Distributed Immersed Boundary Simulation in Titanium

Article *in* SIAM Journal on Scientific Computing · August 2006

DOI: 10.1137/040618734 · Source: dx.doi.org

2 authors:

Edward Givelberg

13 PUBLICATIONS **169** CITATIONS

SEE PROFILE

Katherine Yelick

University of California at Berkeley and Lawren…

284 PUBLICATIONS **11,396** CITATIONS

SEE PROFILE

SIAM J. SCI. COMPUT.
Vol. 28, No. 4, pp. 1361–1378

DISTRIBUTED IMMERSED BOUNDARY SIMULATION IN TITANIUM*

E. GIVELBERG† AND K. YELICK‡

Abstract. The immersed boundary method is a general numerical method for modeling elastic boundaries immersed within a viscous, incompressible fluid. It has been applied to several biological and engineering systems, including large-scale models of the heart and cochlea. These simulations have the potential to improve our basic understanding of the biological systems they model and aid in the development of surgical treatments and prosthetic devices. Despite the popularity of the immersed boundary method and the desire to scale the problems to accurately capture the details of the physical systems, parallelization for large-scale distributed memory machines has proved challenging. The primary difficulty is in achieving a load-balanced computation, while maintaining low communication costs when modeling the interactions between the fluid and the moving immersed boundary. In this paper we describe a parallelized algorithm for the immersed boundary method that is designed for scalability on distributed memory multiprocessors and clusters of SMPs. It is implemented using the Titanium language, a Java-based language designed for high performance scientific computing. Our software package, called `IB`, takes advantage of the object-oriented features of Titanium to provide a framework for simulating immersed boundaries that separates the generic immersed boundary method code from the specific application features that define the immersed boundary structure and the forces that arise from those structures. Our results demonstrate the scalability of our design and the feasibility of large-scale immersed boundary computations with the `IB` package.

Key words. fluid-structure interactions, immersed boundary method, distributed algorithm, large-scale computation

AMS subject classifications. 65C20, 68W15, 65Y05, 76D05, 35Q30, 73K70

DOI. 10.1137/040618734

1. Introduction.

The immersed boundary method is a general numerical method for computational modeling of systems involving fluid-structure interactions. Complex systems where elastic (and possibly active) tissue is immersed in a viscous, incompressible fluid arise naturally in biology and engineering. The immersed boundary method was developed by Peskin and McQueen to study the patterns of the blood flow in the heart [17, 15]. It has subsequently been applied to a variety of problems, such as platelet aggregation during blood clotting [6], the deformation of red blood cells in a shear flow [5], the flow in collapsible thin-walled vessels [22], the swimming of eels, sperm, and bacteria [7, 4], the flow past a cylinder [14], two-dimensional [3] and three-dimensional models of the cochlea [10], valveless pumping [13], and flexible filament flapping in a flowing soap film [26]. For a recent review of the research in immersed boundary computations and further applications, see [18].

Realistic immersed boundary simulations of complex systems, such as the heart and the cochlea, require very large computing resources: The heart model experiments were carried out on the Cray T90 [19, 16], and the cochlea was constructed on the HP Superdome at Caltech [11]. Large computational grids are necessary to reduce the numerical error and to incorporate the finer details of the simulated system into

*Received by the editors November 10, 2004; accepted for publication (in revised form) February 16, 2006; published electronically August 7, 2006.
http://www.siam.org/journals/sisc/28-4/61873.html
†17 Shoham, Haifa, Israel (givelber@cims.nyu.edu).
‡Computer Science Division, University of California, Berkeley, CA 94720 (yelick@eecs.berkeley.edu).

the model. For example, higher resolution in the heart model reveals the patterns of the turbulent flow around the valves. Similarly, in the cochlea the microstructure of the organ of Corti is of crucial importance to the dynamics of the whole system. Consequently, numerical experiments with both systems often required days of dedicated computing. Both the Superdome and the Cray T90 are shared memory machines; thus the parallelization of the serial immersed boundary code was achieved mainly (but not exclusively) with the help of compiler directives. Yet, the use of finer grids in simulations of complex systems eventually leads to computations that exceed the capabilities of the available shared memory systems. Large-scale simulations involving hundreds, and perhaps thousands, of processors must be carried out on machines with distributed architecture, which require the development of algorithms such as the one presented in this paper.

The main challenge in designing distributed algorithms consists of the need to balance the cost of communicating information among the available processors with the need to efficiently utilize these processors. On most distributed architecture multiprocessor machines, communication between processors consists of an initial phase, required to establish the connection, followed by the actual transfer of the data. The cost of the initial phase, called *latency*, is typically quite high. For example, on one of the most commonly available machines, the IBM SP RS/6000, the measured latency is 7.6 microseconds, while the bandwidth is about 240 MB/sec (see [2]), making small message transfer prohibitively expensive. It is therefore necessary to impose the additional constraint of bulk communication on the design of distributed algorithms.

The immersed boundary method uses a mixed Lagrangian–Eulerian formulation and has the important property that virtually any complex model of the immersed boundary force generation can be easily incorporated. The design of the distributed algorithm should preserve this property. The fluid and the immersed material are naturally described by separate computational grids. The fluid is modeled by a three-dimensional rectangular grid, while the immersed material is typically modeled as a collection of elastic fibers (one-dimensional grids) or as an elastic shell (a two-dimensional grid). Simulation proceeds in a series of time steps, where during each time step the elastic forces are computed on the material grids and then spread to the fluid grid. The fluid equations are solved yielding a new fluid velocity, which is then interpolated to the material grids and is finally used to update their position relative to the fluid.

Distributed memory implementations of the immersed boundary method have proved quite challenging. The interaction between the fluid and immersed boundaries is the primary source of difficulties in the design of a distributed algorithm. Typically, the immersed boundary structures are not evenly distributed throughout the fluid domain and, furthermore, their position may change significantly with time. The resulting system may therefore exhibit a significant amount of irregular communication. The first step toward distributed immersed boundary computations was taken by Sabbagh, who constructed a distributed Navier–Stokes solver in his doctoral dissertation [23]. Previous attempts to design a distributed algorithm for the whole immersed boundary method included a Split-C version on the Thinking Machine CM5 and an earlier Titanium version [24] on the Cray T3E, both machines having support for lightweight (i.e., low latency) communication. Despite the great need, so far no distributed memory implementation has been used in any immersed boundary model.

The heart and the cochlea are the two examples motivating our present work in developing the algorithm and the software package `IB` for immersed boundary com-

putations in Titanium. Titanium is an explicitly parallel dialect of Java developed at the University of California-Berkeley to support high-performance scientific computing on large-scale multiprocessors, including massively parallel supercomputers and distributed-memory clusters with one or more processors per node. Other language goals include safety, portability, and support for building complex data structures. Titanium is open-source software available from the UC Berkeley Computer Science Department [1].

