Experience in Using Titanium for Simulation of Immersed Boundary Biological Systems

Siu Man Yau
UC Berkeley
smyau@cs.berkeley.edu

Abstract

The Immersed Boundary Method is a numerical method developed by Charles Peskin and David McQueen of the Courant Institute of Mathematical Sciences to simulate a class of biological systems [1]. The method is an effective way to simulate biological systems that can be represented by elastic fibers immersed in an incompressible fluid. Examples of such biological systems include the cochlea [4], small swimming animals [11], and blood flow in the heart [2]. These systems are complex, and detailed simulations require the use of large-scale parallel machines, which have become increasing difficult to program and optimize in recent years. In this paper we describe our experience using a Java-based parallel programming language, Titanium, to implement the immersed boundary method. Titanium extends Java by providing features for high performance, including a static parallelism model within a global address space. Our software, which is publicly available, is the first version of the immersed boundary method in three dimensions to run on distributed memory as well as shared memory machines. In this report, we describe the implementation and some of the techniques used for performance tuning in Titanium. We also present a performance model that can be used to understand and predict the performance of our implementation on current and future machines. Our implementation is designed to be generic, in that the core immersed boundary method is separated from parts of the model that are specific to the heart or other systems, so that application writers can take our generic package and add in their simulation-specific code.

I. Introduction

The Immersed Boundary Method is a numerical method developed by Charles Peskin and David McQueen of the Courant Institute of Mathematical Sciences to simulate a class of biological systems [1]. The method is an effective way to simulate systems that can be represented by elastic fibers immersed in an incompressible fluid. Using this method, McQueen and Peskin developed a simulation of the mammalian heartbeat [2], which has been used in medical research, such as the evaluation of artificial heart valves.

Since this method's development, many research groups have found other uses for the immersed boundary method. For example, George Oster used the immersed boundary method to simulate sea urchin embryo cells [3]; Ed Givelberg and Julian Bunn used it to simulate the cochlea [4]; John Stockie used it to simulate pulp fibers for his PhD thesis [6]; Aaron Fogelson at the University of Utah used it to simulate blood platelet coagulation; Lisa Fauci at Tulane University used it to simulate small swimming animals [11]; Peter Kramer in Rensselaer Polytechnic Institute is adding thermodynamics to the immersed boundary method [26]; and New York University and UC Berkeley plan to use it to develop simulations of insect flight.

However, previous implementation of the simulation software can only be run on shared memory parallel machines, which limits the number processors that their code can scale to. Moreover, each research group that wants to write their own immersed boundary simulation has to replicate the core simulation code from the Courant version. To remedy this situation, we have implemented a generic immersed boundary software package written in Titanium for distributed architectures. Titanium is a Java-like language that supports the Single Program Multiple Data (SPMD) programming model, and is portable to most parallel machines [5]. A research group that wants to write an immersed boundary simulation software can simply take this package and add in their simulation-specific code.

Currently we have written and tuned the generic software package, and have used it to run a contractile torus simulation, which is a scaled-down version of the mammalian heart simulation. The package scales up to 64 processors, and has run on distributed parallel machines such as the Cray T3E [17] and the Millennium cluster [20]. It has also been run on Origin 2000 [19], a shared memory machine. The performance of the generic package is comparable with the original code written in FORTRAN 77. We have also demonstrated generic software's adaptability by using it to run a component of the cochlea simulation [7]. The original mammalian heart simulation is in the works [25].

II. Related work

Charles Peskin and David McQueen originally developed the Immersed Boundary Method [2]. Aaron Fogelson of the University of Utah has developed the Immersed Boundary Interface Software (IBIS), which is a generic package that can be used to run various immersed boundary simulations, but it only runs on single processor machines [9]. Nathaniel Cowen at Courant also wrote a FORTRAN 77 based 3D generic immersed boundary code that can run on the C90, a vector machine [23]. Our software is based on Cowen's code base. Edward Givelberg and Julian Bunn have used the immersed boundary method to simulate the cochlea with impressive performance from the CACR Superdome [24]. However, the Superdome is a shared memory machine and their software cannot be easily adapted to other applications of the method. George Oster, Jun Yang, Steve Steinberg and Katherine Yelick developed a two-dimensional immersed boundary simulation in split-C, which was demonstrated on Fogelson's platelet coagulation problem and on a simulation of epithelial cells in sea urchin embryos, but it is not readily adaptable to three-dimensional problems [10]. They also developed a performance model to understand the performance of his two-dimensional code [10].

III. Immersed Boundary Method Simulation

In the immersed boundary method simulation, the immersed boundary is represented by a collection of fiber points, and the fluid surrounding the boundary is represented by fluid velocity-, force- and pressure-fields defined on a rectangular lattice. The simulation is performed in time steps. At each time step, the fiber point data structure updates its force value to reflect specific forces that happens on the immersed boundary. In the heart and torus, that would be the contraction of the heart muscles, in the cochlea simulation, that would be the pressure of airwaves. Then, the fiber points exert force onto the fluid lattice as a sum of smoothed Dirac Delta functions of fiber forces evaluated at the lattice points. The velocity of each cell of the

fluid lattice is then calculated from these local forces using the Navier-Stokes equation of an incompressible fluid. Finally, the fiber's velocity is calculated from its surrounding fluid's velocity as a sum of smoothed Dirac Delta functions of the fluid velocities evaluated at the fiber points. After that, the fibers points are moved into a new position, based on their velocities. At the next time step, based on the new position of the fiber, the forces on the fiber points can be recalculated, and the whole operation is repeated [1].

Parallelization Strategy

Since the fluid lattice is rectangular, the lattice is easily partitioned by decomposing it into slabs. In fact, since we are using FFTW [8] in part of our Navier-Stokes Solver, we must partition the fluid grid using slab decomposition. Our group is currently investigating the use of multigrid algorithms for the Navier-Stokes solver, which could yield more flexibility on how to partition the fluid grid and improve scalability.

In contrast, we have much more flexibility on how to partition the fiber points. The partitioning strategy is the deciding factor on the amount of communication overhead and load imbalance, and thus the scalability of the application. The following discusses each phase in the immersed boundary simulation, with special emphasis on the implications of parallelizing them for distributed memory architectures for the original mammalian heart simulation.

Fiber Activation

This is the phase in the method that is specific to the heart simulation or other applications. At the end of this phase, each fiber point will carry a force value to be exerted onto the fluid lattice.

In the heart simulation, the fiber points are arranged into sequences called fibers. These fibers represent muscle fibers in the heart. In this phase, the fiber points interact with their neighbors on the same fiber. A timestep-dependent parameter tells the fiber if it should be contracting, and the fiber points in that fiber will look at one of their neighbors to see if they are close enough. If not, the fiber point will calculate an amount of force that pulls it towards that neighbor according to a spring law.

To calculate the force, each fiber point needs to know the coordinates of its neighbors. If its neighboring fiber point is stored on another processor (i.e., the heart fiber was "cut" by a processor boundary), the fiber point will have to send a message to the other processor to get its neighbor's coordinates. In order to minimize communication in this phase, we need to minimize the fiber cuts. On the other hand, the amount of computation is directly proportional to the number of fiber points that a processor owns. Therefore, to achieve load balance, we would like a partitioning strategy that puts the same number of fiber points on each processor. An optimal partitioning strategy for this phase of the computation in isolation would be to spread the fibers evenly across processors, ensuring that each processor has nearly same number of fiber points, but avoiding any division in fibers. Fortunately, the fibers are relatively short — a typical heart simulation might have more than 10,000 fibers with an average of 80 points per fiber, so partitioning whole fibers will still result in partitions that are fairly balanced.

Spread Force

In this phase, each fiber point will update the 4x4x4 fluid cells surrounding it by adding the fluid force that it will exert on them. The amount of force exerted on the fluid cell by a fiber point is calculated as a smoothed Dirac Delta function of the fiber force evaluated at the fluid cell. In particular, any fluid cell that is 4 units away from a fiber point will not experience force from the fiber point.

