A PARALLEL DYNAMIC-MESH LAGRANGIAN METHOD FOR SIMULATION OF FLOWS WITH DYNAMIC INTERFACES*

JAMES F. ANTAKI†, GUY E. BLELLOCH‡, OMAR GHATTAS§
IVAN MALČEVIĆ§, GARY L. MILLER¶, AND NOEL J. WALKINGTON||

Abstract. Many important phenomena in science and engineering, including our motivating problem of microstructural blood flow, can be modeled as flows with dynamic interfaces. The major challenge faced in simulating such flows is resolving the interfacial motion. Lagrangian methods are ideally suited for such problems, since interfaces are naturally represented and propagated. However, the material description of motion results in dynamic meshes, which become hopelessly distorted unless they are regularly regenerated. Lagrangian methods are particularly challenging on parallel computers, because scalable dynamic mesh methods remain elusive. Here, we present a parallel dynamic mesh Lagrangian method for flows with dynamic interfaces. We take an aggressive approach to dynamic meshing by triangulating the propagating grid points at *every timestep* using a scalable parallel Delaunay algorithm. Contrary to conventional wisdom, we show that the costs of the geometric components (triangulation, coarsening, refinement, and partitioning) can be made small relative to the flow solver. For example, in a simulation of 10 interacting viscous cells with 500,000 unknowns on 64 processors of a Cray T3E, dynamic meshing consumes less than 5% of a time step. Moreover, our experiments on up to 64 processors show that the computational geometry scales about as well as the flow solver. Therefore we anticipate that overall scalability on larger problems will be as good as the flow solver's.

1. Motivation. Flows with dynamic interfaces arise in many fluid-solid and fluid-fluid interaction problems, and are among the most difficult computational problems in continuum mechanics. Examples abound in the aerospace, automotive, biomedical, chemical, marine, materials, and wind engineering sciences. These include large-amplitude vibrations of such flexible aerodynamic components as high aspect ratio wings and blades; flows of mixtures and slurries; wind-induced deformation of towers, antennas, and lightweight bridges; hydrodynamic flows around offshore structures; interaction of biofluids with elastic vessels; and materials phase transition problems.

One of our target problems is modeling the flow of blood at the microstructural level. Blood is a mixture of deformable cellular bodies (primarily red blood cells) suspended in an essentially Newtonian fluid (plasma). The cells themselves are composed of a fluid gel (hemoglobin) contained within a solid membrane. The flow of the fluid–solid mixture is illustrated in Figure 1.1a, which depicts a snapshot of blood flow through a 12μm arteriole. Because of the computational difficulties of resolving numerous dynamically deforming cellular interfaces, as illustrated in Figure 1.1b, no one to date has simulated realistic blood flows at the microstructural level. Yet such simulations are necessary in order to gain a better understanding of blood damage—which is central to improved artificial organ design—and for the development of more rational macroscopic blood models.

We are specifically interested in simulating the flow of blood in artificial heart assist devices, such as the axial flow blood pump being developed at the University of Pittsburgh Medical Center [16]. Such pumps have regions in which the length scales of interest are of the order of tens of cell diameters, for example in bearing journals and near blade tips. Significant blood damage can occur in these "hot spots." Here, macroscopic models fail, and microstructural computations are necessary to quantify the important membrane stresses and deformations. To accurately model the interactions among the cells, plasma, and solid walls in a typical clearance region, a computational domain covering $200 \times 200 \times 25$ red blood cells is necessary. Adequate spatial discretization results in $\mathcal{O}(10^9)$ grid points, necessitating multi-teraflop computers. The challenge is to develop parallel numerical and geometric algorithms capable of resolving the numerous dynamic interfaces and scaling to the thousands of processors that characterize such machines.

2. Background on flow solvers for dynamic interface problems. Parallel flow solvers on fixed domains are reasonably well understood. In contrast, simulating flows with dynamic interfaces is much more challenging. The

*Computing services on the Pittsburgh Supercomputing Center's Cray T3E were provided under PSC grant ASC-990003P.

†McGowan Center for Artificial Organ Development, University of Pittsburgh Medical Center, Pittsburgh, PA 15217, USA (antakijf@msx.upmc.edu). Supported in part by NSF grant BES-9810162.

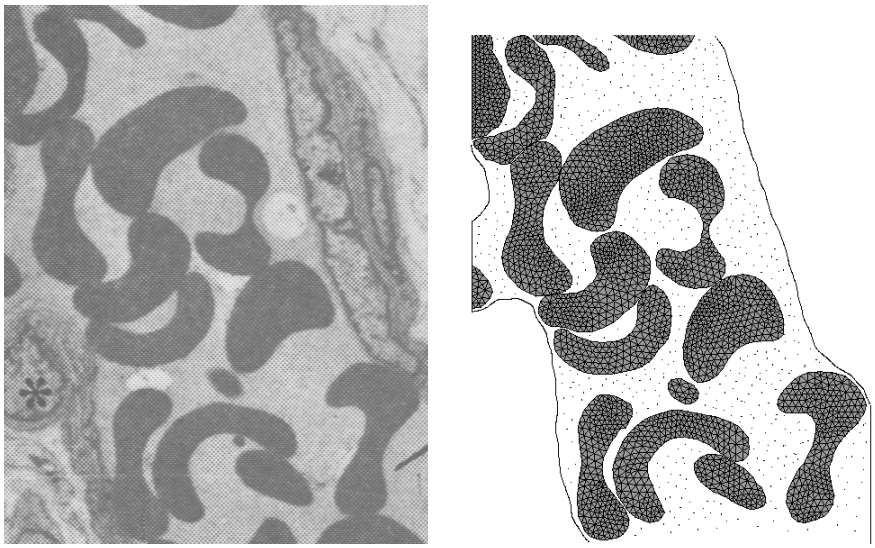
‡Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA (guyb@cs.cmu.edu). Supported in part by NSF grant EIA-9706572.

§Laboratory for Mechanics, Algorithms, and Computing, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, USA (oghattas@cs.cmu.edu, malcevic@andrew.cmu.edu). Supported in part by NSF grant ECS-9732301 and NASA grant NAG-1-2090.

¶Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA (glmiller@cs.cmu.edu). Supported in part by NSF grants EIA-9706572 and CCR-9902091.

||Department of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, PA 15213, USA (noelw@andrew.cmu.edu). Supported in part by NSF grants DMS-9973285 and CCR-9902091.