The complexity of an immersed boundary computation is determined by the sizes of the fluid and the immersed boundary grids, and by the size of the time step. The heart model uses a 128^3-point fluid grid, with the heart muscle and the valves modeled by a collection of elastic fibers, totaling approximately 600,000 points. The cochlea model, on the other hand, uses a 256^3-point fluid grid, with the immersed material modeled as a set of elastic shells and bony walls, totaling approximately 750,000 points. Extensive numerical experiments with the cochlea have shown that the 256^3-point fluid grid is not sufficient for many numerical experiments (see [10]). The `IB` package has already been used to construct a 128^3-point heart model and a 512^3-point cochlea model, and a 256^3-point heart model is presently under construction. This work will be described in future publications.

The rest of the paper is organized as follows. In the next section we introduce the immersed boundary equations. These equations form the basis of the numerical method, which is described in section 3. The algorithm and the data structures utilized in our implementation are outlined in section 4. Sections 5 and 6 make the case for a wider adoption of the Titanium programming language for large-scale scientific computing: section 5 surveys the main features of the Titanium programming language, and section 6 briefly surveys some features of the `IB` software package. In the following section we demonstrate the feasibility of large-scale immersed boundary computations using our software. We conclude with a summary and a discussion of our plans for further research.

2. The immersed boundary equations. The immersed boundary method is based on a mixed Eulerian–Lagrangian formulation of the fluid-immersed material system. The fluid is described in the standard Cartesian coordinates on $\mathbb{R}^3$, while the immersed material is described in a different curvilinear coordinate system. Let ρ and μ denote the density and the viscosity of the fluid, and let $\mathbf{u}(\mathbf{x},t)$ and $p(\mathbf{x},t)$ denote its velocity and pressure, respectively. The Navier–Stokes equations of a viscous incompressible fluid are

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)=-\nabla p+\mu\nabla^2\mathbf{u}+\mathbf{F}, \tag{2.1}$$

$$\nabla\cdot\mathbf{u}=0, \tag{2.2}$$

where $\mathbf{F}$ denotes the density of the body force acting on the fluid. For example, if the immersed material is modeled as a thin shell, then $\mathbf{F}$ is a singular vector field, which is zero everywhere, except possibly on the surface representing the shell. While the immersed boundary equations are valid in any domain in $\mathbb{R}^3$, we will assume that the domain is rectangular and the boundary conditions are periodic. This enables us to use fast Fourier methods for the solution of the Navier–Stokes equations (2.1) and (2.2).

Let $\mathbf{X}(\mathbf{q},t)$ denote the position of the immersed material in $\mathbb{R}^3$. For a shell, $\mathbf{q}$ takes values in a domain $\Omega\subset\mathbb{R}^2$, and $\mathbf{X}(\mathbf{q},t)$ is a 1-parameter family of surfaces

indexed by t; i.e., $\mathbf{X}(\mathbf{q},t)$ is the middle surface of the shell at time t. Let $\mathbf{f}(\mathbf{q},t)$ denote the force density that the immersed material applies on the fluid. Then

$$\mathbf{F}(\mathbf{x},t) = \int \mathbf{f}(\mathbf{q},t)\delta(\mathbf{x}-\mathbf{X}(\mathbf{q},t))\,d\mathbf{q}, \tag{2.3}$$

where δ is the Dirac delta function on $\mathbb{R}^3$. This equation merely says that the fluid feels the force that the immersed material exerts on it, but it is important in the numerical method, where it is one of the equations determining fluid-material interaction. The other interaction equation is the no-slip condition for a viscous fluid:

$$\begin{aligned}\frac{\partial \mathbf{X}}{\partial t} &= \mathbf{u}(\mathbf{X}(\mathbf{q},t),t)\\ &= \int \mathbf{u}(\mathbf{x},t)\delta(\mathbf{x}-\mathbf{X}(\mathbf{q},t))\,d\mathbf{x}.\end{aligned} \tag{2.4}$$

The system has to be completed by specifying the force $\mathbf{f}(\mathbf{q},t)$ of the immersed material. In a complicated system such as the cochlea, the immersed material consists of many different components: membranes, bony walls, an elastic shell representing the basilar membrane, and various cells of the organ of Corti, including outer hair cells, which may actively generate forces. For each such component it is necessary to specify its own computation grid and an algorithm to compute its force $\mathbf{f}$. It is in the specification of these forces that models for various system components integrate into the macromechanical model.

3. The first-order immersed boundary numerical method. We describe here a first-order immersed boundary numerical scheme, which is the easiest to implement. The main algorithmic ideas developed in section 4 apply also to other versions of the immersed boundary method, such as the formally second-order immersed boundary method [14]. The fluid equations are discretized on a periodic rectangular lattice of mesh width h. The immersed material is described by a collection of one-, two- or three-dimensional computational grids, whose mesh width is typically approximately $h/2$. Some of the computational grids used in the cochlea model are shown in Figure 3.1.

The computation proceeds in time steps of duration Δt. It will be convenient to denote the time step by the superscript. For example, $\mathbf{u}^n(\mathbf{x}) = \mathbf{u}(\mathbf{x}, n\Delta t)$. At the beginning of the nth time step, $\mathbf{X}^n$ and $\mathbf{u}^n$ are known. Each time step consists of four parts, as shown in Procedure 1 (see [18] for more details).

PROCEDURE 1. The time step of the first-order immersed boundary method.

1: Compute $\mathbf{f}^n$, the force that the immersed boundary applies to the fluid.
2: Compute $\mathbf{F}^n$, the external force on the fluid.
3: Compute $\mathbf{u}^{n+1}$, the new fluid velocity.
4: Compute $\mathbf{X}^{n+1}$, the new position of the immersed boundary.