If a fiber point is spreading its force to a fluid cell that is not on the same processor, it will need to send the force data to the processor that owns the fluid cell. Therefore, the amount of communication in this phase is proportional to the number of fiber points that are not partitioned on the processor that owns the fluid cell it interacts with. To minimize the amount of communication in this phase, we would like to have a partitioning strategy that places all the fiber points on the same processor as its underlying fluid grids. On the other hand, the amount of computation is directly proportional to the number of fiber points that a processor owns. So we would like to place equal number of fiber points on each processor.

Since for most simulations we have seen, the fiber points are clustered around the center, it is usually impossible to achieve both optimal load balance and minimal communication, given the slab partition of the fluid grid. Therefore, depending on the performance characteristics of the machine we are running the simulation on, it is sometimes beneficial to favor one partitioning strategy over another.

Since there can be multiple fiber points living on different processors that need to interact with the same fluid cell, we need to synchronize their updates. We use the processor that owns the fluid cell as a synchronization point. Each processor will “bundle” up the force updates to fluid cells on the same processor into one large data structure, and copy to another processor in a single message. This approach amortizes the communication cost and simplifies the synchronization problem. After all processors have “bundled” up all the force update requests, each processor will look at all other processors for updates to its cells and process them one by one. This is in accordance with the “owner computes” rule: the processor that owns the fiber points computes the forces it will spread, but the processor that owns the fluid cells is responsible for adding those forces onto the cell.

Because Titanium supports bulk remote copy of rectangular sub-arrays, we have opted to send our “bundled” up requests in the form of rectangular sub-array of deltas. Each processor will allocate the smallest rectangular sub-array that contains all the fluid cells that its fiber will touch for each remote processor (the bounding box), and spreads the force on the sub-array. When all processors are finished updating these delta sub-arrays, each processor will do a remote bulk copy on each remote processor for those rectangular sub-arrays of deltas and use those values to update its fluid grid.

Navier-Stokes Solver (NS Solver)

In our implementation of the immersed boundary method, and all others we are aware of, the NS Solver is implemented using a 3d FFT on the fluid grid. We are using FFTW as our FFT kernel, with a kernel written in C code and one written in Titanium that we can fall back on for platforms that do not support FFTW.

In the NS Solver, we first calculate the right-hand side of the NS equation, using nearest-neighbor updates on the fluid force grid. Then we take an FFT of the right-hand side, and find the velocities in Fourier space. Then we take an inverse FFT to get back the velocity grid in normal space.

The NS Solver that has two communication phases: the all-to-all communication phase in the transpose of 3d FFT, and the ghost-cell copy in the nearest-neighbor computation. The all-to-all communication puts pressure on the bisection bandwidth of the machine. Since we are currently using FFTW to handle all the FFT transforms, we must use slab decomposition to partition the fluid grid. The slab decomposition also minimizes the number of ghost cells that need to be copied. This phase does not involve the fibers, so the fiber partition scheme is irrelevant here.

Interpolate Velocity

In this phase, after the velocities have been calculated on the fluid grid, we calculate the velocity on each fiber point by interpolating the velocity on the 4x4x4 fluid grid around the point. The velocity that the fiber point gets is computed as the sum of smoothed Dirac Delta functions of the fluid velocity grid evaluated at the fiber point. This phase has the same communication issue as the spread-force phase. If a fiber point needs to interact with fluid grid that is not on the same processor, it will need to communicate through the network. In fact, the interpolate phase uses the same code base as the spread-force phase. As in the spread-force phase each processor finds out all the fluid grid information that it needs from all other remote processors, and does a remote bulk copy operation on each of them. This amortizes the communication cost of several requests. The common code base also helped with performance tuning the code, as we tried several different approaches, and any partition strategy that works well for spread-force phase will also work well for the interpolate velocity phase.

IV. Titanium

Titanium is a superset of Java. It inherits all of Java's features, except for threads, and instead uses a Single Program Multiple Data (SPMD) model for parallelism. Like Java, parallel processes/threads communicate through a single address space, although Titanium partitions the addresses so that the programmer can control data layout. There are two types of references in Titanium, those that are guaranteed to point to data local data and those that may point to remote data. In addition to Java's features, Titanium has user-defined immutable classes, which are used for implementing small objects without the usual level of indirection, operator overloading, templates, and region-based memory management. The language also adds true multidimensional arrays to Java, and adds a rich calculus of array operators for taking sub-arrays, transposes, and other array level operations. Titanium arrays are not distributed — each array lives on only one processor, and they are aggressively optimized by the compiler [5]. Using the template mechanism and global references, we have built a distributed array library [22].

A Titanium program is compiled by the Titanium compiler into C code with lightweight messaging layers such as Active Messages [13], Shmem [14], LAPI [15], or, for portability, MPI [16] [12]. The compiler currently runs on the Cray T3E [17] using Shmem, SP3 [18] using LAPI, SGI Origin 2000 [19] using POSIX threads, and the Millennium cluster [20] of Intel Pentium

machines with Myrinet, the Meteor cluster [21] of enhanced Redhat Linux boxes and SP3 using MPI. Although we have only run our immersed boundary method on the Millennium cluster, the T3E, and the IBM SP, it is expected to run in any platform that supports Titanium and FFTW. Particular simulations may be limited by the amount of memory or processing power on the machine.

V. Generic Package

The Titanium Generic Immersed Boundary Software (TiGIBS) package provides a framework for application writers to develop their code more efficiently. It provides a high level API for a tuned and optimized library that performs the part of the simulation that is common to all immersed boundary code, while they provide the rest of the code that is specific only to their simulation. The application-specific part consists of the entire activation phase and fiber points partitioning algorithm. It is most easily written in Titanium, although mixing Titanium with C or Fortran is also possible. The interaction phases and the computation fluid dynamics phase are included in the package.

The package utilizes the Java class hierarchy to communicate with the rest of the simulation software. All fiber points are required to extend an abstract "immersed boundary point" class, which contains the fields – force, coordinates, and velocity – that are required to interact with the fluid grid. The fluid grid is represented by a distributed array data structure that is part of Titanium's standard library. An application writer has access to all those fields and the contents of the fluid grid, and uses these fields to communicate with the TiGIBS package.

A typical use of the TiGIBS is as follows. During initialization, the program reads in an input file that describes the dimension of the fluid space, the viscosity of the fluid and other parameters such as the number of timesteps to simulate, or the duration of each timestep. The program creates a TiGIBS object, which takes care of the immersed boundary code. The program then either reads from a file, or calls some subroutine that would generate fiber points and register them to the TiGIBS object. These fiber points must extend the abstract "immersed boundary point" class in order to register successfully. Then the program can proceed through a number of simulation timesteps. At each of these timesteps, the program calculates the force carried by each fiber points and updates the fields in each fiber points to reflect that change. Then the program can simply call the TiGIBS object to advance one timestep, after which the TiGIBS object will have spread these forces onto the fluid, solved the NS equation for the fluid velocity in the next timestep, and interpolated the velocity back to the fiber points. So the coordinates and force values on the fiber points can be seen as the "input parameters" to the TiGIBS simulation call, and the velocity vector for those points is the "output parameter" for that call. Typically the program updates the coordinates of each fiber point using their velocities. After a certain number of timesteps, the program also reads the fluid grid and dump out the velocity field and pressure fields to an output file, and also the velocity and coordinates of each fiber points to an output file for analysis and visualization. See the appendix for detailed description of the interface.

This approach allows more flexibility – but more work – on the application writers' part when they are writing the simulation. In particular, the fiber points' partitioning scheme - one of the most important factors that affect the performance - is not determined by the package. One could envision a less general immersed boundary packages that can be developed on top of this generic

package and tailored to a specific kind of simulation. For example, in the heart and torus simulations, the codes for reading the input files, saving output files, and activating the fiber points are almost identical. A package can be written on top of the TiGIBS that is tuned for simulations like the heart and torus – simulations whose fiber points are arranged into1D meshes and whose activations only depend on their nearest neighbor. However, this code would be useless for other simulations such as the cochlea, which arranges its fiber points into a 2D mesh. Thus we decided that this code should be a separate package, and not included in TiGIBS.

