + \boldsymbol{f}, \tag{1}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{2}$$

$$\frac{\partial \boldsymbol{X}(s,t)}{\partial t} = \boldsymbol{u}(\boldsymbol{X}(s,t),t) = \int_{\Omega} \boldsymbol{u}(\boldsymbol{x},t)\delta(\boldsymbol{x} - \boldsymbol{X}(s,t))\, dx, \tag{3}$$

$$\boldsymbol{f}(\boldsymbol{x},t) = \int_{\Gamma} \boldsymbol{F}(s,t)\delta(\boldsymbol{x} - \boldsymbol{X}(s,t))\, ds, \tag{4}$$

$$\boldsymbol{F}(s,t) = \sigma \frac{\partial^2 \boldsymbol{X}(s,t)}{\partial s^2}, \tag{5}$$

where

- $\boldsymbol{u}(\boldsymbol{x},t) = (u(\boldsymbol{x},t), v(\boldsymbol{x},t))$ is the fluid velocity and $p(\boldsymbol{x},t)$ is the pressure at the point $\boldsymbol{x}$ at time t,
- ρ is the constant fluid density, μ is the constant dynamic viscosity, and
- $\delta(\boldsymbol{x}) = \delta(x)\delta(y)$ is the 2D Dirac-delta function.

The fluid flow is described by the incompressible Navier–Stokes equations (1) and (2). Because the external force $\boldsymbol{f}(\boldsymbol{x},t)$ is coupled to the immersed fiber, the fiber is allowed to exert a force onto the fluid through the integral (4). Note, because the fluid and fiber are described on two different grids, a Dirac-delta function $\delta(\boldsymbol{x})$ is used to interpolate between them. Furthermore, we assume that the immersed fiber points are linked together by linear springs with spring constant σ and resting length $L = 0$. This gives us the force density relation (5). Lastly, we impose a no-slip and penetration condition at the immersed boundary interface. Therefore, the immersed boundary travels at the local fluid velocity given by (3).

3. Immersed Boundary Algorithm

In order to solve the immersed boundary equations, we construct two different grids. The immersed fiber, parameterized by $s \in [0,1)$, is discretized using N_b points that are equally spaced with width $h_b = 1/N_b$. As a short-hand, we define Lagrangian variables at time t_n by

$$\boldsymbol{X}_m^n = (X_1((m-1)h_b, t_n),\ X_2((m-1)h_b, t_n))$$

where $m = 1, \ldots, N_b$.

The domain Ω is divided into $N \times N$ uniform cells (Cartesian grid) on the periodic square $[0,1) \times [0,1)$ where each cell has a width $h = 1/N$. The pressure is specified at cell centers

$$p_{i,j}^n = p(\boldsymbol{x}_{i,j}, t_n)$$

where $i, j = 1, \ldots, N$ and

$$\boldsymbol{x}_{i,j} = \left((i - \tfrac{1}{2})h,\ (j - \tfrac{1}{2})h\right).$$

The fluid velocity uses a marker-and-cell (MAC) [12] discretization, illustrated in Figure 1, where the x-component of the velocity is defined on the east and west edges and the y-component on the north and south edges of each cell. More precisely, we denote the velocity field as

$$\boldsymbol{u}_{i,j}^{\mathrm{MAC},n} = \left(u_{i,j}^{\mathrm{MAC},n},\ v_{i,j}^{\mathrm{MAC},n}\right)$$

$$u_{i,j}^{\mathrm{MAC},n} = u((i-1)h, (j-\tfrac{1}{2})h, t_n)$$

$$v_{i,j}^{\mathrm{MAC},n} = v((i-\tfrac{1}{2})h, (j-1)h, t_n).$$

Lastly, it is important to note that $\boldsymbol{X}_m^n$ does not in general conform to the Eulerian grid.

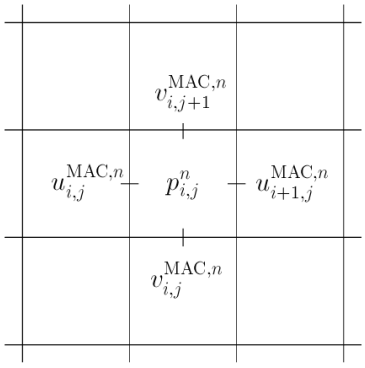


FIGURE 1. Location of fluid variables on the staggered grid.

3.1. **Discrete Delta Function.** In order to evaluate the integrals in (3) and (4), we are required to discretize a Dirac-delta function. As outlined in [18], the immersed boundary method requires the discrete delta function to satisfy certain criteria. Without justification, we use the discrete delta function

$$\delta_h(\boldsymbol{x}) = \frac{1}{h^2}\phi\left(\frac{x}{h}\right)\phi\left(\frac{y}{h}\right)$$

where

$$\phi(r) = \begin{cases} \frac{1}{8}(3 - 2|r| + \sqrt{1 + 4|r| - 4r^2}), & 0 \le |r| < 1 \\ \frac{1}{8}(5 - 2|r| - \sqrt{-7 + 12|r| - 4r^2}), & 1 \le |r| < 2 \\ 0, & 2 \le |r| \end{cases}.$$

The above delta function is a popular choice in the immersed boundary community [9, 15, 13]. However, the algorithm described is not dependent on this particular choice.

3.2. **Difference Operators.** Before discussing the algorithm, we require several differential operators. We denote $\boldsymbol{D}_{xx}$ and $\boldsymbol{D}_{yy}$ as the standard second-order difference stencils

$$\boldsymbol{D}_{xx}p_{i,j} = \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{h^2}$$

and

$$\boldsymbol{D}_{yy}p_{i,j} = \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{h^2},$$

which approximates the second derivative. These operators can only be applied to variables defined at cell centers or cell edges on the Eulerian grid. Similarly, we denote the difference stencil

$$\boldsymbol{D}_{ss}F_m = \frac{F_{m+1} - 2F_m + F_{m-1}}{h_b^2},$$

which can be applied to variables on the Lagrangian grid. Condensing the notation further, we define the vector operation

$$\boldsymbol{D}_{xx}\boldsymbol{u}_{i.j}^{\text{MAC}} = \begin{bmatrix} \boldsymbol{D}_{xx}u_{i.j}^{\text{MAC}} \\ \\ \boldsymbol{D}_{xx}v_{i.j}^{\text{MAC}} \end{bmatrix}.$$

Lastly, we approximate the divergence and gradient operations. Because fluid variables are defined at different locations (cell centers and edges), we require operators whose output and input are at different locations. Therefore, we define the gradient approximation as

$$\boldsymbol{G}^{C\to\text{MAC}}p_{i,j} = \left(\frac{p_{i,j} - p_{i-1,j}}{h}, \frac{p_{i,j} - p_{i,j-1}}{h}\right),$$

which takes a cell-center variable as input and returns a MAC variable. Likewise, we define the divergence approximation as

$$\boldsymbol{D}^{\text{MAC}\to C} \cdot \boldsymbol{u}_{i,j}^{\text{MAC}} = \frac{u_{i+1,j} - u_{i,j}}{h} + \frac{v_{i,j+1} - v_{i,j}}{h},$$

which takes a MAC variable as input and returns a cell-center variable.