In part 1 the force $\mathbf{f}^n$ that the immersed boundary applies to the fluid is computed. For simple materials, such as fibers, this is a straightforward computation (see [20]). More complex material models may be considered. For a detailed description of an elastic shell immersed boundary force computation, see [9]. In part 2 the interpolation equation (2.3) is used to compute the external force on the fluid $\mathbf{F}^n$ from the force $\mathbf{f}^n$ applied by the immersed boundary. The force $\mathbf{F}^n$ is used in the Navier–Stokes solver

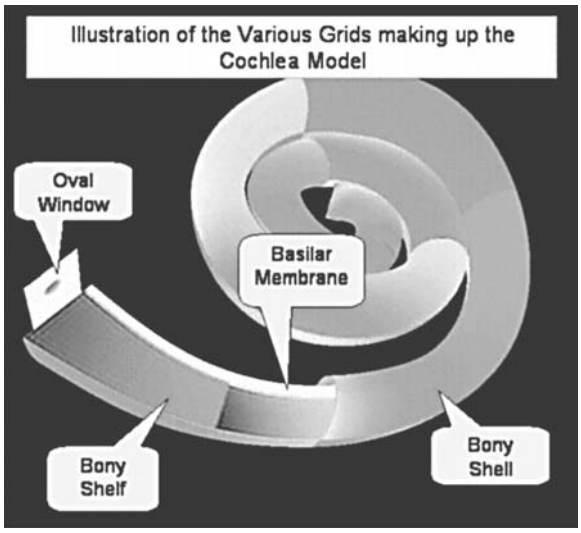

FIG. 3.1. *The immersed material grids of the cochlea model.*

in part 3 to compute the new fluid velocity $\mathbf{u}^{n+1}$. In part 4 this velocity is used in the interpolation equations (2.4) to compute the new position $\mathbf{X}^{n+1}$ of the immersed material.

We shall now describe the computations in steps 2–4 in detail, beginning with the Navier–Stokes equations. We will make use of the following difference operators which act on functions defined on the fluid lattice:

$$D_i^+\phi(\mathbf{x}) = \frac{\phi(\mathbf{x}+h\mathbf{e}_i)-\phi(\mathbf{x})}{h}, \tag{3.1}$$

$$D_i^-\phi(\mathbf{x}) = \frac{\phi(\mathbf{x})-\phi(\mathbf{x}-h\mathbf{e}_i)}{h}, \tag{3.2}$$

$$D_i^0\phi(\mathbf{x}) = \frac{\phi(\mathbf{x}+h\mathbf{e}_i)-\phi(\mathbf{x}-h\mathbf{e}_i)}{2h}, \tag{3.3}$$

$$\mathbf{D^0} = (D_1^0, D_2^0, D_3^0), \tag{3.4}$$

where $i = 1, 2, 3$ and $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$ form an orthonormal basis of $\mathbb{R}^3$.

In step 3 we use the already known $\mathbf{u}^n$ and $\mathbf{F}^n$ to compute $\mathbf{u}^{n+1}$ and p^{n+1} by solving the following linear system of equations:

$$\rho\left(\frac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\Delta t} + \sum_{k=1}^{3} u_k^n D_k^{\pm}\mathbf{u}^n\right) = -\mathbf{D}^0 p^{n+1} + \mu\sum_{k=1}^{3} D_k^+ D_k^- \mathbf{u}^{n+1} + \mathbf{F}^n, \tag{3.5}$$

$$\mathbf{D}^0 \cdot \mathbf{u}^{n+1} = 0. \tag{3.6}$$

Here $u_k^n D_k^{\pm}$ stands for upwind differencing:

$$u_k^n D_k^{\pm} = \begin{cases} u_k^n D_k^-, & u_k^n > 0, \\ u_k^n D_k^+, & u_k^n < 0. \end{cases}$$

Equations (3.5) and (3.6) are linear constant coefficient difference equations and, therefore, can be solved efficiently with the use of the fast Fourier transform (FFT) algorithm.

We now turn to the discretization of (2.3), (2.4). Let us assume, for simplicity, that $\Omega \subset \mathbb{R}^2$ is a rectangular domain over which all of the quantities related to the shell are defined. We will assume that this domain is discretized with mesh widths Δq_1, Δq_2 and that the computational lattice for Ω is the set

$$\mathbf{Q} = \{(i_1 \Delta q_1, i_2 \Delta q_2) \mid i_1 = 1, \ldots, n_1, \quad i_2 = 1, \ldots, n_2\}.$$

In step 2 the force $\mathbf{F}^n$ is computed using the following equation:

$$\mathbf{F}^n(\mathbf{x}) = \sum_{\mathbf{q} \in \mathbf{Q}} \mathbf{f}^n(\mathbf{q}) \delta_h(\mathbf{x} - \mathbf{X}^n(\mathbf{q})) \Delta \mathbf{q}, \tag{3.7}$$

where $\Delta \mathbf{q} = \Delta q_1 \Delta q_2$ and δ_h is a smoothed approximation to the Dirac delta function on $\mathbb{R}^3$ described below.

Similarly, in step 4 the update of the position of the immersed material $\mathbf{X}^{n+1}$ is done using the equation

$$\mathbf{X}^{n+1}(\mathbf{q}) = \mathbf{X}^n(\mathbf{q}) + \Delta t \sum_{\mathbf{x}} \mathbf{u}^{n+1}(\mathbf{x}) \delta_h(\mathbf{x} - \mathbf{X}^n(\mathbf{q})) h^3, \tag{3.8}$$

where the summation is over the lattice $\mathbf{x} = (hi, hj, hk)$, where i, j, and k are integers.

The function δ_h, which is used in (3.7) and (3.8), is an approximation to the Dirac delta function. It is constructed to have compact support in order to reduce the complexity of the computation. The most commonly used δ_h spreads the force (in phase 2) of a single material point to a $4 \times 4 \times 4$ cube within the fluid grid. Similarly, in step 4 of the algorithm the fluid velocity of a $4 \times 4 \times 4$ cube within the fluid grid is needed to determine the fluid velocity of an immersed material point. This function is defined by the following equation:

$$\delta_h(\mathbf{x}) = h^{-3} \phi\left(\frac{x_1}{h}\right) \phi\left(\frac{x_2}{h}\right) \phi\left(\frac{x_3}{h}\right),$$

where

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

For an explanation of the construction of δ_h, see [20].

4. The distributed algorithm. As a consequence of the mixed Eulerian–Lagrangian formulation of the immersed boundary method, the fluid and the immersed material are described by different data structures, all of which must be distributed among available processors. The main challenge in the design of the distributed algorithm lies in achieving a load-balanced computation while keeping the communication costs low. Furthermore, there are several natural design requirements that we impose on the algorithm. The algorithm must perform well for systems with sparse one- or two-dimensional immersed boundaries, as well as for systems with thick three-dimensional immersed structures. The immersed boundary may or may not move much during the simulation, and the performance of the algorithm should not depend on the mobility of the immersed boundary. In practice, the ability to easily integrate complex models of immersed material structures into the simulation model is very important. We would like to maintain this property of the immersed boundary method in distributed computations.

We begin the description of the algorithm by listing several basic design choices:

- The fluid is partitioned into a number of domains, which are assigned to all available processors.
- The immersed boundary is described by a collection of grids, each of which is wholly assigned to one of the available processors.
- The assignment of the data structures to processors is constant throughout the computation: Neither fluid points, nor material points migrate between processors.

Previous attempts to develop a distributed immersed boundary algorithm were based on different design choices, for example, splitting immersed boundary structures among processors. Our assumptions significantly simplify the development of the distributed algorithm, and while they appear to be somewhat restrictive, the resulting algorithm is nevertheless efficient, flexible, and scalable.