Using the TiGIBS, we have written a contractile torus simulation and a component of a cochlea simulation [7]. The original mammalian heart simulation is in the works [25].

VI. Performance Model

To help understand the performance characteristics of the heart code, we created a performance model using the available data. Our goal is to have a performance model to estimate the amount of time needed to run a single timestep given information about a specific run of the simulation. For performance reasons, two of the computation kernels in the package are written in C; the following performance numbers are not that of pure Titanium code.

Methodology

Initially, we aimed for the model to take raw data, i.e., simulation parameters such as number of fiber points and fluid grid size, and machine parameters such as the peak FLOP rate, cache size, network latency and bandwidth – and have the model generate a prediction of the run time. This method was found to be too complicated. For instance, the communication time during the interaction phase depends on how the fiber points are partitioned, which can be different from model to model. Cache behavior is also affected by the way the fiber points are partitioned and the amount of work done during the fiber activation phase, which is application-specific. Therefore we turned to a more high-level model.

Instead of using low-level parameters such as FLOP rate and network latency, we used higher-level parameters to predict the run time. The parameters used are derived from experiments. These parameters are listed in the following tables. Application writers need to do small runs of parts of their code on the machine to get these parameters. Then he can use the model to estimate the run times of the larger run.

Machine-specific Parameters	**Symbol**	**Meaning**	**Value on the T3E**
Cost of a single fiber point activation	K_{act}	The time it takes to complete activation for one fiber point	0.045 ms / fiber point
Cost of remotely accessing a fiber point	L_{act}?	The time it takes to read the coordinates of a fiber point from a remote processor. (Equivalent to de-referencing 3 Titanium remote pointers).	0.05125 ms / fiber point

Cost of a single fiber point update	K_{int}?	The time it takes to spread a fiber's force to its surrounding fluid (or interpolate surrounding fluid's velocity to a fiber)	0.11 ms / fiber point
Remote array copy latency	L_{arr}?	Latency of sending the fluid delta grids to/from remote processors. (Equivalent to the latency of 3 remote Titanium array copies).	0 ms (Due to the low latency of T3E, for the amount of data we are transmitting, the message overhead is negligible.)
Titanium array copy Bandwidth	B_{ti}?	Remote Titanium array copy bandwidth	0.01 ms / fluid cell
Cost of transferring a ghost cell	B_{ghs}?	Bandwidth of transferring 3 Titanium sliced arrays.	0.01587 ms / ghost cell
Cost of calculating RHS per grid cell	K_{rhs}?	The time it takes to do a nearest-neighbor computation to solve for the RHS of the NS equation per fluid cell	0.00946 ms / fluid cell
Number of processors	? ?	The number of processors used to run the simulation	N/A
Cost of serial 1D FFT	K_{fft}?	The constant in FFT	0.05ns
Transpose cost	B_{tpo}?	The Bandwidth of FFTW to do a 3d transpose	3.2ns per fluid grid cell
Constant of Fourier space calculation	K_{fsp}?	The time it takes to calculate the solution of NS equation in the Fourier Space per fluid grid cell	9ns / fluid cell
Transpose latency	L_{tpo}?	The fixed overhead of sending one transpose inside FFTW	0 (The latency was found to be negligible on the T3E due to the low latency)
Application specific parameters	**Symbol**	**Meaning**	**Value for the heart simulation on 64 processors**
Max. no. of fiber points on a single processor	C	The number of fiber points on the processor that has the largest number of fiber points allocated to it.	10254
Max. no. of processor	X	Each processor can have a number of fibers	5528

boundary-crossing fibers on a single processor		crossing processor boundaries. This term is the max. of that number over all processors.	
Size of bounding boxes of fibers owned by processor i on remote processor j	$B_{i,j}$	The size of the bounding box that can hold the group fiber points which are owned by processor i, but interact with part of the fluid grid on processor j.	17228 (this is the $Max._i (??_j B_{i,j})$ term over all processors i, j
Processor Boundary surface area	A	The area of the surface in the fluid grid between parts owned by different processors	128^2
One side of fluid grid	n	Size of the fluid grid	128

Table 1: Summary of parameters used in the performance model

i) Fiber Activation

In the fiber activation phase, the run time is limited by the speed at which the processor that owns the largest number of fibers can update its fibers' force values. The speed at which that processor can update its fibers' force values is in turn determined by the time to compute and communicate, i.e., the number of fiber points on this processor, and the number of fiber points that need to go across processor boundary to get the information from its neighbor. There are thus two terms in the performance model for fiber activation:

$$C * K_{act} + X * L_{act}$$

From the expression, it is evident that a well-partitioned fiber point set is important to efficiency: the points must be balanced so that the C term (max. number of fiber points on a single processor) will not be too large, and there should not be too many fiber running across processor boundaries (minimizing the X term).

In the heart simulation, the fiber activation consists of a relatively minor portion of the total run time.

ii) Interaction phases

In the interaction phases, the code is divided into two parts: there is a computation part where the code spreads force to or interpolate velocity from a buffer that contains the corresponding fluid grid; and there is a communication part where the fluid in this buffer is copied to the fluid grid using the Titanium's remote array bulk copy method. The computation part is determined by the speed at which the processor that owns the most number of fibers can compute the force being spread or the velocity being interpolated. This in turn is determined by the number of fiber points in the processor. For the communication part the runtime is determined by the amount of fluid grid that a processor needs to send to interact with that is owned by another processor. So there are also two terms for this part of the performance model:

K_{int} * C + (L_{arr} * N + B_{ti} * Max.$_i$ ($\sum_j B_{i,j}$)) *over all processors i, j*

From the expression, it is clear that in addition to having load balanced with little fiber cuts by processor boundaries, we would want a partition that where most of the fiber points are owned by the same processor that owns its underlying grid, so the term ($\sum_j B_{i,j}$) can be minimized.

iii) Navier-Stokes Solver

There are three parts to the NS Solver: the first part is the calculation of the right-hand side of the NS equation. This involves nearest-neighbor computation on the fluid grid, which requires communication across the processor boundaries of the fluid grid. The second part is the FFT and inverse FFT, which are done by FFTW. The last part is the embarrassingly parallel solve inside Fourier Space.

In the first part, there is a communication across processor boundaries, which we do by copying ghost cells. The amount of communication is determined by the size of the boundary surface, while the amount of computation is determined by the number of fluid grid points per processor:

B_{ghs} * A + K_{rhs} * n^3 / N

The 3D FFT is determined by FFTW, which we can model with O(nLogn) model. Users can find these constants by running the FFTW applications alone.

7 * K_{fft} * (n Log (n)) * $3n^2$ / N + n^3 * B_{tpo} / N + L_{tpo} * N

The constant "7" indicates that there are 7 FFTs in the NS Solver: 3 forward 3D FFTs on the force grid, 3 inverse 3D FFTs on the velocity grid, and 1 inverse 3D FFT on the pressure grid. The $3n^2$ term indicates that the 1D FFTs need to be done $3n^2$ times to make up one 3D FFT. We divide the whole term with the number processors since the 1D FFTs are spread evenly on all processors. The communication term assumes that the bisection bandwidth is not saturated, so the time needed to communicate is proportional to the amount of data each processor needs to send, plus the latency of each message, which is proportional to the number of messages it sends (i.e., the number of processors).

In the last part, there is no communication involved. The runtime is a linear function of the number of fluid grids cells per processor:

K_{fsp} * n^3 / N

The NS Solver's run time is the sum of these terms.

Validation

We measured the model against the actual results of the heart simulation on the T3E. As is shown in the figure, the model falls within 10-20% of the actual run time.