3.3. **Algorithm.** Next, we describe the algorithm used in this project. The novelty of the algorithm comes from using Guermond and Minev [11] fluid solver. This allows the momentum solve and project steps to be split directionally. This is advantageous when parallelizing the method.

The remainder of the algorithm uses standard techniques. The integrals in the immersed boundary equations are solved using a first-order quadrature and the fiber evolution equation is solved using a second-order Adams-Bashforth method. Lastly, the algorithm treats the nonlinear advection term explicitly, denoted as $\text{N}(\boldsymbol{u}_{i,j}^{\text{MAC},n})$, which is discussed in the next section.

Now, suppose the state variables are known at the nth time-step, the algorithm is then defined by the following procedure.

(1) Evolve the immersed boundary position to the $n + 1/2$ time-step:

1: $\boldsymbol{U}_m^n = \sum_{i,j} \boldsymbol{u}_{i,j}^{\text{MAC},n}\delta_h(\boldsymbol{x}_{i,j}^{\text{MAC}} - \boldsymbol{X}_m)h^2$ ▷ Velocity of immersed boundary (Lagrangian grid).

2: $\dfrac{\boldsymbol{X}_m^{n+1} - \boldsymbol{X}_m^n}{\Delta t} = \dfrac{3}{2}\boldsymbol{U}_m^n - \dfrac{1}{2}\boldsymbol{U}_m^{n-1}$ ▷ Evolve immersed boundary, eqn. (3).

3: $\boldsymbol{X}_m^{n+1/2} = \frac{1}{2}\left(\boldsymbol{X}_m^{n+1} + \boldsymbol{X}_m^{n}\right)$ ▷ Average Position.

(2) Spread the force density onto the Eulerian grid:

1: $\boldsymbol{F}_m^{n+1/2} = \sigma D_{ss} \boldsymbol{X}_m^{n+1/2}$ ▷ Calculate Force density.

2: $\boldsymbol{f}_{i,j}^{\mathrm{MAC},n+1/2} = \sum_m \boldsymbol{F}_m^{n+1/2}\, \delta_h(\boldsymbol{x}_{i,j}^{\mathrm{MAC}} - \boldsymbol{X}_m^{n+1/2}) h_b$ ▷ Distribute Force onto Eulerian grid.

(3) Solve incompressible Navier–Stokes equation:

1: $p_{i,j}^{*,n+1/2} = p_{i,j}^{n-1/2} + \psi_{i,j}^{n-1/2}$ ▷ Predict Pressure.

2: ▷ Momentum Solve - ADI in x-direction.

$$\rho\left(\frac{\boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1/2} - \boldsymbol{u}_{i,j}^{\mathrm{MAC},n}}{\Delta t/2} + \mathrm{N}(\boldsymbol{u}^{\mathrm{MAC},n})\right) = \mu\left(\boldsymbol{D}_{xx}\boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1/2} + \boldsymbol{D}_{yy}\boldsymbol{u}_{i,j}^{\mathrm{MAC},n}\right) - \boldsymbol{G}^{C\to\mathrm{MAC}} p_{i,j}^{*,n+1/2} + \boldsymbol{f}_{i,j}^{\mathrm{MAC},n+1/2}$$

3: ▷ Momentum Solve - ADI in y-direction.

$$\rho\left(\frac{\boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1} - \boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1/2}}{\Delta t/2} + \mathrm{N}(\boldsymbol{u}^{\mathrm{MAC},n+1/2})\right) = \mu\left(\boldsymbol{D}_{xx}\boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1/2} + \boldsymbol{D}_{yy}\boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1}\right) - \boldsymbol{G}^{C\to\mathrm{MAC}} p_{i,j}^{*,n+1/2} + \boldsymbol{f}_{i,j}^{\mathrm{MAC},n+1/2}$$

4: $(1 - \boldsymbol{D}_{xx})(1 - \boldsymbol{D}_{yy})\,\psi_{i,j}^{n+1/2} = \frac{-1}{\Delta t}\boldsymbol{D}^{\mathrm{MAC}\to C} \cdot \boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1}$ ▷ Penalty Step.

5: $p_{i,j}^{n+1/2} = p_{i,j}^{n-1/2} + \psi_{i,j}^{n+1/2} - \frac{\mu}{\rho}\boldsymbol{D}^{\mathrm{MAC}\to C} \cdot \left(\frac{1}{2}(\boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1} + \boldsymbol{u}_{i,j}^{\mathrm{MAC},n})\right)$ ▷ Predict Pressure.

3.4. **Nonlinear Advection.** For this project, we approximate the non-linear advection term

$$\boldsymbol{u}(\boldsymbol{x}_{i,j}^{\mathrm{MAC}}, t_n) \cdot \nabla \boldsymbol{u}(\boldsymbol{x}_{i,j}^{\mathrm{MAC}}, t_n) \approx \mathrm{N}(\boldsymbol{u}_{i,j}^{\mathrm{MAC},n})$$

in the same manner as the unpublished MIT report [21]. This approach combines an upwinding and center difference scheme that estimates the advection term $\mathrm{N}(\boldsymbol{u})$ at the current time-step. Because the flow is potentially driven by either the diffusion or advection term, the two difference schemes are combined in such a manner to give a suitable discretization. Note this is not a WENO-like spatial discretization because the difference stencil does not vary spatially. Instead, the operator is applied uniformly to the discrete velocity field at each time-step.

Ideally, the non-linear advection term should be approximated at the $n + \frac{1}{2}$ time-step using a more sophisticated technique. However, in order to reduce the algorithm's complexity, we want to keep the approximation as simple as possible.

Before describing the discretization, we need to define several operators.