4.1. Interprocessor communication. As mentioned in the introduction, the interprocessor communication on most available distributed computers must be carried out in bulk, in order to avoid the penalty of high latency. Fine-grain communication is the main reason for the difficulties with scalability encountered in previous attempts to develop a distributed immersed boundary algorithm. All interprocessor communications in our algorithm use the following simple mailbox model: Each processor maintains a mailbox for incoming mail and a mailbox for outgoing mail for every processor, including itself. The act of sending mail from processor i to processor j means that the jth outgoing mailbox of processor i is copied onto the ith incoming mailbox of processor j.

4.2. Force computation. The force computation phase is the only part of the immersed boundary time step which depends on the specific structure of the immersed material. In phases 2 and 4 of the time step the material is treated simply as an unordered collection of points. In most applications force computation is by far the cheapest part of the immersed boundary time step.

The assumption that all data structures related to a given piece of the immersed boundary are completely contained in the memory of some processor implies that the computation of the force in phase 1 is entirely local, i.e., requires no communication, and is very efficient. However, we will see that in order to carry out the remaining, more expensive, phases of the immersed boundary time step efficiently, it is necessary to distribute the immersed material structures as evenly as possible among all available processors. In practice, certain portions of the immersed boundary may be quite large, leading to load imbalance in the computation. Our experience has shown that it is generally possible to subdivide such large structures among processors in an efficient way. For example, the basilar membrane in the cochlea model is a relatively large structure, modeled as an elastic shell. In order to achieve a load balanced computation we have partitioned the shell into a number of components assigned to different processors. As a result, the computation of the elastic force of the shell requires a preliminary communication phase in which boundary data for each component are exchanged between processors that own neighboring shell components. Each processor bundles the data it communicates, sending a single message to each of its neighbors. The total amount of data exchanged in this phase is typically very small, and the force computation phase remains significantly cheaper than the other phases of the immersed boundary time step. In order to simplify the discussion, in the remainder of this paper we assume that *each immersed boundary component is assigned in its entirety to one of the available processors.*

4.3. The fluid solver. The solution of the discretized Navier–Stokes equations (3.5)–(3.6) is the central and most expensive part among the four parts that compose each time step of the first-order immersed boundary numerical method. The general framework of the immersed boundary method allows the use of fluid solvers different from the one described in section 3. Our design ensures that this remains true in the distributed case as well. It will be convenient, however, to describe the approach in the case of the specific fluid solver we have implemented.

Our fluid solver is perhaps the simplest possible distributed solver. It depends on the availability of efficient distributed three-dimensional FFT functions. We assume that the transform data is partitioned into slabs. When a transform of an $N_1 \times N_2 \times N_3$-point array $A[i_1, i_2, i_3]$ is computed using p processors (where N_1 is divisible by p), processor q stores the data slab

$$A[i_1, i_2, i_3]: \quad i_1 = \frac{N_1}{p}q, \ldots, \frac{N_1}{p}(q+1)-1, \quad i_2 = 0, \ldots, N_2-1, \quad i_3 = 0, \ldots, N_3-1.$$

Accordingly, fluid variables, such as velocity, pressure, and body force, are stored in such slabs: U_1, U_2, U_3, P and F_1, F_2, F_3. The application of the upwind differencing operator in (3.5) makes it necessary, however, to pad each such slab with two ghost planes corresponding to $i_1 = \frac{N_1}{p}q - 1$ and $i_1 = \frac{N_1}{p}(q+1)$. (Notice that because of the periodicity of the domain the calculation of the index i_1 is carried out modulo N_1.) In addition to the fluid variables mentioned above, we will also require three global fluid variables C_1, C_2, and C_3 for temporary storage, each occupying one slab per processor. The variables C_1, C_2, and C_3 store complex data on which a complex-to-complex Fourier transform is performed. The pressure variable is not required for time-stepping, but the pressure can be computed, if necessary, using additional storage, C_4, and one additional inverse Fourier transform computation. We require all three dimensions of the fluid slabs to be divisible by 4 (not counting the ghost planes in the first dimension), in order to facilitate the fluid cache implementation (see below).

We rewrite the Navier–Stokes equations (3.5)–(3.6) in the following form:

$$\rho\frac{\mathbf{u}^{n+1}}{\Delta t} + \mathbf{D}^0 p^{n+1} - \mu\sum_{k=1}^{3} D_k^+ D_k^- \mathbf{u}^{n+1} = \mathbf{C^n}, \tag{4.1}$$

$$\mathbf{D}^0 \cdot \mathbf{u}^{n+1} = 0, \tag{4.2}$$

where

$$\mathbf{C^n} = \rho\frac{\mathbf{u}^n}{\Delta t} - \rho\sum_{k=1}^{3} u_k^n D_k^{\pm}\mathbf{u}^n + \mathbf{F}^n. \tag{4.3}$$

The Navier–Stokes solver is shown in Procedure 2. At the beginning of this computation the slabs of U_1, U_2, and U_3 contain the values of the fluid velocity computed at the end of the nth time step. The first step is a preliminary communication phase where the ghost planes of the slabs corresponding to the three velocity components are exchanged. This can be done efficiently, with each processor sending a single message to each of its two neighbors. The barrier command synchronizes the processors, ensuring that all of the processors have completed their work before the next step can begin. At the end of the communication phase every processor has all the necessary data to compute the values stored in the slabs C_1, C_2, C_3 using (4.3). Notice that the right-hand side of (4.3) is complex data, while the left-hand side is real. The

PROCEDURE 2. The Navier–Stokes solver.

1: Exchange ghost planes for U_1, U_2, U_3;
2: *barrier*;
3: Compute C_1, C_2, C_3 using (4.3);
4: *barrier*;
5: Compute forward FFT of C_1, C_2, C_3;
6: *barrier*;
7: Solve transformed equations in C_1, C_2, C_3;
8: *barrier*;
9: Compute inverse FFT of C_1, C_2, C_3 and store the result in U_1, U_2, U_3.

Navier–Stokes equations (4.1)–(4.2) can now be solved using the distributed FFT. We assume that the FFT functions perform the transform "in place," so that the solution of the transformed equations is again a local computation requiring no communication. After the inverse Fourier transform has been completed, we copy the new fluid velocity from the complex C_1, C_2, C_3 into the real slabs U_1, U_2, U_3.

4.4. Fluid-structure interactions. The main difficulty in the design of the distributed immersed boundary algorithm lies in efficiently carrying out phases 2 and 4 of the immersed boundary time step. These phases require communication of information between the fluid grid and the immersed boundary grids. Computationally, these phases are expensive: On a single processor, depending on the immersed boundary system being simulated, each of these two phases may account for as much as 20–30% of the computing time.