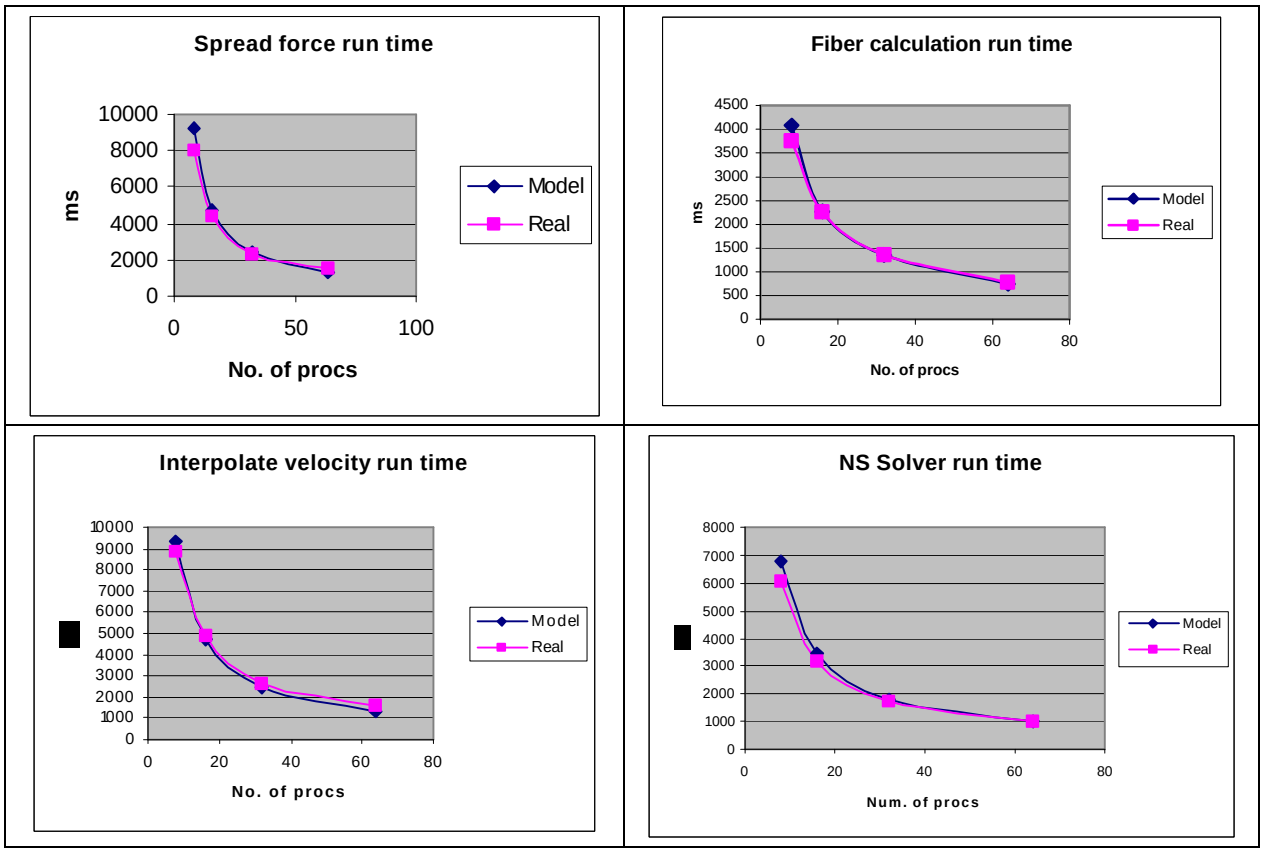


Figure 1: Predicted vs. actual run time of the heart simulation on the T3E

Although the performance model was not 100% accurate, it was instrumental in helping us find a partitioning strategy for the heart simulation.

VII. Performance tuning in Titanium

The performance of a Titanium program is difficult to predict. Since Titanium code is compiled first into C code before being compiled into machine code, the performance of a Titanium program depends on how the Titanium compiler and optimizer generate C code, and how the C compiler and optimizer generate machine code. To help improve the performance of the Titanium immersed boundary method package, we have rewritten some of the computational kernels in C, experimented with different fiber point partition strategies and tried using a new sparse array copy construct in Titanium to cut down communication overhead. Most of the performance analysis and tuning work was done on the Cray T3E, with analysis of sparse array copy done on the Millennium.

Native code

Titanium has a powerful array abstraction built into the language and optimized extensively by the compiler. In particular, the foreach construct allows efficient accesses to arrays to be written conveniently. However, Titanium is not tuned for unpredictable or non-uniform accesses to arrays, since in those cases, foreach loops cannot be used. Instead, we must rely on the creation of Titanium points, which are tuples of integers, and use these Titanium points to calculate an

index into the array. Such calculations are potentially time consuming, since Titanium's runtime library has to support arrays that have special layouts, such as a strided array, or the projection, translation, or slice of other arrays. There are two places in the immersed boundary code that requires such non-uniform and unpredictable access patterns. In those cases, in order to get better performance, we are forced to write the computation kernel in C and use Titanium's native C interface to link it into the rest of the Titanium code.

In the interaction phases, we need to access the parts of the fluid grid that interact with the fiber points – either writing to the force grid in the spread-force phase, or reading the velocity from it in the interpolate velocity phase. The parts of the grid that we need to access are the grid cells that are next to where the fiber points are. As the positions of the fiber points are determined by the input, and as the fiber points move to different coordinates during the simulation, this access pattern is unpredictable. The Titanium method to access the grid – by creating a point and using the point to calculate the index into the array - proved to be too slow. Therefore we have decided to move interaction phase's computation kernel into native C code which understands the fluid grids' structure and by-pass some code to support special Titanium arrays. After we moved the interaction phase into C code, we observed a 33% speedup. We think this kind of speed up is due to assumptions about the array layout (e.g., whether the array is 1-strided, 0-based and is not derived from operations such as slice, inject, permute, and restrict) that the native code can make but the Titanium runtime array implementation does not.

The FFT kernel's butterfly access pattern is a non-uniform access pattern to the fluid grid, and as a result, the pure Titanium version is quite slow as well. We obtained about 4 times speedup from moving the code into C. However, the strided memory access patterns in a straightforward FFT implementation still perform poorly on modern memory hierarchies. Packages like FFTW optimize for these hierarchies by using recursive decompositions of the FFT to improve locality and they also optimize for other architectural features through an automatic tuning process. Another advantage of moving the FFT code into C is that it gives us the option of using pre-tuned FFT packages such as FFTW. On the T3E, we linked the FFTW library into the native C code and use FFTW to take care of our FFT transformations; the calculations in Fourier space are done in C. Making this change gave us another 4 times speed up over the Navier-Stokes solver based on C code.

NS Solver based on:	Pure Titanium	Native C	FFTW
Run time (in ms)	27737	7003	1730

Thus, native interface solves two of Titanium's weaknesses: the performance of non-uniform and unpredictable array access and the ability to interface into pre-tuned software packages. However, the part written in C will not be type checked by the Titanium type checker and is thus unsafe. Moreover, the part written in C may not be portable to other platforms. Therefore, the use of native C code in Titanium should be restrained only to cases where there is an obvious benefit.

Partitioning Strategies

As discussed above, the fiber point partitioning strategy is a critical piece in determining the speed of a simulation. We experimented with different partitioning strategies and eventually used

the performance model guided us to determine a good strategy on the T3E to run the torus and heart simulations.

The most intuitive partitioning strategy is to allocate fiber points to the same processor that owns its fluid grid. But since the fiber points of the heart are located around the center, there is serious load imbalance problem, with the C term (Max no. of points per processor) unacceptably high. In fact, in the 8-processor partitions of the heart and torus using this strategy, the two fringe processors do not get any fiber points at all.