- **Interpolation Operators (Average):** The operator

$$\left(\bar{u}^c_{i,j},\ \bar{v}^c_{i,j}\right) = \left(\frac{u^{\mathrm{MAC}}_{i,j} + u^{\mathrm{MAC}}_{i+1,j}}{2},\ \frac{v^{\mathrm{MAC}}_{i,j} + v^{\mathrm{MAC}}_{i,j+1}}{2}\right)$$

  interpolates a MAC velocity field to cell centers. Similarly, the operator

$$\left(\bar{u}^v_{i,j},\ \bar{v}^v_{i,j}\right) = \left(\frac{u^{\mathrm{MAC}}_{i,j} + u^{\mathrm{MAC}}_{i,j-1}}{2},\ \frac{v^{\mathrm{MAC}}_{i,j} + v^{\mathrm{MAC}}_{i-1,j}}{2}\right)$$

  interpolates a MAC velocity field to cell corners (vertices). Note that $\bar{u}^v_{i,j}$ is located on the bottom left corner of a cell.

- **Jump Operators (Differences):** The operator

$$\left(\tilde{u}^c_{i,j},\ \tilde{v}^c_{i,j}\right) = \left(\frac{u^{\mathrm{MAC}}_{i+1,j} - u^{\mathrm{MAC}}_{i,j}}{2},\ \frac{v^{\mathrm{MAC}}_{i,j+1} - v^{\mathrm{MAC}}_{i,j}}{2}\right)$$

  and

$$\left(\tilde{u}^v_{i,j},\ \tilde{v}^v_{i,j}\right) = \left(\frac{u^{\mathrm{MAC}}_{i,j} - u^{\mathrm{MAC}}_{i,j-1}}{2},\ \frac{v^{\mathrm{MAC}}_{i,j} - v^{\mathrm{MAC}}_{i-1,j}}{2}\right)$$

  are required difference quantities.

- **Gradient Operators:** The operator

$$\begin{aligned}\boldsymbol{G}^{C\to\mathrm{MAC}} p_{i,j} &= \left(\frac{p_{i,j} - p_{i-1,j}}{h},\ \frac{p_{i,j} - p_{i,j-1}}{h}\right) \\ &= \left(\boldsymbol{G}^{C\to\mathrm{MAC}}_{(1)} p_{i,j},\ \boldsymbol{G}^{C\to\mathrm{MAC}}_{(2)} p_{i,j}\right)\end{aligned}$$

calculates a MAC gradient from a cell center variable. Similarly, we define

$$\begin{aligned}\bar{\boldsymbol{G}}^{V\to\mathrm{MAC}}u^v_{i,j} &= \left(\frac{u^v_{i,j+1}-u^v_{i,j}}{h},\ \frac{u^v_{i+1,j}-u^v_{i,j}}{h}\right)\\ &= \left(\bar{\boldsymbol{G}}^{V\to\mathrm{MAC}}_{(1)}u^v_{i,j},\ \bar{\boldsymbol{G}}^{V\to\mathrm{MAC}}_{(2)}u^v_{i,j}\right)\end{aligned}$$

that approximates the MAC field

$$\bar{\boldsymbol{G}}^{V\to\mathrm{MAC}}u^V_{i,j} \approx (\partial_y u_{i,j},\ \partial_x u_{i,j})^{\mathrm{MAC}}$$

from vertex data.

Using these operators, we approximate the non-linear advection term, described in the MIT report [21], as

$$\mathrm{N}(\boldsymbol{u}^{\mathrm{MAC},n}_{i,j}) = \begin{bmatrix} \mathrm{N}(\boldsymbol{u}^{\mathrm{MAC},n}_{i,j})_{(1)} \\ \\ \mathrm{N}(\boldsymbol{u}^{\mathrm{MAC},n}_{i,j})_{(2)} \end{bmatrix},$$

$$\mathrm{N}(\boldsymbol{u})_{(1)} = \boldsymbol{G}^{C\to\mathrm{MAC}}_{(1)}\left((\bar{u}^C)^2-\gamma|\bar{u}^C|\tilde{u}^C\right)+\bar{\boldsymbol{G}}^{V\to\mathrm{MAC}}_{(1)}\left((\bar{u}^V\bar{v}^V)-\gamma|\bar{v}^V|\tilde{u}^V\right),$$

$$\mathrm{N}(\boldsymbol{u})_{(2)} = \bar{\boldsymbol{G}}^{V\to\mathrm{MAC}}_{(2)}\left((\bar{u}^V\bar{v}^V)-\gamma|\bar{u}^V|\tilde{v}^V\right)+\boldsymbol{G}^{C\to\mathrm{MAC}}_{(2)}\left((\bar{v}^C)^2-\gamma|\bar{v}^C|\tilde{v}^C\right),$$

$$\gamma = \min\left(1.2\Delta t\max_{\forall i,j}\left(|u^{\mathrm{MAC}}_{i,j}|\right),\ 1.2\Delta t\max_{\forall i,j}\left(|v^{\mathrm{MAC}}_{i,j}|\right),\ 1\right).$$

4. Benchmark Problem

To test the correctness and performance of our algorithm, we consider an oscillating elliptical membrane. We supplement the system of equations described in Section 2 with the initial conditions

$$\boldsymbol{X}(s,t) = \left(\frac{1}{2}+r_{\min}\cos(2\pi s),\ \frac{1}{2}+r_{\max}\sin(2\pi s)\right)$$

and

$$\boldsymbol{u}(\boldsymbol{x},t) = (0,\ 0)$$

corresponding to an elliptical membrane in a stationary fluid. For completeness, we set the following parameters:

$$\mu = 1.0,\ \rho = 1.0,\ \sigma = 10,000,\ r_{\min} = 0.2,\ \text{and } r_{\max} = 0.4.$$