To appear in *Proceedings of Supercomputing 2000*, Dallas, TX, November 2000.
0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.



(a) (b)

FIG. 1.1. *(a) Electron micrograph of blood flow in a 12μm arteriole, showing interaction of cells with surrounding plasma and walls. The red blood cells are severely deformed—in a hydrostatic state they assume the shape of axisymmetric biconcave disks. (b) Numerical simulation of fluid-solid mixture, in this case a mesh-based method for the cells coupled with a particle method for the plasma.*

most natural framework for dynamic interface problems is a *Lagrangian* method, in which the interface representation is embedded in the material description of flow. The mesh convects and deforms with the flow, and thus interfaces that are well-resolved initially remain sharp and convect and deform with the flow. However, unless the interface motion is small, the mesh quickly distorts and/or becomes inappropriately coarse or fine, and some form of remeshing is required. The necessary dynamic unstructured mesh algorithms—those needed to coarsen, refine, retriangulate, and repartition the mesh as it evolves over time—are regarded as particularly challenging on highly parallel computers. Indeed, the 1997 Petaflops Algorithms workshop assessed the prospects of a number of high performance computing algorithms scaling to petaflops computers [18] (the assessment can be taken to be valid for multi-teraflops systems as well). Algorithms were classified according to Class 1 (scalable with appropriate effort), Class 2 (scalable provided significant research challenges are overcome), and Class 3 (possessing major impediments to scalability). The development of dynamic unstructured grid methods—including mesh generation, mesh adaptation and load balancing—were designated Class 2, and thus pose a significant, yet we believe achievable, research challenge. In contrast, static-grid PDE solvers (for example flow solvers on fixed grids) were classified as Class 1.

Motivated by the difficulties of dynamic meshes, the usual approach to solving flow problems with large interfacial motion is to relax the idea of conforming to the moving interfaces, and instead employ a regular, fixed grid throughout the domain. This description of motion is thus *Eulerian* (i.e. spatial). The effect of the interface is then incorporated through other means. Notable examples are *volume-of-fluid* methods pioneered by Hirt and Nichols [12], which use a volume-fraction field variable to distinguish different phases; the *immersed boundary method* of Peskin [19], in which interface forces appear as body forces in the fluid; *fictitious domain methods* of Glowinski [9], in which interface conditions are satisfied weakly with Lagrange multipliers; and *level set methods* introduced by Osher and Sethian [17], in which the transition in a smooth level set function delineates the interface. The attraction of all of these methods is that meshes are static and need not evolve in time, and that fast solvers are possible on the regular grids. This eases the parallelization of these methods.

The major drawback with fixed-grid methods, on the other hand, is the resulting fixed spatial resolution, particularly disadvantageous if one needs to vary resolution sharply throughout the domain. This is the case for example when interfacial dynamics strongly influence system behavior. Of particular interest to us, the dynamics of red cell membranes are critical to the cell damage process; high resolution is required near cell–plasma interfaces to accurately account for the membrane forces. Using a fine mesh everywhere to achieve high resolution in localized regions is far too costly, even on a teraflop computer.

A popular alternative is to use a moving "body-fitted" mesh that respects moving interfaces. This is usually

done in the context of an Arbitrary Lagrangian-Eulerian (ALE) method (the description is Lagrangian near moving interfaces and Eulerian at fixed boundaries, with appropriate mediation in between). The appeal of these methods is that when interfacial motion is small, mesh topology can be kept fixed over time. Typically, the deformation of the mesh is found by treating the fluid mesh as an elastic domain, and solving an auxiliary Dirichlet problem, with boundary motion imposed by the moving interfaces. Solution of the elasticity problem is readily parallelizable. Using such ALE methods, several groups have carried out large-scale parallel simulations of complex flow problems with dynamic interfaces, including nonlinear aeroelasticity simulations by Farhat's group [8] and simulations of liquid-particle systems by Tezduyar's group [23] and Joseph's group [14].

However, keeping the mesh topology fixed in time breaks down for problems with large relative interfacial motion—as occurs in blood flow, for example. In this case, significant mesh distortion and entanglement (and hence ill-conditioning of the CFD Jacobians) occurs, and the fixed mesh topology cannot be retained as the interfaces evolve. The mesh must now be regenerated frequently to conform to evolving and deforming boundaries. Another drawback is that a fixed-topology mesh that is appropriately resolved for one time instant will not be appropriate some number of time steps later. For example, steep refinement is needed in regions where cells approach either each other or solid surfaces. For problems with large interfacial motion, the mesh topology must change and adapt in response to the motion of the interfaces. And unless this is done in parallel, it quickly becomes a bottleneck for large problems. Even occasional remeshing [13] will not work for problems that involve highly deformable interfaces, such as microstructural blood flow. A much more aggressive approach to dynamic meshing is called for.

3. A parallel dynamic-mesh Lagrangian method. In this paper we take a completely different approach to dynamic-interface flow problems: in contrast to fixed-grid and fixed-topology-mesh methods, we adopt a Lagrangian framework that evolves the grid points at each time step. To maintain mesh quality and permit adaptivity, we generate in parallel an *entirely new* Delaunay triangulation of the dynamically-evolving grid points at *each time step*. The key to such an aggressive approach to meshing is our recent development of a parallel Delaunay triangulation algorithm that is theoretically optimal in work, requires polylogarithmic depth, and is very efficient in practice [5]. Given an arbitrary set of grid points, distributed across the processors, the algorithm returns a Delaunay triangulation of the points and at the same time partitions them for load balance and minimal communication across the processor boundaries.

Our 2D results show that frequent triangulation can be done with little overhead to the flow solver. For example, in a cylinder vortex shedding problem with half a million unknowns on 64 processors of a Cray T3E, the dynamic mesh computations account for less than 5% of the cost of a typical (implicit) time step. Our approach runs completely counter to methods commonly used in parallel adaptive PDE solvers, which often make incremental changes to the existing mesh at each time step, live with load imbalances for as long as possible before repartitioning, and accept suboptimal partitions (and hence increased interprocessor communication) in exchange for less costly reassignment of mesh objects to new processors.