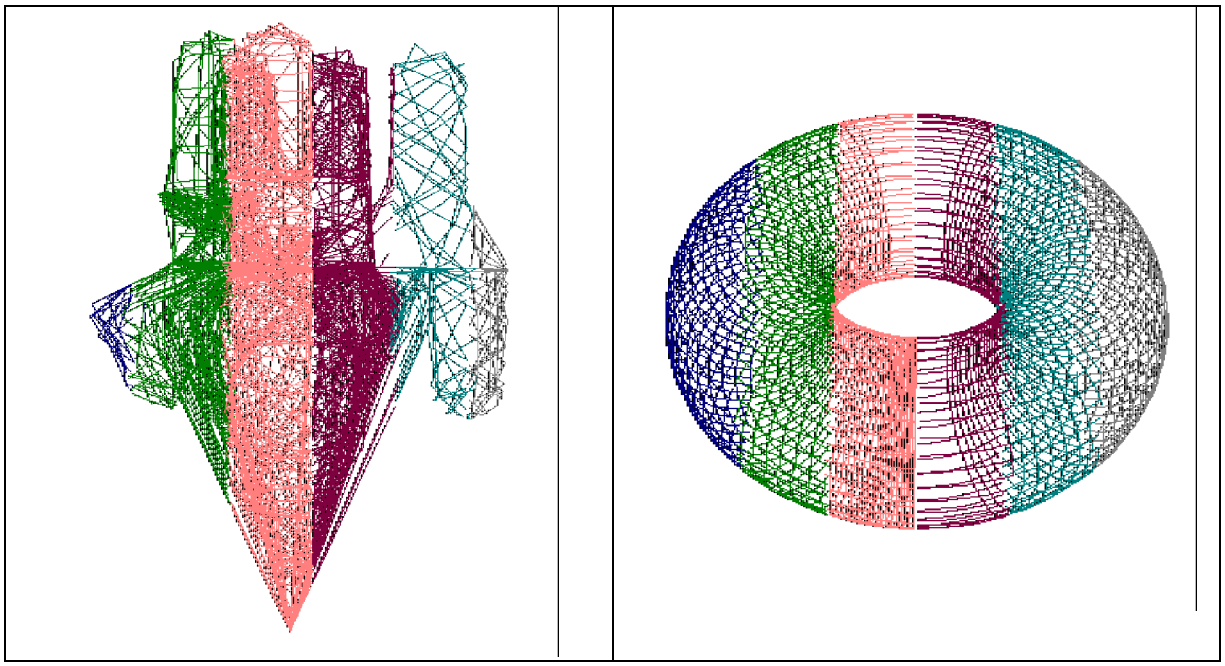


Figure 2: A heart and torus partitioned to maximize locality. Each different color represent a fiber allocated to different processors. These are 8 processor partitions, but only six processors have fibers on them in either case.

We tried some heuristics to load balance the torus. For example, using a "pizza-cutter" strategy to partition the torus gave us better performance numbers [7]. In that strategy, we partition the torus as though we are lying the torus down on a table and cutting it like a pizza. However, non-symmetric models such as the heart do not readily lend themselves to these heuristics.

The more general partitioning strategy is to maximize the locality of the fiber points, while retaining load balance. In this strategy, each processor is allowed a maximum number of fiber points, typically the average number of fiber points per processor. First, each processor gets as many fiber points that interacts with its fluid cells as possible, but not more than the limit. After that, some processors will have not enough points, and some will have points left over. The processors that do not have enough points will then ask for processors that have leftovers for points. Eventually all processors will have the same number of points. While this strategy reduces the C parameter (Max. no. of fiber points on a single processor) and B_{ij} parameter (Size of bounding boxes on remote processors), it makes no attempt to lower the X parameter (Max. no. of processor boundary-crossing fibers on any single processor), unlike the pizza cutter.

Luckily, due to the low latency on the T3E (?), communications due to fiber cutting processor boundaries is only a minor percentage of the total run time.

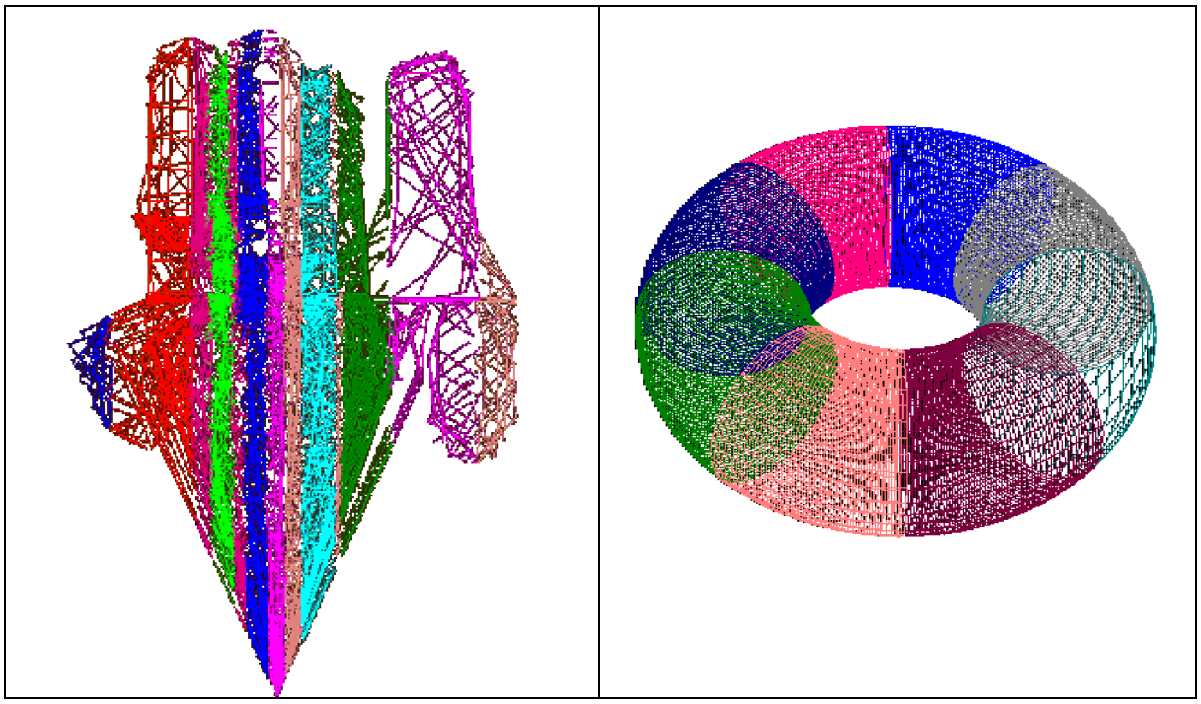
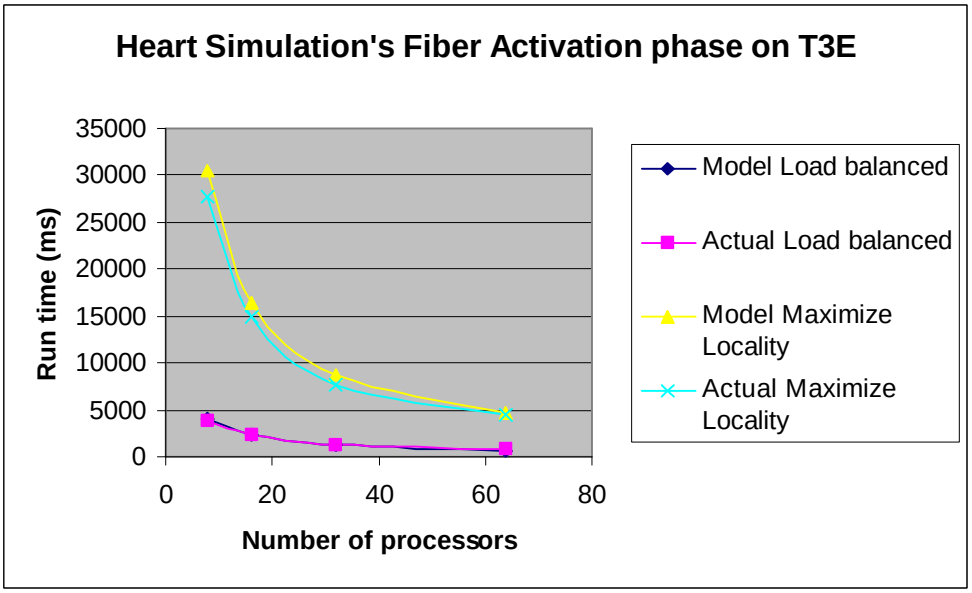


Figure 3: Heart and torus partitioned with load-balanced strategies

The run times of these two partition strategies for the heart are shown below:



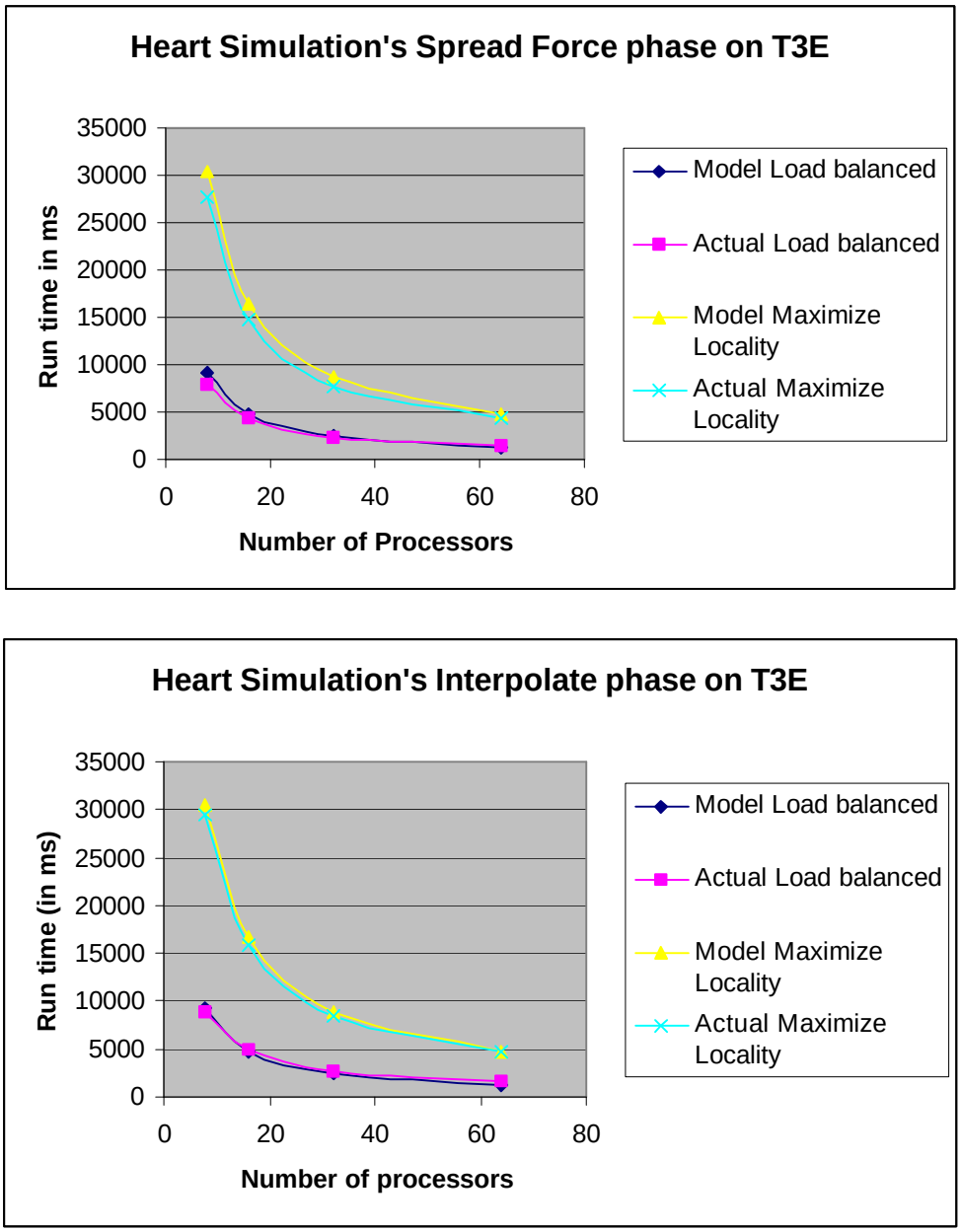


Figure 4: Run time graphs for the two partitioning strategies on the T3E

We see that the new partitioning scheme is several times more efficient than the naïve maximized-locality scheme. We notice that in the fiber activation phase, the new partitioning scheme is consistently about 6 times faster than the naïve scheme; while in the interaction phases, the new partitioning scheme is about 3 times faster. This is because the fiber points tend to aggregate around the center processors, making the processor in the center to do most of the work in the naïve scheme. The number of fibers in the most "worked" processor is about 6 times more in the naïve scheme, so we see a 6 times difference in fiber activation. But the advantage gained by having a more balanced load is slightly offset by needing to copy the bounding boxes during the interaction phases, so we only see a 3 times difference there. It is also important to point out that since we are doing the experiments on the T3E, where the network latency is

relatively low compared to CPU speed, so the communication inside the fiber activation phase is only a minor percentage of the total run time (6.2% on 64 processors). If we move to a machine with higher network latency, the communication inside the fiber activation phase will become significant. We have seen that millennium actually slows down in the fiber activation phase when we move from 16 processors to 32, since the fibers cut by processor boundaries have increased, and communication in fiber activation became a dominant factor.

As is evident from the discussion, the optimal partitioning strategy depends on the machine architecture, in particular, the CPU speed and network latency and bandwidth. However, the partitioning strategy is designed to be outside of the generic immersed boundary package. This is to allow application writers to take into account factors not visible to the TiGIBS package, such as the communication and computation trade offs in the fibers activation phase, while designing their own partitioning strategies. For applications where the fiber activation phase is not a performance concern, we hope to write a tool that will help application writers to find a near-optimally scalable partitioning strategy given information about a machine's CPU and network.

For our heart simulation, we achieved a 5.5 times speed up from 8 to 64 processors (the single processor speed cannot be measured as the application is too big to fit on one processor). The speed up is limited by two factors: 1) most fiber points are aggregated around the center and it is necessary to communicate the updates to the processor that owns the part of the fluid where the fibers are aggregated; and 2) the number of fiber cutting across processor boundaries is large enough to limit the speed up of the fiber activation part to about 5 times.

Sparse array copy

We noticed that in the interaction phases, although we are sending whole bounding boxes by Titanium's bulk array copy across processors, we are not using all data within them. This led us to investigate the use of a new construct in Titanium array, the sparse array copy. Sparse array copy allows a Titanium program to copy a non-rectangular array remotely. Ideally, we would specify a non-rectangular array with only the fluid grid points that a processor need, and copy just that array, that would make the message size smaller and reduce communication overhead.

Jason Duell, Wren Montgomery and Simon Yau investigated the use of sparse array copy on the interaction phase of the model [7]. They found that using the sparse array copy has a drawback: they need to construct the non-rectangular array to be used in the sparse copy. Since there can be more than one fiber point in a fluid grid cell, and each fiber point interacts with a 4x4x4 fluid sub-grid around it, a fluid grid can potentially need to interact with more than one fiber point. In fact, since the fiber points tend to aggregate near the center of the fluid grid, fluid grid cells normally interact with more than one fiber point. Making sure that a fluid grid cell is added to the non-rectangular array at most once is an issue. In effect, the algorithm that constructs the non-rectangular array needs to keep track of each fluid grid cell in the experiment and make sure that it doesn't add the same grid cell into the non-rectangular array twice. In fact, this accounting overhead is so large that the speed gained by reducing network traffic is offset by this overhead, and sparse array copy does not work.

Figure 5 shows the breakdown of the communication cost in the spread-force phase of a torus simulation on the Millennium. "Original" and "megabox" versions use the bulk array copy, "hash" and "boolean grid" versions use array copy, where they use a hash-table based and

Boolean grid based accounting mechanism respectively. As seen from the graph, the set-up cost of the two sparse array copy versions caused the communication cost of the sparse copy version to be several times greater than the bulk copy version.

However, their work also suggested that in a simulation where less than 10% of the bounding box is filled (the torus has 60% filled), the sparse array copy would pay off.