When discretizing the system of equations, outlined in Section 3, we set

$$N = 63,\ N_b = 189,\ \text{and } \Delta t = 10^{-5}.$$

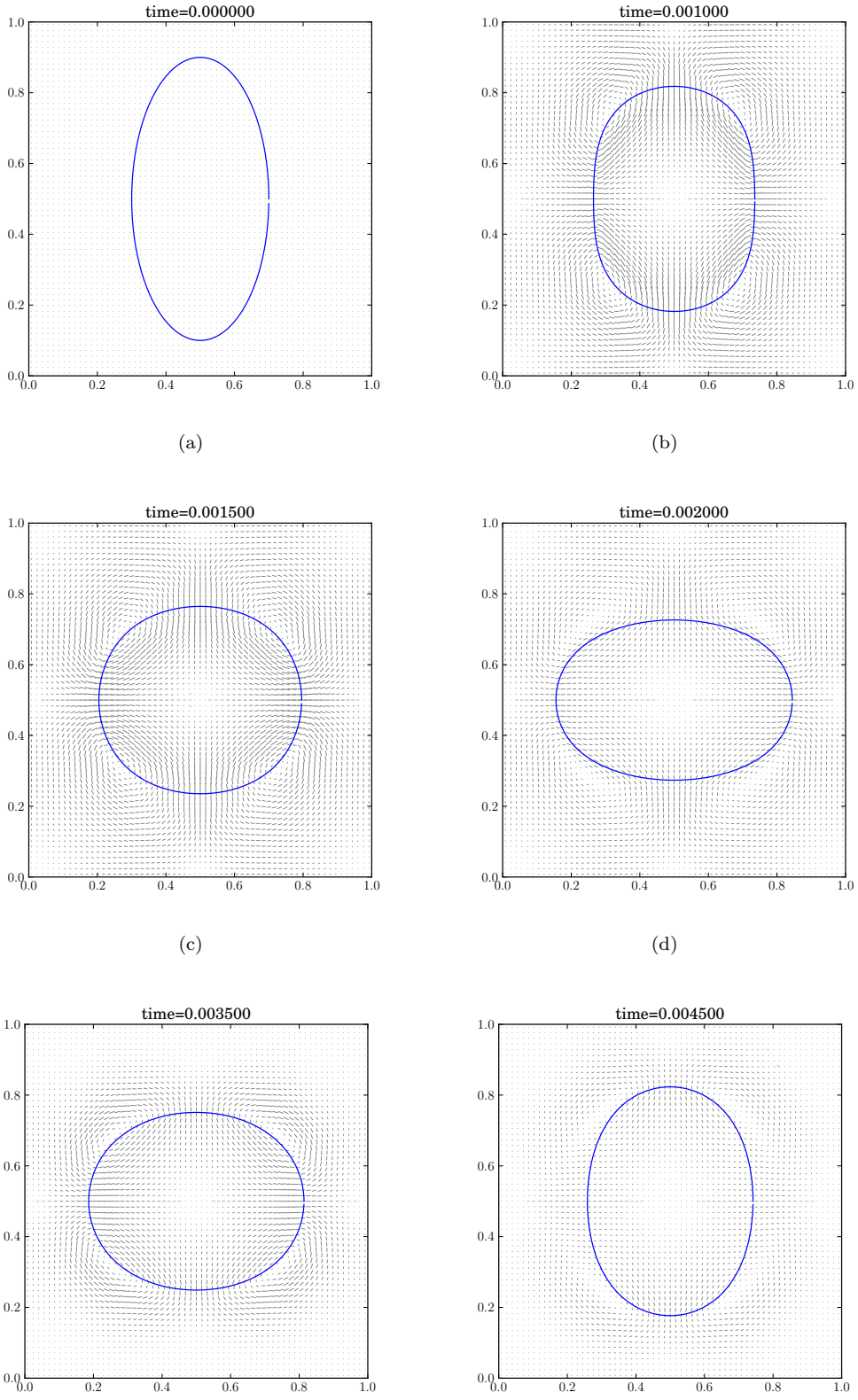


FIGURE 2. Evolution of elliptical membrane problem described in Section 4.

Note, because the spring constant σ is stiff, the dynamics occur over a small time-scale ($0.04 \geq t$) and require a small time-step to resolve. The evolution of the system is shown in Figure 2 and is consistent with that of Griffith [10]. Because the membrane is closed and the fluid is incompressible, we know that the volume inside the oscillating ellipse remains constant. Therefore, the ellipse oscillates until it reaches a steady state, a circle with radius $r = \sqrt{r_{\min} r_{\max}}$. By plotting the maximum and minimum radius of the ellipse in time, shown in Figure 3, we verify that the approximate solution converges to the correct steady state.

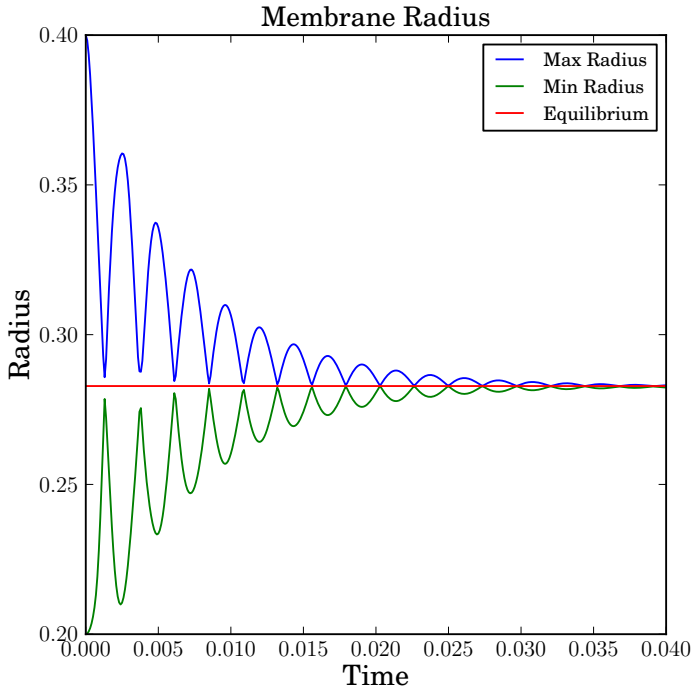


FIGURE 3. Radius of ellipse for the elliptical membrane problem described in Section 4 with $N = 127$, $N_b = 381$, and $\Delta t = 10^{-6}$.