In phase 2 immersed boundary forces must be communicated to the processors that own the appropriate portions of the fluid grid. In many applications, however, the immersed boundary interacts with only a small portion of the fluid grid, which may be owned by a fraction of all available processors. Communicating the immersed boundary forces to these processors will lead to a load imbalance in the computation. Furthermore, since each immersed boundary point interacts with a $4 \times 4 \times 4$ cube within the fluid grid, it is not easy to describe the set of fluid points that a given immersed boundary grid interacts with. To solve this problem we break up phase 2 into several stages and introduce a specialized cache-like data structure in each processor. We shall refer to this data structure simply as *fluid cache*.

The force-spreading phase of the immersed boundary method is described in Procedure 3. First, each processor spreads the force of every immersed boundary grid point it owns into its fluid cache. At the end of this phase the processor fluid cache contains the representation of the forces applied to the portion of the fluid grid in the vicinity of the material grids possessed by that processor. It contains, however, only the forces contributed by the grids of that processor and not the forces due to grids owned by other processors. Next, every processor sends the contents of its cache to the processors that own the corresponding portions of the fluid force array. This communication is carried out in bulk: All information intended for a given recipient is bundled into the mailbox intended for that recipient and then sent as a single message. In the worst case, in this stage every processor sends a single large message to every other processor, including itself. Finally, when the communication has been completed every processor scans the messages it received and updates the portion of the fluid force array that it owns. At the end of this stage each force slab contains contributions from forces of *all* immersed material grids.

PROCEDURE 3. Force spreading for processor i $(1 \leq i \leq p)$.

```
1: for each immersed boundary grid do
2:   if the grid is owned by processor i then
3:     for every grid point do
4:       spread the grid point force f^n into the fluid cache;
5:     end for
6:   end if
7: end for
8: Pack the cache into mailboxes.
9: Send mail to each processor.
10: barrier;
11: for j = 1, ..., p do
12:   add the force data received from processor j to the force slab F_1, F_2, F_3;
13: end for
```

The first, local stage of force spreading is by far the most time-consuming part of the distributed procedure. Under most circumstances, the assignment of immersed boundary grids to processors at the start of the simulation can ensure that the total number of immersed boundary grid points owned by each processor is approximately the same. This will ensure a nearly perfect load balance distribution in the local force-spreading part of the computation. The communication stage depends on the total number of immersed boundary grid points and on their relative position. Significant savings can be achieved when portions of the immersed boundary that remain in proximity throughout the simulation are assigned to the same processor. In practice, this is usually known to the designer of the simulation experiment, who can carry out this assignment at the start of the simulation.

The final phase of the immersed boundary time step involves computing the new position of the immersed boundary by interpolating the new fluid velocity from the fluid grid to the immersed boundary grids. This phase is similar to the force-spreading phase, except that the fluid velocity data, instead of the fluid force data, is now communicated, and the information is propagating in the opposite direction: from the fluid grid to the immersed material grids. It is important to notice that the fluid portion that interacts with a given immersed boundary point in stage 2 is exactly the same as in stage 4. This information is stored in the fluid cache (see below) and is used in line 2 of Procedure 4.

4.5. The fluid cache. To finish the description of the computation we now provide the details of the fluid cache construction. We imagine the rectangular fluid grid as partitioned by a lattice into a collection of small $4 \times 4 \times 4$ cubes. Since we require the fluid slab dimensions to be divisible by 4, each fluid cube is completely contained in one slab. The fluid cache data structure in every processor consists of an $(N_1/4) \times (N_2/4) \times (N_3/4)$ array of pointers to cubes.

Initially this structure is empty, and a cube is allocated only when a point inside the cube is referenced. The fluid cache is easy to update, yet it does not contain an excessive amount of information. In addition to the pointer array, a linked list containing all the indices of the cubes that have already been allocated is maintained. Thus, at the end of the force-spreading phase each processor will have a list of all fluid cubes that interact with the portions of the immersed boundary that it owns. In effect, the data in the fluid cache describes within the fluid a narrow neighborhood of the

PROCEDURE 4. Move for processor i $(1 \le i \le p)$.

```
1: for j = 1, ..., p do
2:    copy the appropriate portions of U_1, U_2, U_3 into the mailbox of processor j;
3: end for
4: Send mail to every processor.
5: barrier;
6: Unpack the mailboxes into the fluid cache.
7: for each immersed boundary grid do
8:    if the grid is owned by processor i then
9:       for every grid point do
10:         advance grid point coordinates using the velocity data in the fluid cache;
11:      end for
12:   end if
13: end for
```

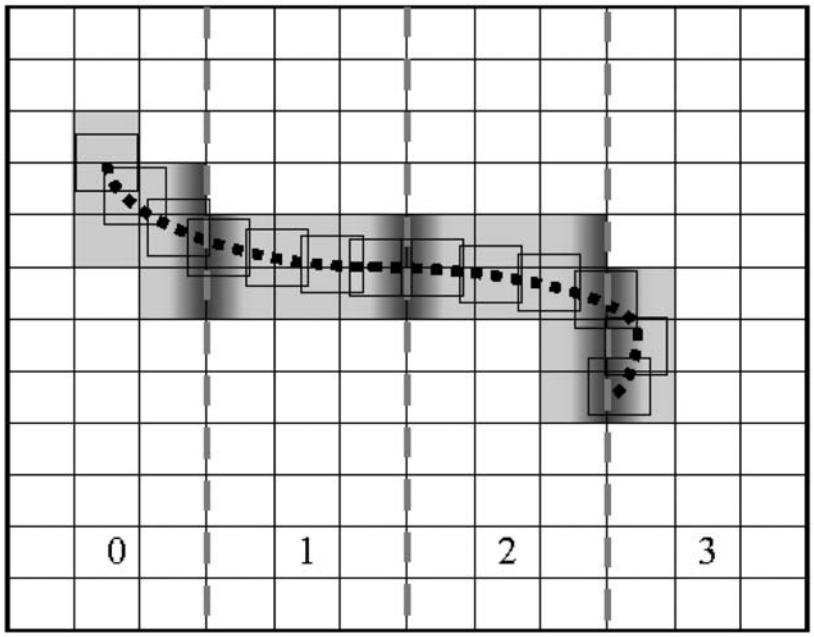

FIG. 4.1. *An illustration of the fluid cache: each point of the immersed boundary interacts with a* $4 \times 4 \times 4$ *cube of fluid around it; the shaded region describes the union of the fluid cache structures of all four processors. The cache of processor i is the intersection of slab i with the shaded region.*

immersed boundary owned by that processor as a collection of cubes (see Figure 4.1). The processor will then send cubes of fluid containing force data to processors that own these cubes. (Recall our assumption that fluid slab dimensions are divisible by 4, which ensures that each fluid cube is owned by exactly one processor.) Each recipient of a list of cubes containing fluid force data will record the cube indices and in the last stage of the algorithm it will return these cubes containing fluid velocity data to the sender.