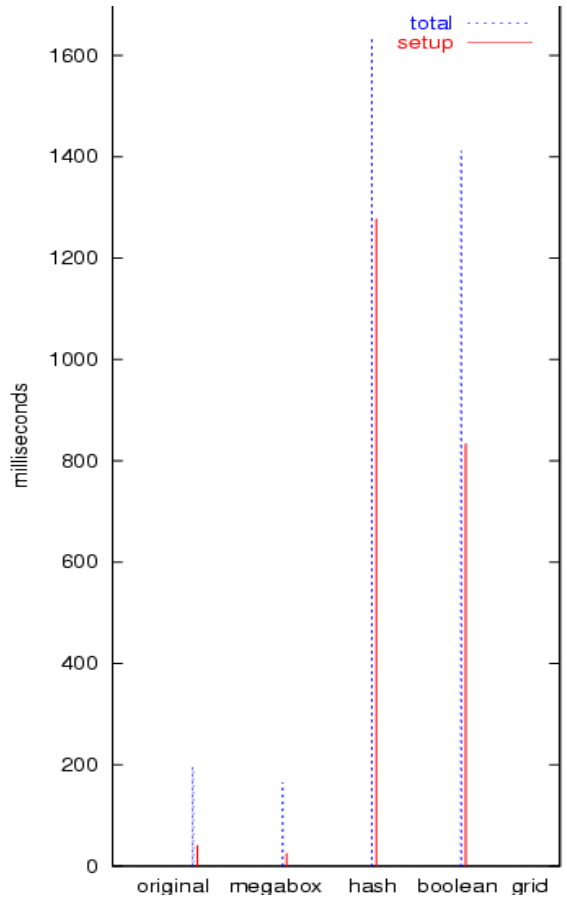


Figure 5. The communication cost of using sparse array copy.

(Source: Duell, Montgomery, Yau [7])

We included the native codes in interaction phase and in the NS solver, including the code that calls FFTW, in the TiGIBS library, but not the sparse array copy.

VIII. TiGIBS Adaptability

To demonstrate TiGIBS's adaptability, we have written three different simulations using the TiGIBS.

Torus Simulation

The torus simulation was written by Nathaniel Cowen from Courant Institute of Mathematics at New York University. It is a scaled-down version of the heart. TiGIBS was originally based on

this code. There are 65,000 fiber points in this model and a 64x64x64 fluid grid. Early performance tuning effort was done on this model on SGI Origin 2000 and the now cluster.

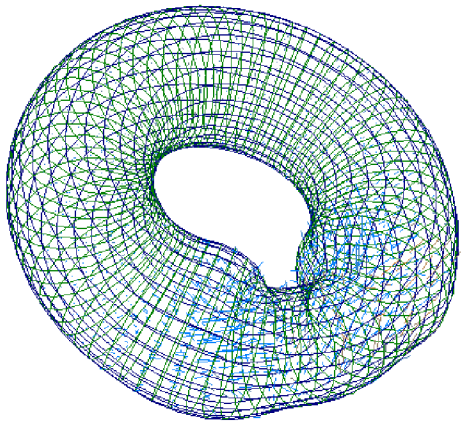


Figure 6. Contracting Torus

Heart Simulation/Experiment 800

The heart simulation is the model developed by Dave McQueen and Charles Peskin at NYU [1] [2]. This model can be used in medical research such as evaluation of artificial heart valves. Most of the performance tuning of the TiGIBS has been done on this model on either the T3E or Millennium.

Cochlea Simulation

Juliann Bunn and Ed Givelberg from Cal Tech and University of Michigan used Immersed Boundary Method to simulate the cochlea [4]. Their code was based on C++ and was written using a different code base from the heart or torus. For instance, their fiber points are not arranged into fibers, but a two dimensional mesh. Jason Duell, Wren Montgomery and Simon Yau adapted the TiGIBS package to run part of the simulation (the oval window) [7]. The TiGIBS framework proved to be adequate for the purposes of simulating the oval window.

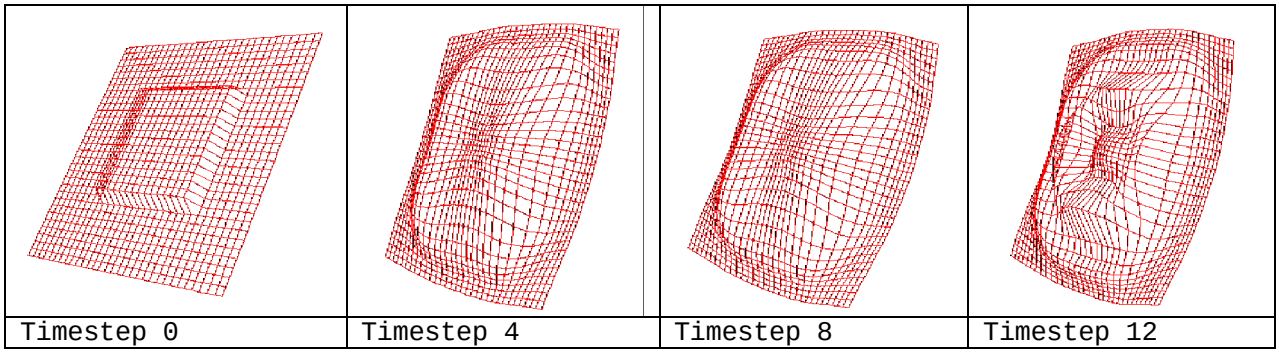


Figure 7. Oval window simulation

IX. Evaluation

Depending on the partitioning strategy and machine involved, the TiGIBS library scales reasonably well. We see a 5.5 times speed up from 8 to 64 processors on the T3E. Unfortunately the speed up is dependent on the fiber partitioning strategy and the latency and bandwidth of the machine on which the simulation is run.

For the runs on the T3E, there is no obvious performance bottleneck – the run time of all four phases are roughly the same:

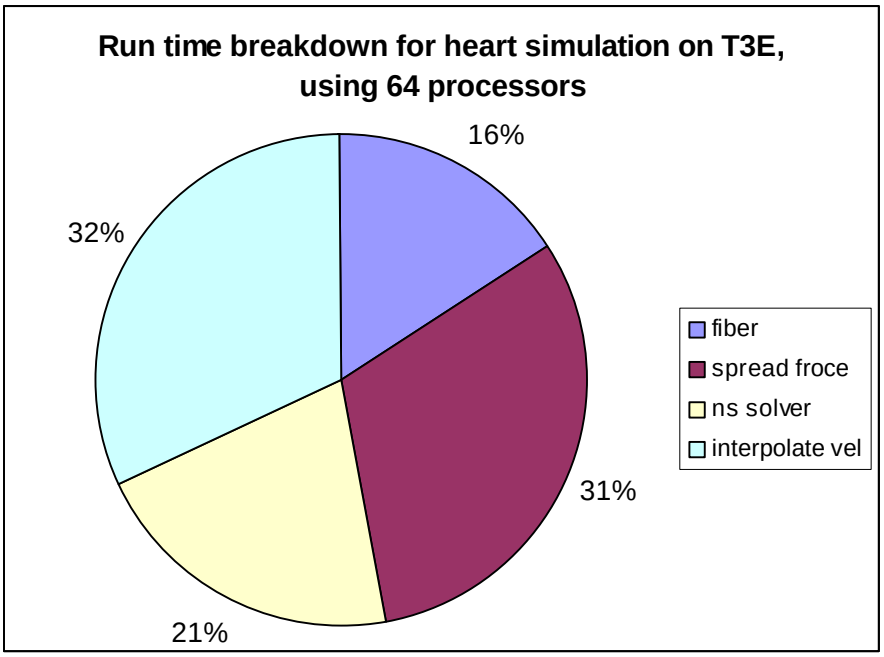


Figure 8. Run time breakdown of Heart Simulation on T3E

The interaction phases each account for about 1/3 of the run time, as the NS Solver account for 1/5 and fiber calculation account for the rest.

X. Future Work

Although we have demonstrated TiGIBS to be an adaptable package with reasonable performance, there still is room for future work.

By using FFT based NS solver, we limit our fluid partitioning scheme to slab-decomposition. A multigrid-based solver would most likely perform best with the fluid grid partitioned into cubes, rather than slabs, and may allow for better simultaneous load balancing and locality for the fiber distribution. An even better option appears to be an adaptive mesh, which would place finer grids where there is most of the fluid activity. The fine fluid grids would likely exist in the part of physical space as the fibers, so the fiber points and fluid mesh could use the same partition and still exhibit good load balance. The multigrid solver also scales linearly with the problem size, rather than the O(n log n) scaling of the FFT, although our performance model indicates that the FFT is still quite efficient for the problem sizes considered.