We begin each time step with the grid points from the previous time step at their newly-acquired positions, but with their old triangulation. Since the new grid point locations may not produce well-shaped triangles everywhere nor desirable edge lengths, we add and remove (in our experience small numbers of) points to maintain a well-spaced criterion and hence quality triangulation, and satisfy edge-based error estimate criteria. Since the description of motion is Lagrangian, newly inserted grid points acquire values of velocity and pressure by simple interpolation on the current mesh. This step is of course highly parallel. We then completely discard the current mesh, and prepare to generate a new triangulation. We next compute a path of Delaunay edges (in 3D it would be a triangulated surface) that bisects the grid points, and continue bisecting recursively until the number of partitions is equal to the number of processors. Since each processor now owns a subdomain of grid points along with their Delaunay-conforming boundaries, we next switch to a purely sequential algorithm to triangulate the remaining grid points on each processor (in our implementation we use Shewchuk's Triangle code [21]). This approach avoids the costly reassignment of mesh entities to processors (as in conventional methods), since it is only the grid points that are partitioned; the (volume) mesh is created only locally. Once the mesh is built, the finite element-based Lagrangian flow solver computes new velocity and pressure fields, the grid points are convected to their new locations by the velocity field, and the entire process repeats.

By dynamically retriangulating the set of evolving grid points at each time step, we are able to track dynamic interfaces with high resolution, without the problems associated with distorted and entangled meshes. Since a new mesh is created at each time step, we can optimally refine, coarsen, and partition the mesh without being constrained by resemblance to prior meshes. And since grid points carry with them velocity and pressure information from one time step to the next, we avoid costly and error-prone projection of one mesh onto another. Below, we describe and

illustrate the parallel Delaunay algorithm (Section 4), give details on the flow solver (Section 5), and present results for a viscous incompressible multicomponent fluid–fluid interaction problem (Section 6). The implementation presented in this paper is for 2D flows, but the basic ideas extend to 3D.

4. Parallel Delaunay triangulation. Many of the theoretically efficient sequential algorithms for Delaunay triangulation are based on the divide-and-conquer paradigm [7, 10, 20]. The algorithms divide the points into regions, solve the problem recursively within each region, and then "sew" the boundaries back together. As well as being theoretically efficient (i.e. $O(n \log n)$ time), these algorithms also tend to be the fastest algorithms in practice [22]. Researchers have also shown that this divide-and-conquer technique can be applied in designing asymptotically efficient parallel algorithms [1, 6]. In this case, however, the approach does not appear to be practical. The problem is that although the sewing step can be parallelized, the parallel version is very much more complicated than the sequential version, and has large overheads.

To remedy this problem our parallel Delaunay algorithm uses a somewhat different approach [5]. We still use divide-and-conquer, but instead of doing most of the work when the recursive calls return we do most of the work at the divide step. This makes our joining step trivial: we just append the list of triangles returned by the recursive calls. This can be seen as similar to the difference between mergesort and quicksort—mergesort does most of its work when returning from the recursive calls, while quicksort does most of its work before making the recursive calls. It turns out that this gives us a much more efficient parallel algorithm. The algorithm is described in Figure 4.1 and illustrated in Figure 4.2. Although so far we have only implemented the algorithm in 2D, the basic idea extends to three dimensions. The main difference is that Step 4 would require a 3D convex-hull.

Algorithm: DELAUNAY(P, B)

Input:
P, a set of points in R^2
B, a set of Delaunay edges of P which is the border of a region in R^2 containing P.
Output: The set of Delaunay triangles of P which are contained within B.

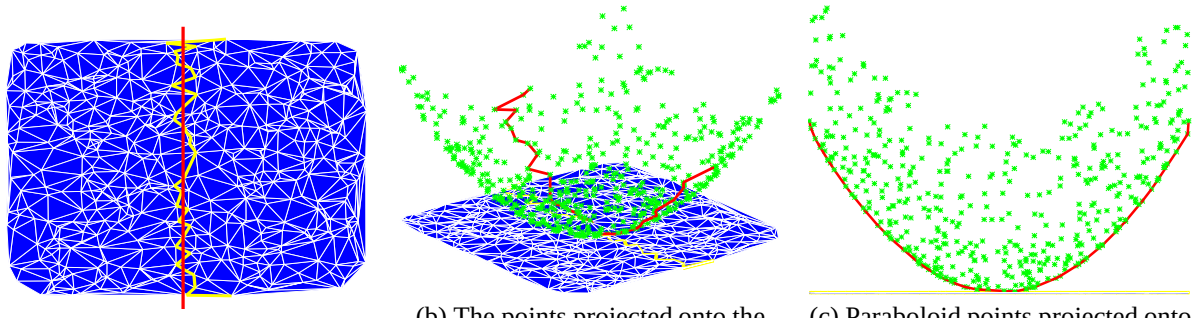
Method:

1. If all the points in P are on the border B, return Delaunay of (B).
2. Find the point q that is the median along the x axis of all internal points (points in P and not on the border). Let $\mathcal{L}$ be the line $x = q_x$.
3. Let $P' = \{(p_y - q_y, ||p - q||^2) \mid (p_x, p_y) \in P\}$. These points are derived from projecting the points P onto a $3D$ paraboloid centered at q, and then projecting them onto the vertical plane through the line $\mathcal{L}$.
4. Let $\mathcal{H} =$ LOWER_CONVEX_HULL(P'). $\mathcal{H}$ is a path of Delaunay edges of the set P. Let $P_{\mathcal{H}}$ be the set of points the path $\mathcal{H}$ consists of, and $\bar{\mathcal{H}}$ is the path $\mathcal{H}$ traversed in the opposite direction.
5. Create two subproblems:
   - $B^L =$ BORDER_MERGE$(B, \mathcal{H})$
     $B^R =$ BORDER_MERGE$(B, \bar{\mathcal{H}})$
   - $P^L = \{p \in P | p \text{ is left of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^L\}$
     $P^R = \{p \in P | p \text{ is right of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^R\}$
6. Return DELAUNAY$(P^L, B^L) \cup$ DELAUNAY(P^R, B^R)

FIG. 4.1. *The projection-based parallel Delaunay triangulation algorithm. Initially B is the convex hull of P. The algorithm as shown cuts along the x axis, but in general we can switch between x and y cuts, and all our implementations switch on every level.*

We implemented the 2D Delaunay algorithm using the Machiavelli toolkit [11], which is a library of routines built on top of MPI along with a small run-time system for load-balancing. Machiavelli was specifically designed for the efficient implementation of parallel divide-and-conquer algorithms on distributed memory machines, and is based on the recursive subdivision of asynchronous teams of processors. It obtains parallelism from data-parallel operations within teams and from the task-parallel invocation of recursive functions on independent teams of processors. When a processor has recursed down to a team containing only itself, it switches to a sequential version of the code. The Machiavelli library provides an abstract model of a vector, which is distributed in a block fashion across the processors of a team, and contains optimized communication functions for many of the idioms that occur in divide-and-conquer algorithms.