Several issues related to cache management require further research. At present, we do not deallocate the cubes in the cache at the end of each time step. At the beginning of each time step we use the list of the cube indices to initialize all the allocated cubes to zero. This works well for portions of immersed boundary that do not move much during the simulation. For a highly mobile immersed boundary it may be necessary to implement a garbage collection mechanism in the fluid cache. While the size of the fluid cache in each processor is proportional to the total size of the fluid, our construction is practical because many of the currently available multiprocessor systems possess large amounts of memory (up to 32 GB per processor). Memory requirements for the fluid cache can be significantly reduced by implementing the cache using more complex data structures, but in such an implementation cache access may be slower. The $4 \times 4 \times 4$ size of the fluid cache cubes is not necessarily optimal. Good results have also been obtained with $4 \times 4 \times 1$ blocks. The choice of the block size in the fluid cache is not directly related to the width of support of the discrete delta function δ_h. With smaller cache blocks the total size of the cache in every processor is larger, while potentially less information needs to be communicated during the fluid-structure interaction phases.

5. The Titanium programming language. We have built a software package, called `IB`, implementing the distributed immersed boundary algorithm described above using the Titanium programming language. This section describes some of the features of the language that make it particularly suitable for large-scale scientific computing projects.

While much of scientific computing is still carried out in Fortran, the complexity of the simulated systems naturally leads to complex data structures. Modern multiprocessor computers require language-level tools to develop distributed algorithms, and programming languages have evolved to support object-oriented programming and to provide mechanisms for distributed programming. These tools have proved to be very successful in many fields, but they have rarely been used in the context of high-performance scientific computing because they often carry with them a performance penalty.

The Titanium programming language was developed with the goal of providing a modern language to meet the needs of scientific supercomputing. It is an explicitly parallel Java-based language designed to support high-performance scientific computing on large-scale multiprocessors, including massively parallel supercomputers and distributed-memory clusters with one or more processors per node [12, 25].

Titanium is a global address space language, closely related to UPC, Co-Array Fortran, and several older research languages based on C and C++. It combines the parallelism model commonly found in message passing modes with the shared address space found in shared memory models. In particular, Titanium uses a static parallelism (SPMD) model in which the number of parallel threads is determined at program startup time. It provides a global address space abstraction through which one thread may directly read or write to the memory of another thread, although the address space is logically partitioned so that each thread has a portion of the address space that is nearby. Programmers have control over data layout, synchronization, and load balancing for high performance, while the global address space simplifies programming but allows for direct expression of distributed data structures.

In spite of the global address space, the Titanium implementation runs on essentially any parallel machine, including shared memory multiprocessors, clusters of uniprocessors, and clusters of SMPs. On machines with hardware support for shared

memory, the compiler generates conventional load and store instructions to access memory that is associated with another thread. On a distributed memory platform, lightweight communication calls are inserted automatically when the accessed data structures reside on a remote processor. Titanium programs can run unmodified on uniprocessors, shared memory machines, and distributed memory machines—performance tuning may be necessary to arrange an application's data structures for distributed memory, but the functional portability allows for development on shared memory machines and uniprocessors.

Titanium preserves the safety properties of Java, which prevent accessing data that is unallocated through array bounds checking, strong typing, and automatic memory management. In addition to the parallelism model, which replaces conventional Java threads, Titanium adds support to improve programming for scientific applications. These include the following:

- user-defined immutable classes (often called "lightweight" or "value" classes);
- flexible and efficient multidimensional arrays with a rich set of operations for defining and manipulating the index set of an array;
- zone-based memory management, in addition to standard garbage collection;
- a type system for expressing and inferring locality and sharing properties of distributed data structures;
- compile-time prevention of deadlocks on barrier synchronization;
- a library of useful parallel collective operations such as barriers, broadcasts, and reductions;
- operator overloading;
- parameterized classes similar to a C++ style template.

6. The `IB` software package. The `IB` software package provides a collection of general-purpose classes necessary for the construction of immersed boundary simulations. This section briefly mentions some of the features of the package as an illustration of the importance of object-oriented programming in construction of large-scale computational models.

The fluid solver uses the FFTW software [8] to compute three-dimensional, distributed Fourier transforms. The fluid solver is contained in the `Fluid` class, which is defined by specifying fluid dimensions, mesh width, density, and viscosity:

```
Fluid fluid = new Fluid(N1, N2, N3, dx, rho, mu);
```

More complicated fluid models can be created using class inheritance. For example, the fluid in the heart model contains sources and sinks. Such a class can be defined as an extension of `IB.Fluid`. The new class will inherit many useful methods for manipulating fluid data structures, while overriding several methods, such as the fluid solver.

The specification of material grids and their type depends on the particular application. The `IB` package provides a number of classes for this purpose. Local properties of the material must be specified in a user-defined class derived from `IB.GridPoint`, and its global properties in a user-defined class derived from `IB.Grid`. For example, an elastic shell material type can be defined by deriving a class `Shell` and a class `ShellPoint`. The class `ShellPoint` naturally contains local elastic parameters of the material. On the other hand, the elastic force of the shell must be specified by defining a `ComputeForce` method within the `Shell` class.

The `IB` package also provides a `GridArray` class that enables handling hierarchical collections of material structures. In the cochlea model several surfaces are further subdivided into collections of grids; in the heart model the muscle fibers are arranged in groups, which are further arranged in layers.

Finally, the `IB` package provides a collection of methods to control the simulation and collect and analyze its output.

7. Software performance. Realistic immersed boundary models, such as the heart or the cochlea, require a significant amount of work to construct. We have therefore constructed a number of simple test models for the purpose of testing the distributed algorithm and tuning the performance of the `IB` software. Each such test model consists of a number of rectangular plates immersed in fluid. The precise nature of the elastic material is not important for the complexity of the immersed boundary computations because the elastic force calculation phase is negligible in time with respect to the other phases of the algorithm. The complexity of the computation is affected only by the total number of immersed boundary points and by their motion relative to the fluid. We have partitioned the immersed material into a number of grids, distributing an equal number of grids and an equal number of points among the processors. We expect to be able to achieve such a distribution in a realistic simulation. The heart, for example, is modeled by thousands of distinct fibers, which can be easily assigned to processors to achieve an even balancing of the load.