The TiGIBS is very generic. As a result, the application writers need to do more work to adapt it to their own simulations. We can have sub-packages that can handle some subsets of the immersed boundary simulations. For example, a sub-package that handles collection of 1D meshes will be useful for the heart and contractile torus simulation; while a sub-package that handles collection of 2D meshes will be useful for the cochlea simulation.

Since the fiber partition scheme is a major factor that affects the performance of the TiGIBS library, it would be helpful to have an automatic tool that helps application writers partition their fibers. Such a tool should be able to take in the network and computational characteristics of a machine, a description of the fiber points, and automatically generate an output specifying which fiber point goes to which processor.

XI. Conclusion

We have shown that it is possible to create a generic library in Titanium to enable application writers to write immersed boundary code for distributed platforms with minimal effort. We have adapted the mammalian heart simulation to the distributed platform using the library, which is the first such implementation of this heart model. In addition, we have demonstrated the flexibility of our software by using it for part of a cochlea model and a simple synthetic torus.

We also developed a performance model of the heart simulation, which can be used to estimate the scalability of our software on other machines. The model requires as inputs the latency and bandwidth of communication, as observed by Titanium applications, and the cost of floating point. The floating-point estimate is best obtains by running serial code on one processor of the machine of interest, as performance varies widely between phases of the computation and is not a simple function of the peak floating point rate.

Our software has been extensively tuned, speeding up by more than 10x since our initial implementation. Some of this tuning is algorithmic, having to do with the partitioning of fibers and data across processors, which would be an issue in any parallel programming system. Our experience also highlights some of the performance issues of Titanium, such as the high overhead of array abstraction, when it is not used with irregular or strided memory accesses. While our current implementation addressed this by using native C code for some key kernels, in the long term, we believe this could be addressed by provided a less general array abstraction in Titanium or by a more highly optimized compiler and runtime that takes advantage of the common special cases that occur in our application.

XII. References:

[1] Charles S. Peskin and David M. McQueen. **A general method for the computer simulation of biological systems interacting with fluids**. Biological Fluid Dynamics, ed.1995 **http://www.math.nyu.edu/~mcqueen/Public/papers/seb/SEB_19971216/SEB_19971216.html**

[2] McQueen, D.M., and C.S. Peskin. 1997. **Shared-memory parallel vector implementation of the immersed boundary method for the computation of blood flow in the beating**

mammalian heart. Journal of Supercomputing 11(3): 213-236.
http://www.math.nyu.edu/~mcqueen/Public/papers/psc/PSC_19971216/PSC.19971216.html
[3] C. Peskin and G. Oster. **Coordinated hydrolysis explains the mechanical behavior of kinesin**. Biophys. J. 68:202s-210s
[4] E. Givelberg, J. Bunn. **Detailed Simulation of the Cochlea: Recent Progress Using Large Shared Memory Parallel Computers.** CACR Technical Report CACR-190, July 2001
http://www.cacr.caltech.edu/Publications/techpubs/cacr.190.pdf
[5] Yelick, K., L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. 1998. **Titanium: A High-Performance Java Dialect**. Concurrency: Practice and Experience, September-November 1998:825–36.
[6] Stockie, J. "**Analysis and Computation of Immersed Boundaries, with application to pulp fibers**", Univ. of British Columbia, Ph. D. Thesis, Sept 1997
[7] Duell, J., Montgomery, W., Yau, S. 2001. **Cochlea Simulation using Titanium Generic Immersed Boundary Software (TiGIBS)**. CS267 Fa01 Class Project Report
http://www.cs.berkeley.edu/~smyau/classes/fa01/cs267/Report.doc
[8] FFTW homepage: **http://www.fftw.org**
[9] Software for Immersed Boundary & Interface Simulation homepage:
http://www.math.utah.edu/IBIS/
[10] Steinberg, S. G., Yang, J., Yelick, K. **Performance Modeling and Composition: A Case Study in Cell Simulation. http://www-db.stanford.edu/~junyang/papers/syy-ipps96.ps**
[11] M. Hopkins and L. Fauci, 2002. **A computational model of the collective fluid dynamics of motile microorganisms**. J. Fluid Mech.,455, p. 149-174.
[12] Dan Bonachea, AMMPI: **http://www.cs.berkeley.edu/~bonachea/ammpi/index.html**
[13] Thorsen von Eicken, David Culler, Seth Copen Goldstein, Klaus Erick Schauser: **Active Messages: a Mechanism for Integrated Communication and Computation**. Proceedings of the 19th International Symposium on Computer Architecture, ACM Press, May 1992
[14] T3E Shmem: **http://www.sdsc.edu/SDSCwire/v3.15/shmem_07_30_97.html**
[15] Gautam Shah, Jarek Nieplocha, Jamshed Mirza, Chulho Kim, Robert Harrison, Rama K. Govindaraju, Kevin Gildea, Paul DiNicola, Carl Bender: **Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP**. IPPS'98
[16] Message Passing Interface: **http://www.mpi-forum.org**
[17] Cray T3E homepage: **http://www.cray.com/products/systems/t3e/**
[18] BlueHorizon users guide: **http://www.npaci.edu/BlueHorizon**
[19] SGI Origin 2000: **http://www.sgi.com/origin/2000**
[20] UC Berkeley Millennium Cluster: **http://www.millennium.berkeley.edu**
[21] Rock cluster at SDSC: **http://rocks.npaci.edu/papers/rocks-documentation/book1.html**
[22] Titanium distributed array library **http://www.cs.berkeley.edu/~smyau/distArray**
[23] Cray C90 at Pittsburg: **http://www.psc.edu/machines/cray/c90/c90desc.html**
[24] CACR HP Superdome: **http://www.cacr.caltech.edu/News/superdome0900/**
[25] Titanium Heart simulation: **http://www.cs.berkeley.edu/~bina/Heart/**
[26] Peter R. Kramer, Andrew J. Majda. **Stochastic Mode Elimination Applied To Simulation Methods For Complex Microfluid Systems. http://www.rpi.edu/~kramep/Public/mtv.pdf**

Appendix: Titanium Generic Immersed Boundary Software API

```
package tigibs;

/* Class to represent a fiber point. All fiber points in TiGIBS must extend
this class */
public abstract class IbPoint {
  // id number.
  int id;

  // the Eulerian coordinates.
  public double x_coord;
  public double y_coord;
  public double z_coord;

  // velocity of this point.
  public double x_vel;
  public double y_vel;
  public double z_vel;

  // the force this point is spreading onto the fluid.
  public double force_pt1;
  public double force_pt2;
  public double force_pt3;

  // constructor
  public IbPoint (int pid, double x, double y, double z);
  // field extractors
  public inline int getId ();
  public inline double getXCoord ();
  public inline double getYCoord ();
  public inline double getZCoord ();
  // zeros out the velocity
  public void zeroOutVel ();
}

/* TiGIBS simulation object. */
public class Tigibs {
  // Register a fiber point in the object */
  public void registerPoint (IbPoint p);
  // Register a marker in the object (markers are points, but don't spread
force
  public void registerMarker (IbPoint p);
  // Performs the spread force, NS Solver, and interpolate velocity steps
  public single void advanceOneIteration ();
  // remove all points from the object
  public single void unregisterAllPoints ();
  // constructor
  public single IbMain (int single numOfCellsX, int single numOfCellsY, int
single numOfCellsZ);
  // field extractor of the fluids
  public single DistArrayDouble3d getU (); // dx/dt
  public single DistArrayDouble3d getV (); // dy/dt
  public single DistArrayDouble3d getW (); // dz/dt
```

```
  public single DistArrayDouble3d getF1 (); // force vector
  public single DistArrayDouble3d getF2 ();
  public single DistArrayDouble3d getF3 ();
  public single DistArrayDouble3d getP (); // pressure
  // updates the coordinates of the registered points according to its
velocity
  public void move ();
}
```