Our algorithm and implementation has several important and useful properties:



(a) The median line $\mathcal{L}$ and the path $\mathcal{H}$.

(b) The points projected onto the paraboloid.

(c) Paraboloid points projected onto the vertical plane.

FIG. 4.2. *Finding a dividing path for Delaunay triangulation. This shows the median line, all the points projected onto a parabola centered at a point on that line, and the horizontal projection onto the vertical plane through the median line. The result of the lower convex hull in the projected space, $\mathcal{H}$, is shown in highlighted edges on the plane.*

- The data are always fully distributed across the processors. This allows us to solve much larger problems than can fit in the memory of one processor.
- The algorithm performance does not depend on the initial point distribution—as well as generating a triangulation, the algorithm also partitions the points among processors to minimize communication.
- The algorithm works very well with both regular and highly irregular point distributions. Figure 4.3 gives run-times for both a random uniform point distribution as well as the "line" distribution which has points that become exponentially denser near a line. The line distribution is the worst case of the many distributions we tried. The line and uniform cases provide useful upper and lower bounds on the grid point distribution, and therefore performance, of what might be expected in CFD simulations.

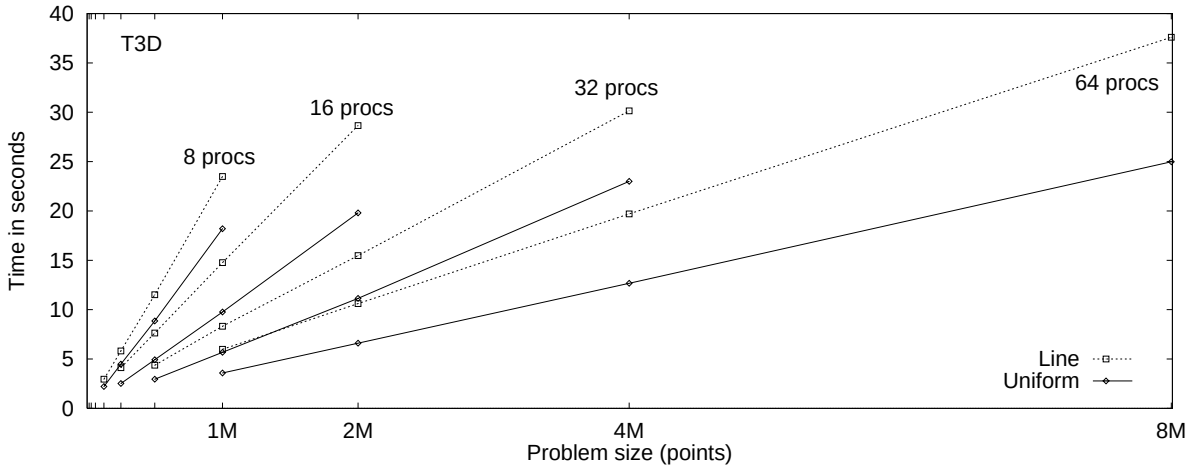


FIG. 4.3. *Scalability of the MPI and C implementation of our parallel Delaunay algorithm on the Cray T3D, showing the time to triangulate 16k-128k points per processor for a range of machine sizes.*

- The algorithm scales well. Figure 4.3 shows performance for up to 64 processors. Increasing problem size for a fixed number of processors (as in any of the curves in the figure) shows near-linear performance, and demonstrates that the algorithmic scalability of the algorithm is almost perfect. Increasing the number of processors while keeping problem size fixed (by following the vertical columns of data points in the figure) reveals 64% relative efficiency over a range of 8 to 64 processors for 1 million points. Of greatest interest to us (since our problems are memory-bound), increasing the number of processors while keeping the number of points per processor fixed (by tracking the terminal data point of each curve in the figure) shows 64% relative efficiency for the line distribution, and 72% for the uniform, over the same 8-fold increase in processors. These parallel efficiencies are similar to those observed in implicit CFD solvers for 8-fold isogranular scaling. Finally, compared to the fastest sequential algorithm we could find [7], on 64 processors

our algorithm can solve a problem that is 32 times larger in the same time. Based on our experiments, and on the algorithm's theoretical scalability properties ($\mathcal{O}(n \log n)$ work and $\mathcal{O}(\log^3 n)$ depth), we expect that our parallel Delaunay algorithm will not appreciably reduce the isogranular scalability of a typical implicit CFD solver.

- The algorithm is very fast. Triangulating 1 million points on 64 processors requires 3.5s for the uniform distribution. This is about an order of magnitude cheaper than a typical timestep in a state-of-the-art implicit CFD solver for a 1 million grid point simulation on the same machine (extrapolated from [15]).

5. A 2D parallel dynamic-mesh Lagrangian flow solver. In this section we give an overview of the flow solver component of our parallel dynamic-mesh Lagrangian method. We do this in the context of Newtonian viscous incompressible flows (which may include multicomponent fluids), although the essential ideas extend to other fluid models as well as fluid-solid interaction problems.

The PDEs for flow of a Newtonian viscous incompressible fluid in Lagrangian form are the conservation of momentum and mass equations,

$$\rho \frac{D\boldsymbol{v}}{Dt} - \boldsymbol{\nabla} \cdot \left[\mu \left(\boldsymbol{\nabla v} + \boldsymbol{\nabla v}^T \right) - \boldsymbol{I} p \right] = \boldsymbol{f}, \tag{5.1}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0, \tag{5.2}$$

and the velocity definition (in terms of particle position $\boldsymbol{x}$),

$$\boldsymbol{v} - \frac{\partial \boldsymbol{x}}{\partial t} = \boldsymbol{0}, \tag{5.3}$$

where $\boldsymbol{v}(\boldsymbol{x}, t)$, $p(\boldsymbol{x}, t)$, and $\boldsymbol{f}(\boldsymbol{x}, t)$ are the velocity, pressure, and body force fields, and μ and ρ are the piecewise homogeneous viscosity and density coefficients. The Lagrangian description of motion involves the material time derivative, which can be approximated for a finite time increment Δt by

$$\frac{D\boldsymbol{v}}{Dt} \equiv \lim_{\delta t \to 0} \frac{\boldsymbol{v}(\boldsymbol{x}(t+\delta t), t+\delta t) - \boldsymbol{v}(\boldsymbol{x}(t), t)}{\delta t} \simeq \frac{\boldsymbol{v}(\boldsymbol{x}_{k+1}, t_{k+1}) - \boldsymbol{v}(\boldsymbol{x}_k, t_k)}{\Delta t}, \tag{5.4}$$