Lastly, we perform a convergence study for this problem in the discrete L2 norm

$$||q_{i,j}||_2 = \Delta x \left(\sum_{i,j} |q_{i,j}|^2 \right)^{1/2}.$$

Because the exact solution to this problem is not known, we estimate the convergence rate by comparing approximate solutions with different mesh sizes. However, because the approximate solution is known only at discrete points, the grid points on the coarser mesh must coincide that of the finer mesh. This is achieved

by letting $\Delta x_{\text{coarse}}/\Delta x_{\text{fine}} = 3$[1]. Therefore, the error between two approximate solutions is estimated by

$$e_2[q;N] = ||q_N - \mathcal{I}^{3N\to N} q_{3N}||_2$$

where $\mathcal{I}^{3N\to N}$ interpolates the finer mesh to the coarser mesh. This allows for an empirical estimate of the convergence rate of q given by

$$r_2[q;N] = \log_3\left(\frac{e_2[q;N]}{e_2[q;3N]}\right). \tag{6}$$

By assuming that the error has the form $e_2[q;N] = C(1/N)^r$, this empirical estimate is easily derived by noting that

$$\begin{aligned}
\log_3\left(\frac{e_2[q;N]}{e_2[q;3N]}\right) &= \log_3\left(\frac{C\left(\frac{1}{N}\right)^r - C\left(\frac{1}{3N}\right)^r}{C\left(\frac{1}{3N}\right)^r - C\left(\frac{1}{9N}\right)^r}\right), \\
&= \log_3\left(\frac{1-\left(\frac{1}{3}\right)^r}{\left(\frac{1}{3}\right)^r\left(1-\left(\frac{1}{3}\right)^r\right)}\right), \\
&= r.
\end{aligned}$$

Using the estimate (6), the convergence rate of the velocity field is shown in Figure 4 for the elliptical membrane problem. From this plot, the convergence rate of the velocity field is observed to be roughly 1.5. This is comparable to the convergence rate observed by Griffith for a similar problem [9].

5. Parallelization of Algorithm

Conceptually, parallelizing the algorithm presented in Section 3 is straightforward. The majority of the algorithm's steps are explicit calculations that are easily parallelized. The exception is parallelizing the linear system solver that is used when evolving the fluid variables in time.

Because of the algorithm's complexity, we focus on parallelizing the main bottlenecks in our serial code. The pre-parallelized version of the code is written primarily in *Python* with performance-critical sections in *Cython*. By profiling the serial code, we observe three major bottlenecks:

(1) **Linear Systems:** Solving the linear systems found in step 3, line 2-4, in Section 3.3.

(2) **Spreading Force:** Spreading the force density onto the Eulerian grid in step 2, line 2, in Section 3.3.

(3) **Interpolating Velocity:** Interpolating the velocity on the Eulerian grid onto the Lagrangian grid in step 1, line 1, in Section 3.3.

[1]This occurs because we are using a staggered grid. Therefore, we must divide each cell in our coarse grid into nine cells to obtain overlapping grid points. Note when using a standard finite difference scheme, we only have to cut the grid spacing by half to get overlapping grid points.

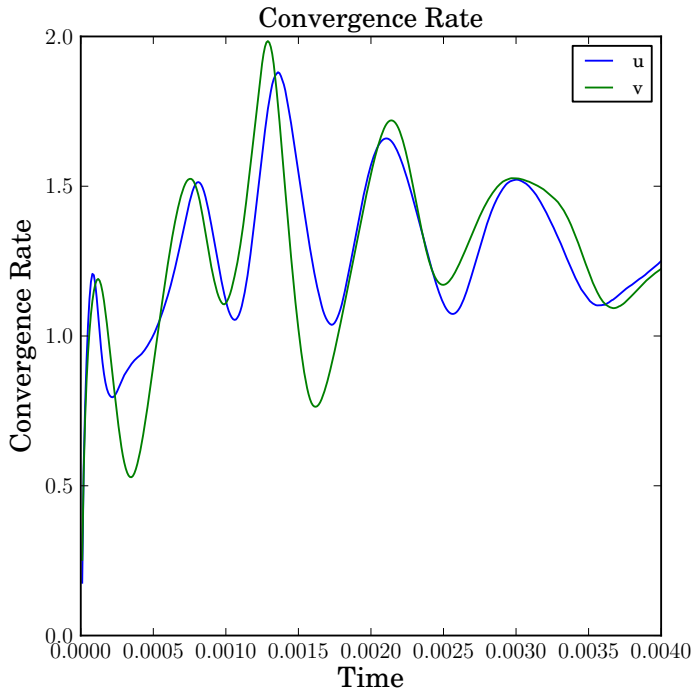


FIGURE 4. Convergence rate (6) in the L2 norm for the elliptical membrane problem described in Section 4 with $N = 64$, $N_b = 3N$, and $\Delta t = 10^{-6}$.

In the attempts to improve the overall performance, each of these bottlenecks are parallelized using *OpenMP*. Note, solving the linear systems is the most significant bottleneck in our serial code constituting roughly 25% of the total execution time.

5.1. **Parallelizing Linear System Solve.** When evolving the fluid variables in time, we are required to solve at minimum five linear systems. However, by splitting the linear systems directionally, these five linear systems of size $N^2 \times N^2$ are deflated into $6N$ linear systems of size $N \times N$. This improves the efficiency of the algorithm in two ways. First, this reduces the overall operation count required to evolve the fluid in time. Secondly, it provides an opportunity to parallelize by subdividing the $6N$ linear systems among different processing units.

To illustrate, let us consider line 2 in step 3 that solves the momentum equation. When solving for the x-component of the velocity, the system reduces to N linear systems expressed as

$$\boldsymbol{A}\vec{u}^{(j)} = \vec{b}^{(j)}$$

where

$$\boldsymbol{A} = \frac{2\rho}{\Delta t}\boldsymbol{I} - \mu \boldsymbol{D}_{xx},$$

$$[\vec{u}^{(j)}]_i = u_{i,j}^{\mathrm{MAC},n+1/2}\,, \text{ and}$$

$$[\vec{b}^{(j)}]_i = \frac{2\rho}{\Delta t}u_{i,j}^{\mathrm{MAC},n} - \rho \mathrm{N}_{(1)}(\boldsymbol{u}^{\mathrm{MAC},n}) + \mu \boldsymbol{D}_{yy} u_{i,j}^{\mathrm{MAC},n} - \boldsymbol{G}_{(1)}^{C\to\mathrm{MAC}} p_{i,j}^{*,n+1/2} + f_{(1),i,j}^{\mathrm{MAC},n+1/2}\,.$$

Because $\boldsymbol{A}$ is periodic and tridiagonal, the linear systems are solved efficiently using Thomas' algorithm [16] requiring only $O(N)$ operations per linear system. Splitting the remaining linear systems in the momentum equations (line 2 and 3, step 3) is straightforward using an analogous splitting strategy. Each of these linear systems result in a periodic, tridiagonal system and is solved in a similar fashion.

