DISTRIBUTED IMMERSED BOUNDARY SIMULATION IN TITANIUM

E. GIVELBERG AND K. YELICK

Abstract. The immersed boundary method is a general technique for modeling elastic boundaries immersed within a viscous, incompressible fluid. The method 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 machine has proven challenging. The primary reason is a classic locality and load balance tradeoff that arises in distributing the immersed boundary data structure across processors. 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 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.

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 [15, 13]. It has subsequently been applied to a variety of problems, such as platelet aggregation during blood clotting [5], the deformation of red blood cells in a shear flow [4], the flow in collapsible thin walled vessels [3], the swimming of eels, sperm and bacteria [6, 2], the flow past a cylinder [12], two-dimensional [1] and three-dimensional models of the cochlea [8], valveless pumping [11] and flexible filament flapping in a flowing soap film [21]. For a recent review of the research in immersed boundary computations and further applications see [16].

Immersed boundary simulation of complex system such as the heart and the cochlea requires very large computing resources: the heart model experiments were carried out on the Cray T90 [14] and the cochlea was constructed on the HP Superdome at Caltech [9]. Numerical experiments with both systems often required days of dedicated computing. Both the Superdome and the Cray T90 are shared memory machines, so the parallelization of the serial immersed boundary code was achieved mainly (but not exclusively) with the help of compiler directives. The great complexity of the simulated systems typically necessitates the use of finer grids leading to larger computations that exceed the capabilities of shared memory systems available. Such finer computational grids are necessary to reduce the numerical error and to incorporate finer details of the system into the model. For example, higher resolution in the heart model can help us understand the turbulent flow around the valves. Similarly, in the cochlea, the micro-structure of the organ of Corti is of crucial importance to the dynamics of the system.

The heart and the cochlea are the two examples motivating our present work developing the algorithm and the software package `IB` for immersed boundary computations in Titanium. Titanium is an explicitly parallel dialect of Java developed at UC 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.

Distributed memory implementations of the immersed boundary method have proved quite challenging. Previous attempts included a Split-C version that scales well on the Thinking Machine CM5, and an earlier Titanium version [19] that scales on the Cray T3E, both machines with support for lightweight communication. Despite the great need, so far no distributed memory implementation has been used in any immersed boundary model. The interaction between the fluid and immersed boundaries are the primary source of both programming and performance problems. While the structure of the immersed boundaries depend on the application domain, it is generally not distributed evenly throughout the fluid domain. If the boundary data is distributed uniformly across processors, the resulting system has a significant amount of irregular communication that arises as the forces between the boundaries and fluid interact. The global address space Titanium aid in programming, but performance can still be problematic if the hardware does not perform fine-grained communication efficiently.

Immersed boundary computations are based on a Lagrangean formulation, where separate computational grids are maintained for the fluid and for the material immersed in it. 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). This framework provides for a straightforward incorporation of complex models of the immersed boundary. Simulation proceeds in a series of time steps, where during each time step the elastic forces are computed on the material grids, 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.

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 [8]). Our goal in developing the Titanium immersed boundary software is to construct a heart model and a cochlea model based on 512^3-point fluid grids.

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. Section 4 surveys the main features of the Titanium programming language, which we use to implement the numerical method. The algorithm and the data structures utilized in our implementation are outlined in section 5. In the following section we demonstrate the feasibility of large scale immersed boundary computations using our software. We conclude with a discussion of our plans for further performance improvements in the Titanium immersed boundary software.

2. The Immersed Boundary Equations

The immersed boundary method is based on a Lagrangean formulation of the fluid-immersed material system. The fluid is described in the standard cartesian coordinates on $\mathbf{R}^3$, 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:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{F} \tag{1}$$

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

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. The numerical method uses a discretization of the Navier-Stokes equations (1) and (2) on a periodic rectangular grid.