where $t_{k+1} \equiv t_k + \Delta t$ and $\boldsymbol{x}_k \equiv \boldsymbol{x}(t_k)$. Implicit backward Euler discretization in time[1] then gives the (spatially-continuous) system of PDEs written at time t_{k+1},

$$\frac{1}{\Delta t} \rho \boldsymbol{v}_{k+1} - \boldsymbol{\nabla} \cdot \left[\mu \left(\boldsymbol{\nabla v}_{k+1} + \boldsymbol{\nabla v}_{k+1}^T \right) - \boldsymbol{I} p_{k+1} \right] = \boldsymbol{f}_{k+1} + \frac{1}{\Delta t} \rho \boldsymbol{v}_k, \tag{5.5}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{v}_{k+1} = \boldsymbol{0} \tag{5.6}$$

$$\boldsymbol{v}_{k+1} - \frac{1}{\Delta t} \boldsymbol{x}_{k+1} = -\frac{1}{\Delta t} \boldsymbol{x}_k, \tag{5.7}$$

where $\boldsymbol{v}_{k+1} \equiv \boldsymbol{v}(\boldsymbol{x}_{k+1}, t_{k+1})$, and $p_{k+1} \equiv p(\boldsymbol{x}_{k+1}, t_{k+1})$. This system appears to be a linear Stokes–like system, but note that the new time step velocity and pressure fields $\boldsymbol{v}_{k+1}$ and p_{k+1} are defined at the new particle positions $\boldsymbol{x}_{k+1}$, which in turn depend on $\boldsymbol{v}_{k+1}$ through (5.7). Thus, (5.5)–(5.7) are coupled and nonlinear.

Spatial approximation by finite elements results in the system of nonlinear algebraic equations at time t_{k+1},

$$\frac{1}{\Delta t} \boldsymbol{M}(\boldsymbol{x}_{k+1}) \boldsymbol{v}_{k+1} + \boldsymbol{A}(\boldsymbol{x}_{k+1}) \boldsymbol{v}_{k+1} + \boldsymbol{B}^T(\boldsymbol{x}_{k+1}) \boldsymbol{p}_{k+1} = \boldsymbol{F}_{k+1} + \frac{1}{\Delta t} \boldsymbol{M}(\boldsymbol{x}_k) \boldsymbol{v}_k, \tag{5.8}$$

$$\boldsymbol{B}(\boldsymbol{x}_{k+1}) \boldsymbol{v}_{k+1} = \boldsymbol{0} \tag{5.9}$$

[1] We could have used Crank-Nicolson, but this would have required us to compute on an intermediate mesh, without permitting a significantly larger time step—here the time steps are limited by mesh distortion. Thus, we opted for the simplicity of backward Euler.

$$\boldsymbol{v}_{k+1} - \frac{1}{\Delta t}\boldsymbol{x}_{k+1} = -\frac{1}{\Delta t}\boldsymbol{x}_k, \tag{5.10}$$

where $\boldsymbol{v}_{k+1}$, $\boldsymbol{p}_{k+1}$, and $\boldsymbol{x}_{k+1}$ are (now) the unknown velocity, pressure, and position vectors associated with grid points at time t_{k+1}, and $\boldsymbol{M}$, $\boldsymbol{A}$, and $\boldsymbol{B}$ are mass, discrete "Laplacian," and discrete divergence finite element matrices. The dependence of these matrices on the new (unknown) particle positions $\boldsymbol{x}_{k+1}$ through the element geometry is shown explicitly in (5.8) and (5.9). In our implementation, we use a Galerkin finite element method with the linear velocity–linear pressure triangle $(\mathcal{P}_1/\mathcal{P}_1)$. To circumvent the Ladyzhenskaya–Babuška-Brezzi condition, we pressure-stabilize à la [24], but with modifications to maintain symmetry of the coefficient matrix of the linearized system. Thus we have $5n$ equations in $5n$ variables, i.e. two velocities components, one pressure, and two position components at each of n grid points.

We choose to solve the nonlinear system (5.8)–(5.10) using a Picard–like iteration in which the matrix dependence on particle position is lagged behind the velocity and pressure update. With iteration index i (and initial guess equal to the previous time step's converged values), this gives

$$\begin{bmatrix} \frac{1}{\Delta t}\boldsymbol{M}(\boldsymbol{x}^i_{k+1}) + \boldsymbol{A}(\boldsymbol{x}^i_{k+1}) & \boldsymbol{B}^T(\boldsymbol{x}^i_{k+1}) \\ \boldsymbol{B}(\boldsymbol{x}^i_{k+1}) & \boldsymbol{C}(\boldsymbol{x}^i_{k+1}) \end{bmatrix} \begin{Bmatrix} \boldsymbol{v}^{i+1}_{k+1} \\ \boldsymbol{p}^{i+1}_{k+1} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_{k+1} + \frac{1}{\Delta t}\boldsymbol{M}(\boldsymbol{x}_k)\boldsymbol{v}_k \\ \boldsymbol{0} \end{Bmatrix}, \tag{5.11}$$

$$\boldsymbol{v}^i_{k+1} - \frac{1}{\Delta t}\boldsymbol{x}^{i+1}_{k+1} = -\frac{1}{\Delta t}\boldsymbol{x}_k, \tag{5.12}$$

where solution of (5.11) yields the updated velocities and pressures given particle positions, and evaluation of (5.12) updates the particle positions given the new velocities. Provided the mesh does not distort too badly, we rarely need more than 2 or 3 Picard iterations per time step (which is another reason to triangulate every time step). We favor this Picard iteration mainly because the resulting Stokes-like linear system (5.11) is reduced in size from $5n$ to $3n$, and its coefficient matrix is rendered symmetric. (Note the presence of $\boldsymbol{C}$, which would have been zero in the absence of pressure stabilization.)

To solve the linear system (5.11), we use a parallel implementation of GMRES(k) from the PETSc library for parallel solution of PDEs [2, 3, 4]. We also use PETSc's domain decomposition preconditioners, in particular overlapping additive Schwarz with incomplete LU factorization on each processor. The results presented in the next section correspond to GMRES(30), overlap of 2, and ILU(0), for which we have obtained good performance. Several thousand iterations are usually required to converge a problem with about a quarter million unknowns on 64 processors, which is reasonable considering the ill-conditioned, saddle-point nature of the original linear system. The key component is pressure stabilization, which reduces the number of Krylov iterations by an order of magnitude.