When solving the penalty step in step 3 of our algorithm (line 4), the linear system

$$(1 - \boldsymbol{D}_{xx})\,(1 - \boldsymbol{D}_{yy})\,\psi_{i,j}^{n+1/2} = \frac{-1}{\Delta t}\boldsymbol{D}^{\mathrm{MAC}\to C} \cdot \boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1}$$

is equivalent to

$$(1 - \boldsymbol{D}_{xx})\,\psi_{i,j}^{*,n+1/2} = \frac{-1}{\Delta t}\boldsymbol{D}^{\mathrm{MAC}\to C} \cdot \boldsymbol{u}_{i,j}^{\mathrm{MAC},n+1}$$

and

$$(1 - \boldsymbol{D}_{yy})\,\psi_{i,j}^{n+1/2} = \psi_{i,j}^{*,n+1/2}\,.$$

By splitting directionally, we reduce each system into N linear systems each having a $N \times N$ periodic, tridiagonal matrix.

By subdividing the linear systems among processing units, we obtain a strategy that is embarrassingly parallel. As shown in Table 1, the parallelized linear solver is strongly scalable and has an efficiency of one. Note, in this section of code, only the linear solves are parallelized, not the construction of these systems. Obviously, the performance of the entire algorithm would improve by parallelizing the construction of these systems. However, because of time constraints, we consider this improvement outside the scope of this project.

TABLE 1. Execution Time, Speedup, and Efficiency for solving $2N$ linear systems of size $N \times N$ in parallel with $N = 4000$.

Threads	Time (sec)	Speedup	Efficiency
8	0.1767	7.93	0.99
7	0.2015	6.96	0.99
6	0.2346	5.97	1.00
5	0.2804	5.00	1.00
4	0.3496	4.00	1.00
3	0.4687	2.99	1.00
2	0.7037	1.99	1.00
1	1.4019	-	-

5.2. **Parallelizing Force Spreading and Velocity Interpolation.** The force spreading and velocity interpolation steps are explicit calculations that are straightforward to parallelize. The force spreading corresponds to the summation found in line 2, step 2, in Section 3.3. Similarly, the velocity interpolation corresponds to the summation found in line 1, step 1, in Section 3.3.

The velocity interpolation calculation is summarized by the pseudo-code in Algorithm 1. Because the algorithm updates only $\boldsymbol{U}[m]$, we do not introduce a race condition when parallelizing the outer loop. Furthermore, because δ_h is sparse, we reduce the computation further by only looping over the indices that result in non-zero contributions. By subdividing the calculation of $\boldsymbol{U}[m]$ among processing units, we observe in Table 2 that this code segment is both strongly scalable and has an efficiency close to one.

Algorithm 1 Velocity Interpolation

```
1: for m = 1, ..., N_b do                                  ▷ Loop through Lagrangian grid.
2:     for i = 1, ..., N do                                ▷ Loop through Eulerian grid.
3:         for j = 1, ..., N do
4:             U[m] = U[m] + u_{i,j}^{MAC,n} δ_h(x_{i,j}^{MAC} − X_m) h^2
5:         end for
6:     end for
7: end for
```

TABLE 2. Execution Time, Speedup, and Efficiency when interpolating the velocity in parallel with $N = 511$ and $N_b = 1533$.

Threads	Time (sec)	Speedup	Efficiency
8	0.00113	7.896	0.987
7	0.001286	6.939	0.991
6	0.001505	5.929	0.988
5	0.001807	4.938	0.988
4	0.002250	3.966	0.991
3	0.003009	2.965	0.988
2	0.004486	1.989	0.995
1	0.008923	-	-

When parallelizing the force spreading operation, the strategy is more involved than the velocity interpolation. From the force spreading pseudo-code in Algorithm 2, we observe that parallelizing the outer loop would introduce a race condition. When using a block scheduler, these race conditions are rare because the support of the delta function is small[2]. Therefore, using an atomic operation (critical section) on line 5 in Algorithm 2, we ensure that no race conditions are introduced. Additionally, because an atomic operation is used instead of a critical section, the entire array $\boldsymbol{f}[i][j]$ is not locked when updating a single element. The algorithm only blocks when two or more threads attempt to update the same memory location. Because these occurrences are rare, we obtain a parallelization strategy that has a high efficiency (see Table 3).

[2]In our implementation, we do not loop over all i and j. We only loop over indexes that give non-zero contributions.

Algorithm 2 Force Spreading

for $m = 1, \ldots, N_b$ **do** ▷ Loop through Lagrangian grid.
  for $i = 1, \ldots, N$ **do** ▷ Loop through Eulerian grid.
    for $j = 1, \ldots, N$ **do**
      $v = \boldsymbol{F}_m^{n+1/2}\, \delta_h(\boldsymbol{x}_{i,j}^{\mathrm{MAC}} - \boldsymbol{X}_m^{n+1/2})h_b$
      $\boldsymbol{f}[i][j] = \boldsymbol{f}[i][j] + v$
    end for
  end for
end for

TABLE 3. Execution Time, Speedup, and Efficiency when spreading force in parallel with $N = 511$ and $N_b = 1533$.

Threads	Time (sec)	Speedup	Efficiency
8	0.001396	7.756	0.969
7	0.001606	6.742	0.963
6	0.001881	5.756	0.959
5	0.002250	4.812	0.962
4	0.002795	3.874	0.968
3	0.003765	2.876	0.959
2	0.005615	1.928	0.964
1	0.010827	-	-

5.3. **Global Timing.** Although there exist efficient strategies to parallelize the entire algorithm, a large portion of our immersed boundary code remains serial. This severely restricts the performance gained from using multiple cores. In Table 4, we measure the execution time per time-step and corresponding speedup for the oscillating ellipse problem. For this problem, the speedup is roughly 1.25 when using 3 threads[3]. Ideally, for perfectly parallelized algorithms, the speedup should be equivalent to the number of threads used. Although the current speedup is far from the ideal, these results could be dramatically improved by parallelizing the remainder of the algorithm.

The limited speedup we observe in our immersed boundary code is strongly connected to Amdahl's law [1]. Amdahl's law states that a program's speedup is bounded if the program is not completely parallelized. For example, if p percent of our immersed boundary code is perfectly parallelized, we expect the total execution time for r threads to be

$$T_r = pT_{\mathrm{serial}}/r + (1-p)T_{\mathrm{serial}}.$$

Therefore, the speedup would be

$$S_r = \frac{1}{p(1/r - 1) + 1}. \tag{7}$$

[3]Because *socrates* does not have the latest versions of Python, Numpy, and Scipy installed, our code does not run on this machine. Therefore, the performance study in this section was performed on the author's personal computer restricting the analysis to three cores. Note, the performance study in Section 5.1 and 5.2 were ran on *socrates*. For completeness, the author's personal computer uses Python 2.7.2, Numpy 1.5.1, Scipy 0.9.0, and Matplotlib 1.0.1

TABLE 4. Average Execution Time and Speedup per time-step for the oscillating ellipse problem (Section 4).