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

$$\mathbf{F}(\mathbf{x},t) = \int \mathbf{f}(\mathbf{q},t)\delta(\mathbf{x}-\mathbf{X}(\mathbf{q},t))\,d\mathbf{q}, \tag{3}$$

where δ is the Dirac delta function on $\mathbf{R}^3$. 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:

$$\begin{aligned}\frac{\partial \mathbf{X}}{\partial t} &= \mathbf{u}(\mathbf{X}(\mathbf{q},t),t)\\ &= \int \mathbf{u}(\mathbf{x},t)\delta(\mathbf{x}-\mathbf{X}(\mathbf{q},t))\,d\mathbf{x}.\end{aligned} \tag{4}$$

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 macro-mechanical 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 and has the important advantage of modularity: incorporating various models of immersed elastic material is straightforward.

Let Δt denote the duration of a time step. 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 n-th time step $\mathbf{X}^n$ and $\mathbf{u}^n$ are known. Each time step proceeds as follows.

(1) Compute the force $\mathbf{f}^n$ that the immersed boundary applies to the fluid. For simple materials, such as fibers, this is a straightforward computation (see [17]). For a detailed description of a shell immersed boundary force computation see [7].
(2) Use (3) to compute the external force on the fluid $\mathbf{F}^n$.
(3) Compute the new fluid velocity $\mathbf{u}^{n+1}$ from the Navier Stokes equations.
(4) Use (4) to compute the new position $\mathbf{X}^{n+1}$ of the immersed material.

We shall now describe in detail the computations in steps 2 — 4, beginning with the Navier-Stokes equations.

The fluid equations are discretized on a rectangular lattice of mesh width h. We will make use of the following difference operators which act on functions defined on this lattice:

$$D_i^+\phi(\mathbf{x}) = \frac{\phi(\mathbf{x}+h\mathbf{e}_i)-\phi(\mathbf{x})}{h} \tag{5}$$

$$D_i^-\phi(\mathbf{x}) = \frac{\phi(\mathbf{x})-\phi(\mathbf{x}-h\mathbf{e}_i)}{h} \tag{6}$$

$$D_i^0\phi(\mathbf{x}) = \frac{\phi(\mathbf{x}+h\mathbf{e}_i)-\phi(\mathbf{x}-h\mathbf{e}_i)}{2h} \tag{7}$$

$$\mathbf{D^0} = (D_1^0, D_2^0, D_3^0) \tag{8}$$

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

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:

$$\rho\left(\frac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\Delta t}+\sum_{k=1}^{3}u_k^nD_k^{\pm}\mathbf{u}^n\right) = -\,\mathbf{D}^0p^{n+1}+\mu\sum_{k=1}^{3}D_k^+D_k^-\mathbf{u}^{n+1}+\mathbf{F}^n \tag{9}$$

$$\mathbf{D}^0\cdot\mathbf{u}^{n+1} = 0 \tag{10}$$

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

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

Equations (9) and (10) are linear constant coefficient difference equations and, therefore, can be solved efficiently with the use of the Fast Fourier Transform algorithm.

We now turn to the discretization of equations (3), (4). Let us assume, for simplicity, that $\Omega \subset \mathbf{R}^2$ 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 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.

$$\mathbf{F}^n(\mathbf{x}) = \sum_{\mathbf{q}\in\mathbf{Q}} \mathbf{f}^n(\mathbf{q})\delta_h(\mathbf{x} - \mathbf{X}^n(\mathbf{q}))\Delta\mathbf{q} \tag{11}$$

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

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

$$\mathbf{X}^{n+1}(\mathbf{q}) = \mathbf{X}^n(\mathbf{q}) + \Delta t \sum_{\mathbf{x}} \mathbf{u}^{n+1}(\mathbf{x})\delta_h(\mathbf{x} - \mathbf{X}^n(\mathbf{q}))h^3\,, \tag{12}$$

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 (11) and (12), is defined as follows:

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

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 [17].

4. The Titanium Programming Language

Titanium 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 [10, 20].

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 addresss space abstraction through which one thread may directly read or write 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 allowing for the 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 instructures to access memory that is associated with another thread. On a distributed memory platform, lightweight communication calls are inserted automatically for pointer dereferences. 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:

- User-defined immutable classes (often called "lightweight" or "value" classes)
- Flexible and efficient multi-dimensional 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 C++ style template.