6. Results and conclusions. The parallel dynamic-mesh Lagrangian method described in this paper has been implemented in C and MPI for 2D multi-component viscous incompressible fluid flows. All major components of the code are parallel: grid-point transport, stabilized velocity–pressure finite element approximation, linear Krylov solver and ILU additive Schwarz preconditioner (provided by PETSc), nonlinear solver, time stepper, and computational geometry algorithms (convex hull, Delaunay, partitioning, coarsening, and refinement).

The multicomponent model supports multiple immiscible fluids (with differing densities and viscosities), and this can be used to distinguish blood plasma from the hemoglobin of multiple cells. For a more realistic blood flow model, we would have to add a finitely-deforming elastic membrane to enclose the cell hemoglobin, which our current model lacks. We plan to do this as a next step in scaling up to 3D blood flow simulations. Still, our current multicomponent model is indicative of the performance of the Lagrangian method for flow simulations of bodies with elastic membranes, since the elastic computations would be overwhelmed by the flow solver and geometric components.

The code has been used to simulate the dynamics of 10 fluid "cells" of differing densities and viscosities, interacting with themselves and a surrounding fluid. Figure 6.1 shows some snapshots from a simulation on 64 processors of the Cray T3E-900 at the Pittsburgh Supercomputing Center. The basic computational entities are the grid points, which move freely with the flow, and are depicted for six different time steps in the figure. An aggressive approach to dynamic meshing is essential here: the mesh becomes hopelessly twisted and distorted in the regions surrounding the deforming cells unless retriangulation is done at each time step. Figure 6.2 shows a closeup of the mesh in the vicinity of two interacting cells. One can see that resolution is enhanced in the fluid layer between approaching cells. This resolution captures the increased fluid pressure within the layer (present in the continuum model) in a natural way, thereby preventing contact between cells without resort to artificial "repulsion" forces that have been used in the past

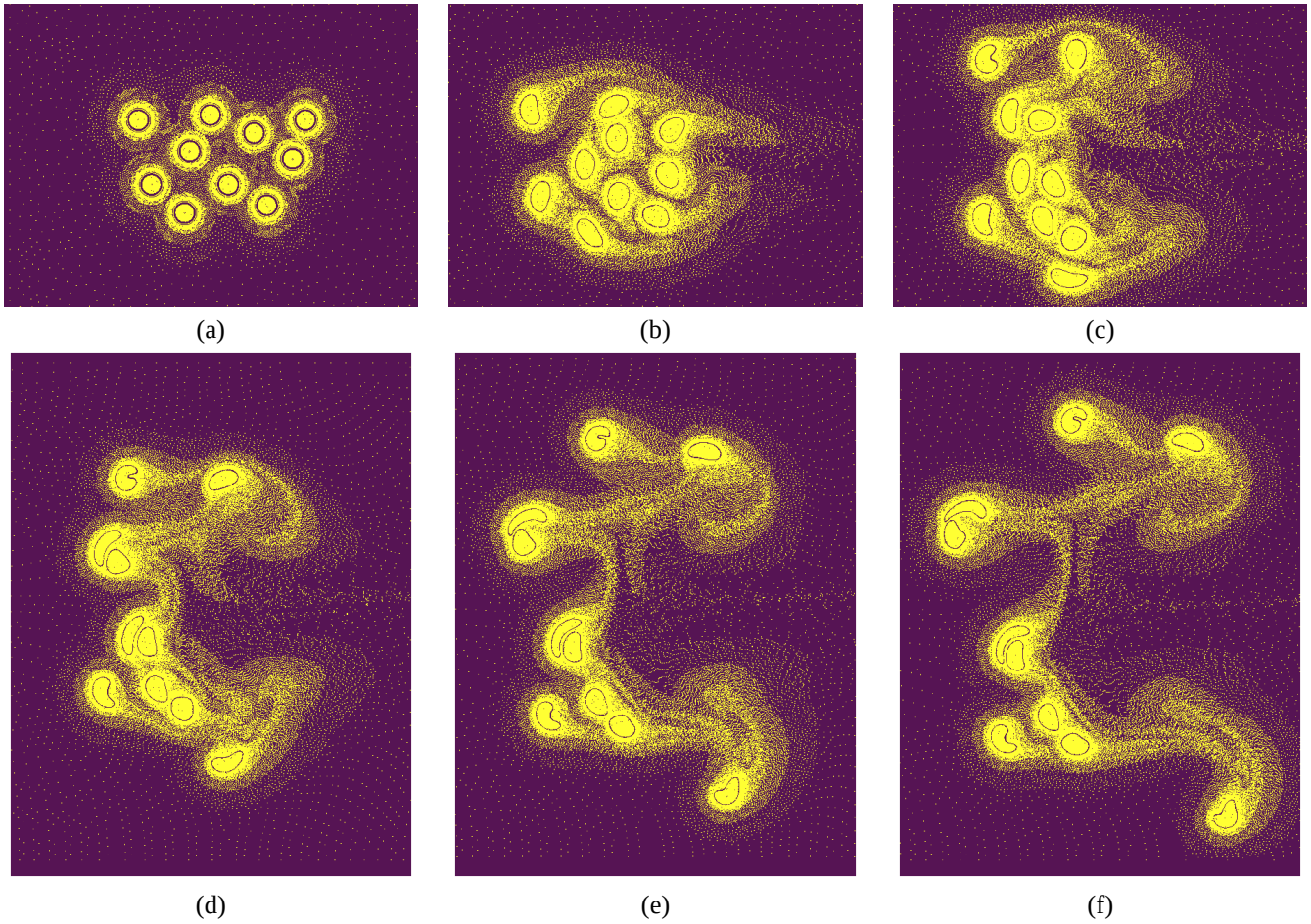


FIG. 6.1. *Snapshots from a multicomponent fluid simulation of 10 viscous cells interacting with each other and with a surrounding less dense/viscous fluid. For clarity, the triangulation is omitted, and only grid points are shown. The simulation was performed with a parallel dynamic-mesh Lagrangian flow solver on 64 processors of a Cray T3E-900. An animation may be viewed at http://www.cs.cmu.edu/∼oghattas/10cell.avi, and gives a much better sense of the motion than do these static images.*

for flow-body simulations. Finally, Figure 6.3 shows mesh partitions for the time instants of Figures 6.1d and 6.1e, and demonstrates that the partitions can change noticeably between two nearby time instants.