In immersed boundary computations, refining the fluid grid necessitates refining the immersed material grids accordingly. This is required to prevent the fluid leaking through the immersed boundary (for volume conservation in immersed boundary computations, see [21]). The cochlea model consists of a number of surfaces and, assuming the fluid grid is of size N^3, its total number of immersed boundary points is proportional to N^2. The heart muscle, on the other hand, though being modeled by a collection of one-dimensional fibers, occupies a three-dimensional volume, and therefore the total number of immersed boundary points in the heart model is proportional to N^3. Accordingly, we have tested the performance of the software on several models with varying numbers of immersed boundary points. Each of our test models contained a number of identical $N \times N$-point plates. We considered models with fluid grid of size N^3, where $N = 256$ and $N = 512$, and the number of plates $n = 1, 16$, and $N/8$. Accordingly, in the discussion below we refer to a model with n plates of size $N \times N$ as an (N, n) model. The models with $n = 1$ possess very little immersed material and therefore provide an insight into the performance of the fluid solver part of the algorithm. The $n = 16$ models resemble the cochlea in the size of their immersed material, while the $n = N/8$ models resemble the heart. The flow in each test model was generated by a periodic sinusoidal force of small amplitude applied to each of the plates. After a short initial phase the performance of the algorithm for each subsequent time step was essentially the same.

Our experiments were carried out on the IBM SP RS/6000 at the National Energy Research Scientific Computing Center (NERSC). This is a distributed memory computer possessing a large number of 16-processor nodes (currently 380 nodes), where each node has between 16 and 64 GB of memory. All of our tests were carried out on either 1, 2, 4, or 8 nodes, with the total number of processors used being 16, 32, 64, or 128. Figure 7.1 demonstrates the scaling of our software when a varying number of processors is used to compute two test problems. For the $(256, 1)$ we also present the data from experiments with 1, 2, 4, and 8 processors, all within a single node. Because our present algorithm requires the width of the fluid slabs (without the ghost planes) to be divisible by 4, we can utilize at most 64 processors on 256^3-fluid problems, and at most 128 processors on 512^3-fluid problems. Table 7.1 summarizes the wall-clock time per time step results for a number of test models, as well as the total number of floating point operations performed (in billions). The computations with

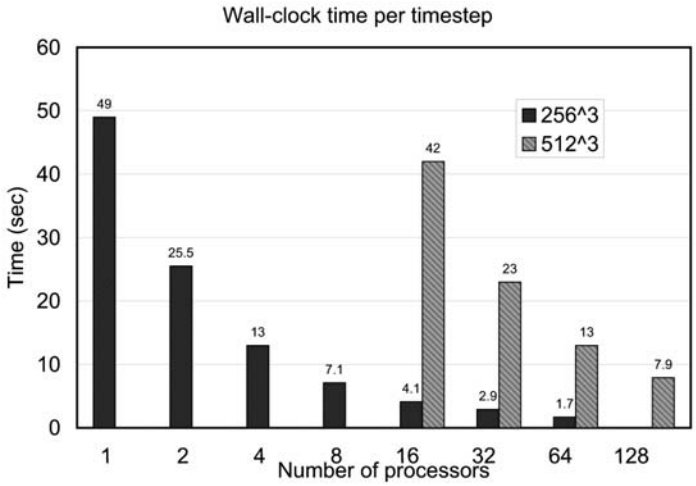

FIG. 7.1. *Execution wall-clock time (in seconds) per time step for the* $(256, 1)$ *and* $(512, 1)$ *models as a function of the number of processors.*

TABLE 7.1
Wall-clock time per time step for various test models.

Model name	Fluid grid size	Total immersed boundary size	Number of processors	Total GFLOPs	Wall-clock time
(256, 1)	256^3	$256^2 = 64$K points	64	4.34	1.7 sec
(256, 16)	256^3	$16 \times 256^2 = 1$M points	64	5.5	2.4 sec
(256, 32)	256^3	$32 \times 256^2 = 2$M points	64	6.78	3.1 sec
(512, 1)	512^3	$512^2 = 256$K points	128	38.4	7.9 sec
(512, 16)	512^3	$16 \times 512^2 = 4$M points	128	43.2	9.6 sec
(512, 64)	512^3	$64 \times 512^2 = 16$M points	128	58.1	15.2 sec

the 256^3-point fluid grid were performed on 64 processors, while the 512^3-point fluid grid computations used 128 processors.

Finally, Table 7.2 shows the timings of the main phases of each time step. The second row refers to the spreading of the forces from the immersed boundary into the local fluid cache structure. Rows 6–10 detail the performance of the fluid solver: "Upwind" refers to the initial step of the fluid solver which uses the slabs F_1, F_2, F_3 and applies the upwind differencing operator to U_1, U_2, U_3. Three Fourier transforms are applied to the components of the result of this computation. The fluid equations are then solved in the Fourier space, and the solution is obtained by means of inverse Fourier transforms. The total time devoted to the fluid solver is recorded in row 11. The last phase of the time step, named "move material," involves both the interpolation of the fluid velocity from cubes to immersed boundary grids and the updating of the immersed boundary position.

The results presented in Table 7.2 enable us to assess the scalability of the individual phases of the computation. The lines labeled with "spread force" and "move material" involve local computations with the fluid cache and show excellent scalability. It is interesting that a significant amount of time is necessary to perform local copy operations ("pack" and "unpack") to and from the fluid cache. No floating point operations are performed in these phases. The communication phases show excellent scaling: Because only fluid information is communicated, doubling the number of

TABLE 7.2
Breakdown of the wall-clock time per time step for various test models. (See explanation in the body of the article.)

	(256, 16)	(256, 32)	(512, 16)	(512, 64)
Compute material force	0.015	0.027	0.028	0.08
Spread force	0.25	0.49	0.52	1.97
Pack force cubes	0.012	0.021	0.023	0.08
Send force cubes	0.22	0.28	0.52	1.4
Update force slabs	0.08	0.1	0.43	0.84
Upwind	0.29	0.29	1.49	1.46
$3\times$ FFT	0.39	0.38	2.12	2.1
Solve	0.06	0.06	0.25	0.25
$3\times$ FFT^{-1}	0.39	0.41	2.21	2.29
Copy fluid velocity	0.06	0.06	0.24	0.25
Total (fluid solver)	1.3	1.31	6.63	6.78
Pack velocity cubes	0.04	0.07	0.14	0.47
Send velocity cubes	0.23	0.28	0.58	1.57
Unpack velocity cubes	0.01	0.02	0.034	0.078
Move material	0.22	0.47	0.47	1.86
Total (time step)	2.4	3.09	9.39	15.16

immersed boundary points may require much less than double the amount of fluid information to be communicated.