5. The Data Structures and the Algorithm

Throughout the rest of this paper we assume that we have $p = 2^k$ processors available for our computation. Though our algorithm may be suitable for a more general number of processors, this assumption will simplify our discussion. The solution of the discretized Navier-Stokes equations (9) – (10) is the central and most costly part of the numerical method. In this part we are using the FFTW software, that provides efficient and portable Fast Fourier Transform functions []. The decision to use FFTW influenced the design of our main data structures. Parallel execution of FFTW routines requires that the transform data be 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 $p|N_1$), processor q stores the data slab

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

Accordingly, fluid 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 (9) 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.) This data structure is called `FluidSlab`.

Immersed material grids can be one, two or three-dimensional rectangular arrays of grid points. The complete specification of the material grid and its type depends on the particular application. The `IB` package provides two abstract classes for this purpose. Local properties of the material must be specified in a class derived from `IB.GridPoint`, its global properties – in a 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 material must be specified by defining an appropriate method within the `Shell` class. The `IB` package provides a utility for registering material types and facilities for generation and manipulation of such collections of objects. The shell material type may be created, for example, as follows:

```
Shell shell = new Shell();
ShellPoint shellpoint = new ShellPoint();
```

```
    IB.MaterialType.register("shell", shell, shellpoint);
```

Having defined the required types of immersed material, the `IB` package user can proceed to generate the material grids. Each immersed boundary grid must be local to a domain of some processor and it is the responsibility of the user to ensure proper load balancing by distributing the grids in a uniform way among processors. The computation of the elastic forces in the first phase of each immersed boundary time step is therefore completely local to each processor. Upon the completion of this step it is necessary to spread these forces to the fluid grid, whose data structures are shared among all processors. To accomplish this efficiently it is necessary to achieve both good processor load balancing and to communicate the data among the processors in bulk. On most available systems sending a large number of small messages is very expensive, and it is preferable to send a single large message instead. We therefore introduce an auxiliary data structure, called `CubeList`, which each processor maintains locally.

We imagine the fluid grid being partitioned into a disjoint set of $4 \times 4 \times 4$ cubes, with cube (j_1, j_2, j_3) having its corner at the point $(4j_1, 4j_2, 4j_3)$. When the force of an immersed material point is spread to the fluid, it can affect at most 8 such cubes. First, each processor spreads the immersed boundary forces to the fluid locally, creating a list of such cubes storing the force contribution to the fluid grid from all immersed boundary grids owned by this processor. Once such list has been created, its contents is sent to other processors, which use it to update their local `FluidSlab` structures F_1, F_2 and F_3. At the beginning of the force spreading phase of the time step each processor has an empty list of cubes. For each immersed boundary point the processor determines the corresponding fluid cubes that need to be updated. If such cubes have not been allocated yet by this processor, they are allocated before the update is performed. When the width of the fluid slab is a multiple of 4 (not counting the ghost planes), each fluid cube is contained in only one fluid slab. Each processor now packs its cubes into p disjoint messages to be sent to all processors (including itself). At the end of this communication phase each processor works through all of the messages it has received, unpacking the cubes and accumulating the cube data into F_1, F_2 and F_3.

The final phase of the time step involves computing the new position of the immersed boundary by interpolating the new fluid velocity, obtained by solving the Navier-Stokes equations, from the fluid grid to the immersed boundary grids. This phase is completely analogous to the force spreading phase, except that the cube lists now contain fluid velocity data, and the information is propagating in the opposite direction: from the fluid grid to the immersed material grids. It is interesting to notice that the cubes *received* during the force spreading phase are exactly the cubes that are *sent* during the velocity interpolation phase. Therefore, when processor p_1 received a list of cubes containing force data from processor p_2, it recorded the ids of the received cubes. It now carves these cubes out of the slabs U_1, U_2 and U_3, packs them in a message and sends it to p_2. As before, upon the completion of the communications phase, each processor scans all the immersed boundary points it owns, using the fluid velocity data stored in the `CubeList` it received to update the position of each point.