Table 6.1 provides timings for the 10 cell problem for a typical time step. Results are given for 32 and 64 processors, for both fixed and isogranular-granular scaling, up to a maximum of a quarter million velocity/pressure unknowns (the number of grid points is thus a third of this). Execution times are in seconds, and are reported for both flow solver and dynamic mesh components, as well as overall. The flow solver component is further broken down into assembly/setup time and linear solver time, while contributions to the dynamic meshing times are shown for the parallel Delaunay, refinement/coarsening, mesh rebuilding, and interface proximity detection steps. (Actually, the latter computation, which essentially determines local feature size for refinement near cell interfaces, has not yet been parallelized, and instead one processor keeps track of the interface interactions.) The timings indicate that the computational geometry components are very cheap relative to the flow solver; in particular, the Delaunay triangulation takes less than 1% of the flow solver time.

Note that because this is a nonlinear implicit time step, the dynamic mesh computations are amortized over several thousand matrix-vector products and preconditioner applications, so that a better-conditioned linear system would reduce the flow solver execution time, and make the computational geometry components relatively more expensive.[2] On the other hand, with an explicit method, the Courant-limited time steps would typically be much smaller, so the

[2]Note, however, that mesh-adaptivity in general results in element sizes that vary over a wide range, and this leads to ill-conditioning of the linear system.

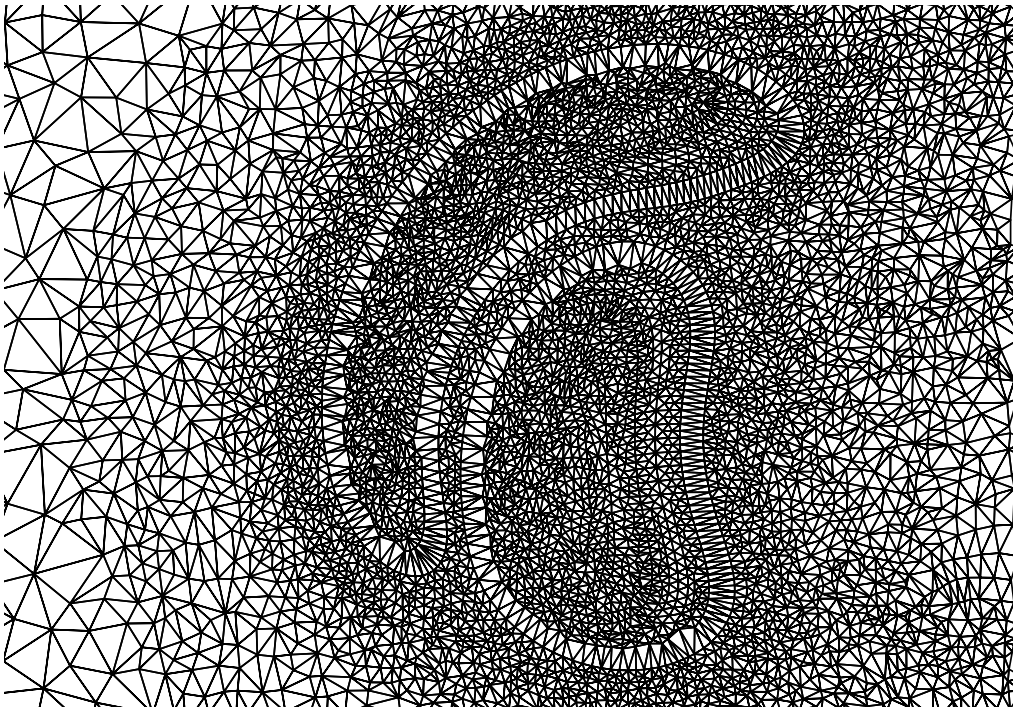
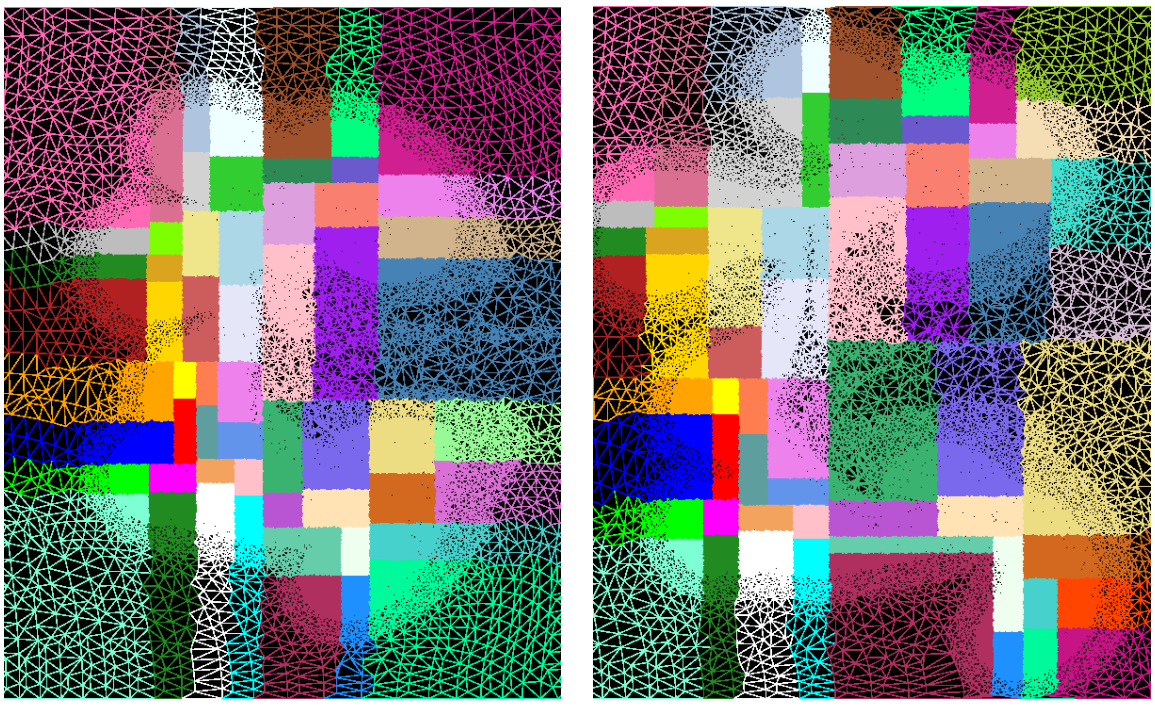