	Time (sec)			Speedup	
N ($N_b = 3N$)	1 Thread	2 Thread	3 Thread	2 Thread	3 Thread
63	0.003741	0.003110	0.002964	1.20	1.26
127	0.011183	0.009141	0.008709	1.22	1.28
255	0.041123	0.034101	0.032589	1.21	1.26
354	0.076610	0.064151	0.060663	1.19	1.26
454	0.128656	0.107846	0.101660	1.19	1.27
511	0.166249	0.141490	0.136243	1.17	1.22
600	0.232783	0.197535	0.187796	1.18	1.24
700	0.330627	0.283366	0.271736	1.17	1.22
800	0.435405	0.386508	0.361179	1.13	1.21
999	0.675969	0.583760	0.550076	1.16	1.23

Note that the speedup $S_r \to 1/(1-p)$ as $r \to \infty$. This means the speedup is bounded when $p \neq 1$. Furthermore, by solving for p in (7), we estimate the percentage of the program parallelized by

$$p = \frac{1/S_r - 1}{1/r - 1}.$$

Using the data in Table 4, we approximate the percentage of the program parallelized as $p \approx .3$. Therefore, we obtain a maximum speedup of 1.428 for our code.

6. Review of Project's Objectives

As identified in our interim progress report, the primary objectives for the course project were to:

(1) implement a serial version of the immersed boundary method using a novel immersed boundary algorithm,

(2) parallelize the three major bottlenecks in the serial code: Linear Solver, Force Spreading, and Velocity Interpolation, and

(3) perform a performance study on a benchmark problem.

Although all of these objectives were reached, the resulting implementation of the immersed boundary method did not perform as expected. This was caused by inadvertently improving the serial performance of our pre-parallelized code when parallelizing. In the original pre-parallelized code, the identified bottlenecks constituted a significant portion of the overall execution time. Therefore, by parallelizing these bottlenecks, we expected to observe a significant increase in performance. However, after parallelizing, these perceived bottlenecks were no longer significant when running the code serially. In fact, by profiling the parallelized code, no significant bottlenecks were observed at all. Therefore, to obtain any significant performance gain from parallelizing, the entire algorithm would have needed to be parallelized.

In Figure 5, we show the execution time per time-step for the pre-parallelized and parallelized code. When the parallelized code is run serially, we observe a substantial difference in execution time when compared with the pre-parallelized version. When concerned about minimizing the total execution time, this improvement in performance is remarkable. However, this performance improvement unfortunately overshadows and hinders the performance gains achieved by parallelizing. For this reason, it is worthwhile to explore the rationale for this improvement in more detail.

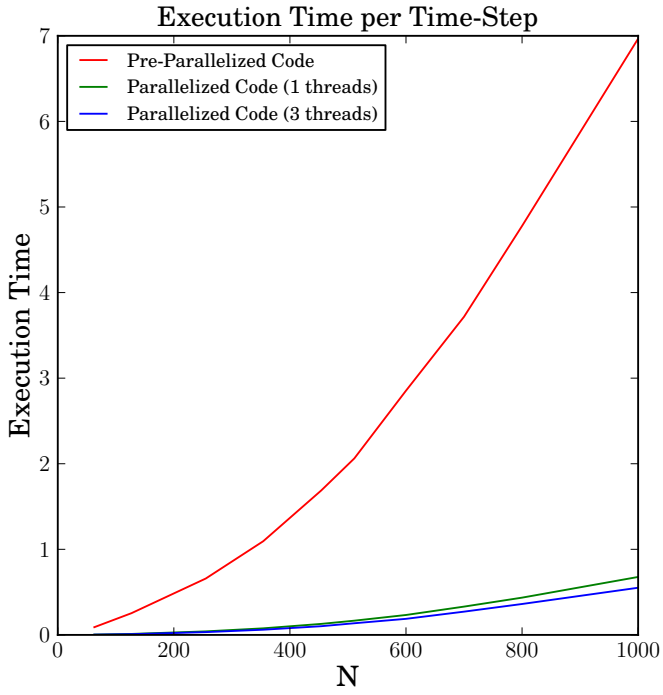


FIGURE 5. Execution time per time-step for the elliptical membrane problem described in Section 4.

The performance increase in Figure 5 is a result of two major factors:

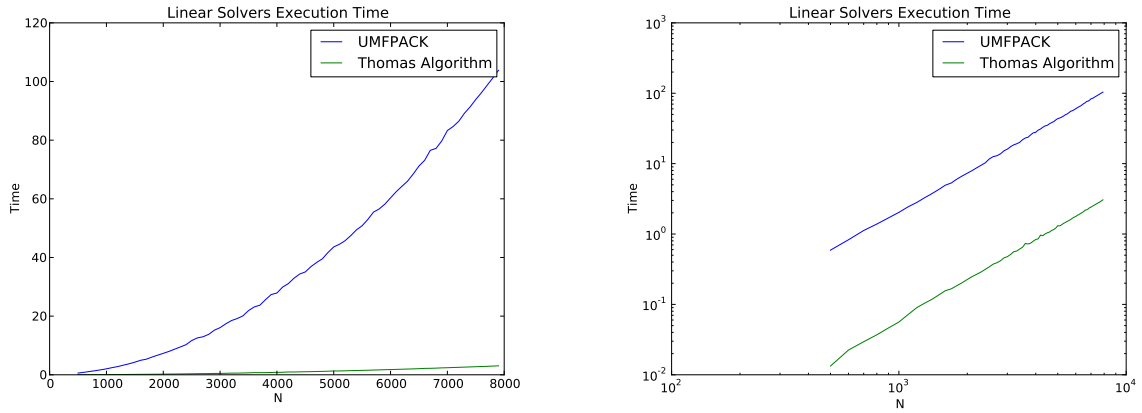
(1) moving sections of code from *Python/Cython* to *C/C++*, and

(2) changing the algorithm used to solve the linear systems.

By far, the second factor is the most significant. In the original pre-parallelized version of the code, we used the default sparse solver in *Numpy*. *Numpy*'s sparse solver uses the Unsymmetric MultiFrontal method from the *C* library *UMFPACK*. When parallelizing, we replaced the linear solver with the more efficient Thomas algorithm.

In Figure 6, we show the difference in execution time between algorithms when solving serially $2N$ linear systems of size $N \times N$. From these plots, we conclude that Thomas' algorithm is substantially more efficient.