6. Software Performance

Since the construction of a heart model, or a cochlea model, is a complicated undertaking requiring a significant amount of work, we have constructed a number of simple test models for the purpose of tuning the performance of the IB software. Each such test model consisted 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 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 [18]). The cochlea model consists of a number of surfaces and, assuming the fluid grid 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 number of immersed boundary points. Each of our test models contained a number of identical $N \times N$-point plates. We considered models with the number of plates $n = 1, 16$ and $N/8$, and $N = 256$ and $N = 512$. Accordingly, in the discussion below we refer to a model with n plates of size $N \times N$ as a (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.

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 there are 380 nodes on this computer and each node has between 16 and 64 GBytes 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 1 also contains data from experiments with 1, 2, 4 and 8 processors, all within a single node. Figures 1 and 2 demonstrate the scaling of our software when a varying number of processors is used to compute the same problem. 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 1 summarizes the wall clock per time step results for a number of test models, as well as the total number of floating point operations computed (in billions) when the maximal number of processors is employed.

Finally, Table 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 `CubeList` 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

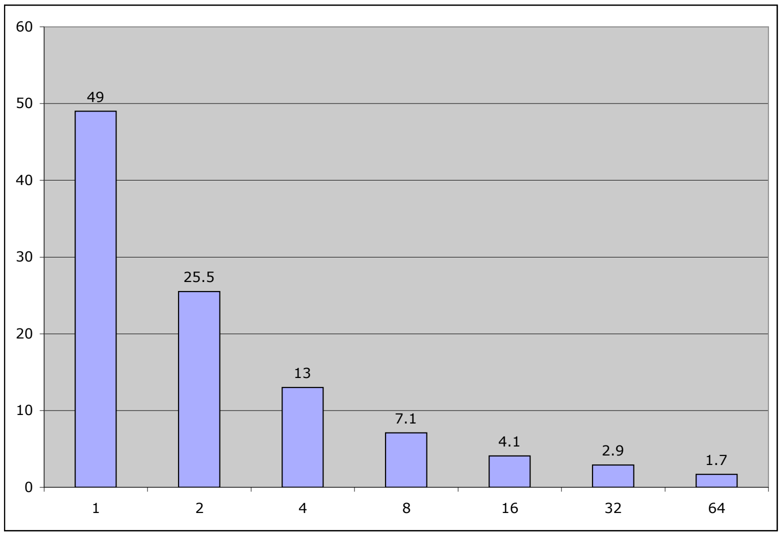

FIGURE 1. Execution wall-clock time (in seconds) per time step for the (256, 1)-model as a function of the number of processors.

TABLE 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

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.

7. SUMMARY AND CONCLUSIONS

We have developed an efficient algorithm for immersed boundary simulations on distributed systems. Using the Titanium programming language we have implemented a software package `IB` based on this algorithm, which provides the package user with a set of versatile classes and utilities to build immersed boundary applications. Our main specific goals are to construct large scale immersed boundary models of the heart and of the cochlea. In this paper we have demonstrated the feasibility of such constructions. Nevertheless, the

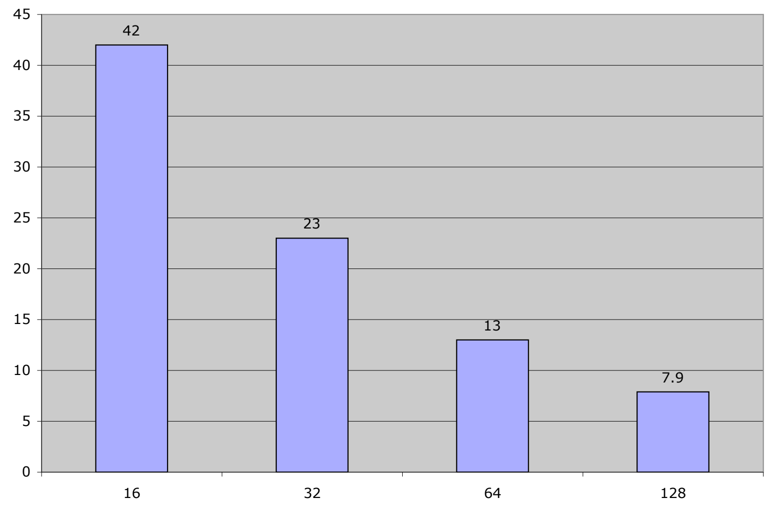

Figure 2. Execution wall-clock time (in seconds) per time step for the (512, 1)-model as a function of the number of processors.