8. Summary and conclusions. We have developed an efficient algorithm for immersed boundary simulations on distributed memory multiprocessor systems. The main difficulty in the design of the algorithm arises because of the need to communicate information between the fluid and the immersed boundary data structures, which are distributed across multiple processors. The difficulty of achieving load balance and maintaining low communication cost in the fluid-structure interaction is exacerbated by the hardware-imposed restriction that the interprocessor communication be carried out in bulk. We resolve this problem with the help of a specialized cache-like data structure, the fluid cache, to communicate information between processors. The fluid cache enables an efficient implementation of the fluid-structure interaction portion of the immersed boundary time step.

We have implemented the algorithm in the software package `IB` using the Titanium programming language. In our implementation the fluid solver uses the FFTW functions for efficient distributed FFT computations.

The `IB` package utilizes the object-oriented features of Titanium and its built-in facilities for generating efficient code, and provides the user with a set of versatile classes and utilities to build large-scale immersed boundary applications. A 512^3 cochlea model based on the `IB` package has been tested, and Peskin and McQueen's 128^3 heart model has already been reconstructed using the `IB` package. A 256^3 heart model is currently being built. This work will be reported in future publications. In this paper we described the development of the basic algorithm and the software performance results that enable such large-scale immersed boundary computations.

At present, these are the largest models that are practical. Indeed, since a single typical experiment with the cochlea model may require more than 10,000 time steps, we estimate that in the case of a 512^3-point fluid grid it will take more than 3 days to complete. Simulating a single beat of the heart, on the other hand, requires on the order of 100,000 time steps. Such an experiment with a 256^3 model could be completed in fewer than 3 days.

The main practical limitation in using the FFTW is the limit in the amount of parallelism that is achievable: The FFTW requires partitioning the fluid data along the first dimension and at most N_1 processors can be deployed in a transform of size $N_1 \times N_2 \times N_3$. In order to be able to simulate larger immersed boundary systems it is necessary to develop an efficient fluid solver which uses more than N_1 processors. Such a fluid solver will require a more complex partition of the fluid into subdomains. We expect the extension of our fluid cache construction, and the distributed algorithm we have presented, to the case of the more general fluid partition to be straightforward.

Our algorithm assumes that the number of immersed boundary points is constant throughout the computation and the partition of these points to processors is carried out at the beginning of the computation and is held fixed throughout the computation. The case where immersed material is either created or destroyed during the simulation introduces an additional difficulty. In such a system the immersed boundary data structures may become concentrated in only a few of the available processors, leading to an unbalanced computation. This problem can be addressed by allowing the immersed boundary data structures to migrate between processors, but periodic reassignment of immersed boundary data structures to processors may be computationally expensive.

REFERENCES

[1] *Titanium Project Home Page*, http://www.cs.berkeley.edu/projects/titanium/ (24 January 2006).

[2] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick, *An evaluation of current high-performance networks*, in Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS), 2003.

[3] R. P. Beyer, *A computational model of the cochlea using the immersed boundary method*, J. Comput. Phys., 98 (1992), pp. 145–162.

[4] R. Dillon, L. J. Fauci, and D. Gaver, *A microscale model of bacterial swimming, chemotaxis and substrate support*, J. Theoret. Biol., 177 (1995), pp. 325–340.

[5] C. D. Eggleton and A. S. Popel, *Large deformation of red blood cell ghosts in a simple shear flow*, Phys. Fluids, 10 (1998), pp. 1834–1845.

[6] L. J. Fauci and A. L. Fogelson, *Truncated Newton method and modeling of complex immersed elastic structures*, Comm. Pure Appl. Math., 46 (1993), pp. 787–818.

[7] L. J. Fauci and C. S. Peskin, *A computational model of aquatic animal locomotion*, J. Comput. Phys., 77 (1988), pp. 85–108.

[8] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, 1998, pp. 1381–1384.

[9] E. Givelberg, *Modeling elastic shells immersed in fluid*, Comm. Pure Appl. Math., 57 (2004), pp. 283–309.

[10] E. Givelberg and J. Bunn, *A comprehensive three-dimensional model of the cochlea*, J. Comput. Phys., 191 (2003), pp. 377–391.

[11] E. Givelberg, J. J. Bunn, and M. Rajan, *Detailed simulation of the cochlea: Recent progress using large shared memory parallel computers*, in Proceedings of the 2001 International Mechanical Engineering Congress, New York, 2001.

[12] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, *Titanium Language Reference Manual*, Technical report UCB//CSD-01-1163, Computer Science Division (EECS), University of California, Berkeley, CA, 2001.

[13] E. Jung and C. S. Peskin, *Two-dimensional simulations of valveless pumping using the immersed boundary method*, SIAM J. Sci. Comput., 23 (2001), pp. 19–45.

[14] M.-C. Lai and C. S. Peskin, *An immersed boundary method with formal second order accuracy and reduced numerical viscosity*, J. Comput. Phys., 160 (2000), pp. 705–719.

[15] D. M. McQueen and C. S. Peskin, *Computer-assisted design of pivoting-disc prosthetic mitral valves*, J. Thorac. Cardiovasc. Surg., 86 (1983), pp. 126–135.

[16] D. M. McQueen and C. S. Peskin, *Shared-memory parallel vector implementation of the immersed boundary method for the computation of blood flow in the beating mammalian heart*, J. Supercomputing, 11 (1997), pp. 213–236.
[17] C. S. Peskin, *Flow Patterns around Heart Valves: A Digital Computer Method for Solving the Equations of Motion*, Ph.D. thesis, Albert Einstein College of Medicine, Yeshiva University, New York, 1972.
[18] C. S. Peskin, *The immersed boundary method*, Acta Numer., 11 (2002), pp. 479–517.
[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*, J. Comput. Phys., 81 (1989), pp. 372–405.
[20] C. S. Peskin and D. M. McQueen, *A general method for the computer simulation of biological systems interacting with fluids*, in Proceedings of the SEB Symposium on Biological Fluid Dynamics, Leeds, England, 1994.
[21] C. S. Peskin and B. F. Printz, *Improved volume conservation in the computation of flows with immersed elastic boundaries*, J. Comput. Phys., 105 (1993), pp. 33–46.
[22] M. E. Rosar and C. S. Peskin, *Fluid flow in collapsible elastic tubes: A three-dimensional numerical model*, New York J. Math., 7 (2001), pp. 281–302.
[23] H. G. Sabbagh, *Solving the Navier–Stokes Equations on a Distributed Parallel Computer*, Ph.D. thesis, New York University, New York, 1996.
[24] S. M. Yau, *Experiences in Using Titanium for Simulation of Immersed Boundary Biological Systems*, Master's report, 2002.
[25] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, *Titanium: A high-performance Java dialect*, Concurrency: Practice and Experience, 10 (1998), pp. 825–836.
[26] L. Zhu and C. S. Peskin, *Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method*, J. Comput. Phys., 179 (2002), pp. 452–468.