As shown in the log-log plot, the two methods have same order of floating-point operations. Using a least-squares fit, we estimate that the execution time grows at the rate 1.914456 for Thomas's algorithm and 1.899491 for *UMFPACK*. Because we are solving $2N$ linear systems, the number of operations for both these methods is roughly $O(N)$ per linear system.



(a) (b)

FIGURE 6. Execution time when solving $2N$ linear system of size $N \times N$.

7. Conclusion

In conclusion, for the course project, we constructed a novel algorithm for the immersed boundary method. By using Guermond and Minev [11] fluid solver, the linear systems occurring in the algorithm were split directionally. This reduced overall computational complexity of solving these systems and provided a clear strategy for parallelizing.

A serial version of the algorithm was written primarily in *Python* with performance-critical sections in *Cython*. By profiling this code, we identified several bottlenecks. The most significant of these bottlenecks was the linear system solver that constituted over 25% of the total execution time.

When parallelizing the algorithm, we focused on three major bottlenecks: Linear Solver, Force Spreading, and Velocity Interpolation. Because implementing the serial version was already a significant undertaking, parallelizing the entire algorithm would have been outside the scope of a course project. Therefore, we restricted our attention to parallelizing the key portions of the algorithm.

The first attempt to parallelize the immersed boundary method involved using *Python*'s native thread support. However, this approach was found to be impractical since the Global Interpreter Lock (GIL) prevented us from utilizing multiple cores. Therefore, *Python* threads are only effective for IO bounded

tasks. Obviously, since our code is computationally bounded, *Python* threads would not be useful. Upon further research, we learnt that the GIL could be disabled when calling *C/C++* functions from *Cython*. This allowed us to write the parallelized code in *C/C++* using *OpenMP*.

Unfortunately, after parallelizing, the observed bottlenecks in the pre-parallelized code were no longer bottlenecks in our parallelized code (ran serially). As a result, we did not observe the speedup that we expected from parallelizing. Because there were no clear bottlenecks in our code, Amdahl's law placed an upper bound on the potential speedup gained from parallelizing that was estimated to be 1.428.

The increase in efficiency between the parallelized and pre-parallelized code was primarily caused by replacing the linear system solver. In the pre-parallelized code, we used *Python*'s default sparse solver that used the Unsymmetric MultiFrontal method. This was replaced with Thomas' algorithm when parallelizing the code. Although both these algorithms had the same order of operations, Thomas' algorithm was observed to be substantially more efficient. Although the execution time dropped significantly by using a more efficient linear solver, this limited the impact of parallelizing this section of code.

Lastly, although we observed limited speed, the portions of the algorithm that were parallelized had a high efficiency. Therefore, there is no reason that the speedup could not be drastically improved by parallelizing the remainder of the algorithm.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] L. A. Miller C. Hamlet, A. Santhanakrishnan. A numerical study of the effects of bell pulsation dynamics and oral arms on the exchange currents generated by the upside-down jellyfish Cassiopea xamachana. *Journal of Experimental Biology*, 214:1911 – 1921, 2011.

[3] R. Cortez, L. J. Fauci, N. Cowen, and R. H. Dillon. Simulation of swimming organisms: coupling internal mechanics with external fluid dynamics. *Computing in Science Engineering*, 6:38 – 45, 2004.

[4] R. H. Dillion, L. J. Fauci, C. Omoto, and X. Yang. Fluid dynamic models of flagellar and ciliary beating. *Annals of the New York Academy of Sciences*, 1101(1):494–505, 2007.

[5] C. Duncan, G. Zhai, and R. Scherer. Modeling coupled aerodynamics and vocal fold dynamics using immersed boundary methods. *Acoustical Society of America Journal*, 120(5):2859–2871, 2006.

[6] E. Givelberg and K. Yelick. Distributed immersed boundary simulation in titanium. *SIAM J. Sci. Comput.*, 28:1361–1378, 2006.

[7] B. Griffith. IBAMR Project. http://code.google.com/p/ibamr/, January 2012.

[8] B. E. Griffith, X. Y. Luo, D. M. McQueen, and C. S. Peskin. Simulating the fluid dynamics of natural and prosthetic heart valves using the immersed boundary method. *Journal of Computational Physics*, 1(1):137177, 2009.

[9] B. E. Griffith and C. S. Peskin. On the order of accuracy of the immersed boundary method: Higher order convergence rates for sufficiently smooth problems. *Journal of Computational Physics*, 208(1):75 – 105, 2005.

[10] Boyce E Griffith. On the volume conservation of the immersed boundary method. *Communications in Computational Physics*, 12:401–432, 2012.

[11] J. L. Guermond and P. D. Minev. A new class of massively parallel direction splitting for the incompressible navierstokes equations. *Computer Methods in Applied Mechanics and Engineering*, 200(2324):2083 – 2093, 2011.

[12] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182 – 2189, 1965.

[13] M. C. Lai and C. S. Peskin. An immersed boundary method with formal second-order accuracy and reduced numerical viscosity. *Journal of Computational Physics*, 160(2):705 – 719, 2000.

[14] L. A. Miller and C. S. Peskin. Computational fluid dynamics of clap and fling in the smallest insects. *Journal of Experimental Biology*, 208:195212, 2005.

[15] Y. Mori and C. S. Peskin. Implicit second-order immersed boundary methods with boundary mass. *Computer Methods in Applied Mechanics and Engineering*, 197(2528):2049 – 2067, 2008.

[16] M. Napolitano. A fortran subroutine for the solution of periodic block-tridiagonal systems. *Communications in Applied Numerical Methods*, 1:11 – 15, 1985.

[17] C. S. Peskin. Flow Patterns Around Heart Valves: A Numerical Method. *Journal of Computational Physics*, 10:252–271, 1972.

[18] C. S. Peskin. The immersed boundary method. *Acta Numerica*, 11:479–517, 2002.

[19] C. S. Peskin and D. M. McQueen. A three-dimensional computational method for blood flow in the heart I. Immersed elastic fibers in a viscous incompressible fluid. *Journal of Computational Physics*, 81(2):372–405, 1989.

[20] A. M. Roma, C. S. Peskin, and M. J. Berger. An adaptive version of the immersed boundary method. *Journal of Computational Physics*, 153(2):509 – 534, 1999.

[21] B. Seibold. A compact and fast Matlab code solving the incompressible Navier–Stokes equations on rectangular domains. MIT, March 2008.

Simon Fraser University, Burnaby, British Columbia, Canada

E-mail address: jwiens@sfu.ca