Table 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

performance measurements we report here indicate that numerical experiments with these models will require considerable time. Indeed, a single typical experiment with the cochlea model may require over 10,000 time steps, which in the case of a 512^3-point fluid grid we estimate will take more than 3 days to complete. Simulating a single beat of the heart, on the other hand, requires over 50,000 time steps. Such an experiment can now be completed in less than 3 days in the case of a model based on a 256^3-point fluid grid. A heart model based on a 512^3-point fluid grid is presently too large to be run using the current version of the `IB` package.

We have demonstrated good scaling of our algorithm and currently the main limitation of our software lies in the fact that the number of processors we can employ on $N_1 \times N_2 \times N_3$-fluid based models is at most $N_1/4$. However, it is clear from the description of our algorithm that this limitation can be removed. We are planning to introduce this improvement in the next version of the `IB` package. We are confident this will substantially reduce the total wall clock time of large immersed boundary computations and will make the construction of a 512^3-based heart model feasible. Finally, we are investigating the use of a multigrid-based Navier-Stokes solvers, which would allow for a blocked decomposition of the fluid mesh to provide further flexibility in the processor configuration and possibly further scalability.

References

[1] R. P. Beyer. A computational model of the cochlea using the immersed boundary method. *J. Comp. Phys.*, 98:145–162, 1992.

[2] R. Dillon, L. J. Fauci, and D. Gaver. A microscale model of bacterial swimming, chemotaxis and substrate support. *J. Theor. Biol.*, 177:325–340, 1995.

[3] Rosar M. E. and Peskin C. S. Fluid flow in collapsible elastic tubes: a three-dimensional numerical model. *New York J. Math.*, 7:281–302, 2001.

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

[5] L. J. Fauci and A. L. Fogelson. Truncated newton method and modeling of complex immersed elastic structures. *Comm. Pure Appld. Math.*, 46:787–818, 1993.

[6] L. J. Fauci and C. S. Peskin. A computational model of acquatic animal locomotion. *J. Comp. Phys.*, 77:85–108, 1988.

[7] E. Givelberg. *Modeling Elastic Shells Immersed in Fluid.* PhD thesis, New York University, 1997.

[8] E. Givelberg and J. Bunn. A comprehensive three-dimensional model of the cochlea. *To be published in J. Comput. Phys.*

[9] 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*, November 2001.

[10] 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, 2001.

[11] E. Jung and C. S. Peskin. Two-dimensional simulation of valveless pumping using the immersed boundary method. *SIAM J. Sci. Comput.*, 23:19–45, 2001.

[12] M.-C. Lai and C. S. Peskin. An immersed boundary method with formal second order accuracy and reduced numerical viscosity. *J. Comp. Phys.*, 160:705–719, 2000.

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

[14] 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*, 11:213–236, 1997.

[15] C. S. Peskin. *Flow patterns around heart valves: A digital computer method for solving the equations of motion.* PhD thesis, Albert Einstein College of Medicine, 1972.

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

[17] C. S. Peskin and D. M. McQueen. A general method for the computer simulation of biological systems interacting with fluids. In *Proc. of SEB Symposium on Biological Fluid Dynamics, Leeds, England,*, July 1994.

[18] C. S. Peskin and B. F. Printz. Improved volume conservation in the computation of flows with immersed elastic boundaries. *J. Comp. Phys.*, 105:33–46, 1993.

[19] Siu Man Yau. Experiences in using titanium for simulation of immersed boundary biological systems. Master's Report, May 2002.

[20] 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(11-13), September-November 1998.

[21] L. Zhu and C. S. Peskin. Simulation of a flapping filament in a flowing soap film by the immersed boundary method. *Submitted to J. Comput. Phys.*

Computer Science Division, University of California, Berkeley
E-mail address: `givelber@eecs.berkeley.edu`
E-mail address: `yelick@eecs.berkeley.edu`