FIG. 6.2. *Closeup of mesh in vicinity of two interacting cells. Layer between cell interfaces is well-resolved.*



(a) (b)

FIG. 6.3. *Dynamically-partitioned mesh corresponding to time steps of Figure 3d and 3e.*

mesh would not change much from time step to time step, and therefore we would triangulate much less frequently. Thus, we would expect to maintain similar ratios between flow solver and dynamic mesh times, even for an explicit method.

problem size	processors	flow solver		dynamic meshing				total
		assembly	solve	Delaunay	refine/coarsen	rebuild	interface	
252,975	64	6.35	60.86	0.45	1.20	0.63	1.17	70.66
252,975	32	10.94	117.9	0.71	2.10	1.06	1.25	133.96
132,975	32	4.89	45.12	0.42	0.97	0.72	1.04	53.16

TABLE 6.1
Timings in seconds for a typical time step for 10 cell problem

Because the dynamic meshing components are such a small part of a time step, and based on the theoretical and experimental scalability of our parallel Delaunay algorithm, we can expect that overall scalability of the dynamic mesh flow solver will be governed by the the flow solver component. Although proper scalability assessment requires timings over a much broader range of processors, here we just examine the drop in parallel efficiency for a doubling of processors from 32 to 64 in order to get a sense of the relative behavior of dynamic meshing and flow solver components for this medium-sized problem.

Scaling from 32 to 64 processors while keeping problem size fixed at a quarter-million unknowns gives us an indication of parallel efficiency independent of the algorithmic efficiency.[3] From the table, we see that the flow solver maintains 96% parallel efficiency in going from 32 to 64 processors, while the dynamic mesh component is 74% efficient. However, if we exclude the currently sequential interface proximity detection step, mesh-related parallel efficiency rises to 85%. In any case, its contribution to overall time is so small that overall parallel efficiency remains at 95%. On the other hand, (roughly) doubling problem size while doubling the number of processors provides a measure of isogranular parallel efficiency. In this case, efficiency for dynamic meshing rises to 87%, because its components are highly algorithmically efficient—at worst $\mathcal{O}(n \log n)$. The situation is reversed for the flow solver—because the linear solver lacks a coarse mesh solve and the linear systems are generally quite ill-conditioned, the linear solver behaves superlinearly (i.e. requires $\mathcal{O}(n^\alpha)$ work, where $\alpha > 1$). As a result, the flow solver's isogranular parallel efficiency is around 71%. Finally, the overall (flow solver and dynamic mesher) isogranular efficiency in scaling from 32 to 64 processors is about 72%.

These results demonstrate that with careful algorithm design, the cost of creating a completely new mesh at every time step (including refinement, coarsening, triangulation, and partitioning) can be made small relative to that of the flow solver—contrary to conventional wisdom. Furthermore, the computational geometry components are roughly as parallel and scalable as the flow solver, and thus should not limit the parallel efficiency of the entire dynamic mesh flow solver (including geometric and numerical components). Much work remains to scale up our implementation to 3D, to hundreds of thousands of interacting cells, and to include realistic membrane models, but we believe the method presented in this paper provides us with the right framework.

REFERENCES

[1] A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C. O. DÚNLAING, AND C. YAP, *Parallel computational geometry*, Algorithmica, 3 (1988), pp. 293–327.

[2] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[3] ———, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.

[4] ———, *PETSc home page*. `http://www.mcs.anl.gov/petsc`, 1999.

[5] G. BLELLOCH, J. HARDWICK, G. L. MILLER, AND D. TALMOR, *Design and implementation of a practical parallel Delaunay algorithm*, Algorithmica, 24 (1999), pp. 243–269.

[6] R. COLE, M. T. GOODRICH, AND C. O. DÚNLAING, *Merging free trees in parallel for efficient Voronoi diagram construction*, in Proceedings of the 17th International Colloquium on Automata, Languages and Programming, July 1990, pp. 32–45.

[7] R. DWYER, *A simple divide-and-conquer algorithm for constructing Delaunay triangulations in $O(n \log \log n)$ expected time*, in 2nd Symposium on Computational Geometry, May 1986, pp. 276–284.

[8] C. FARHAT, *Nonlinear transient aeroelastic simulations.* `http://caswww.colorado.edu/ charbel/project.html`.

[9] R. GLOWINSKI, T. PAN, AND J. PERIAUX, *Fictitious domain methods for incompressible viscous flow around moving rigid bodies*, in The Mathematics of Finite Elements and Applications, John Wiley & Sons, 1996, pp. 155–174.

[3] Actually, this is not strictly speaking true: the $P = 32$ and $P = 64$ cases correspond to somewhat different linear system preconditioners (the smaller the subdomain, the less effective the preconditioner). But the effect is not pronounced.

[10] L. GUIBAS AND J. STOLFI, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Transactions on Graphics, 4 (1985), pp. 74–123.
[11] J. C. HARDWICK, *An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms*, in Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments, IEEE, Apr. 1996, pp. 105–114.
[12] C. W. HIRT AND B. D. NICHOLS, *Volume of fluid (VOF) method for the dynamics of free boundaries*, Journal of Computational Physics, 39 (1981), pp. 201–225.
[13] A. JOHNSON AND T. TEZDUYAR, *3D simulation of fluid-particle interactions with the number of particles reaching 100*, Computer Methods in Applied Mechanics and Engineering, 145 (1997), pp. 301–321.
[14] D. D. JOSEPH, *Direct simulation of the motion of particles in flowing liquids*. `http://www.aem.umn.edu/Solid-Liquid_Flows`.
[15] D. E. KEYES, D. K. KAUSHIK, AND B. F. SMITH, *Prospects for CFD on Petaflops systems*, in CFD Review 1998, M. Hafez and K. Oshima, eds., World Scientific, 1998, pp. 1079–1096.
[16] *McGowan Center for Artificial Organ Development: Artificial Heart Development.* `http://www.upmc.edu/mcgowan/ArtHeart`.
[17] S. OSHER AND J. SETHIAN, *Fronts propagating with curvature dependent speed: Algorithms based on Hamilton Jacobi formulations*, Journal of Computational Physics, 79 (1988), pp. 12–49.
[18] *The 1997 Petaflops Algorithms Workshop summary report.* `http://www.hpcc.gov/cicrd/pca-wg/pal97.html`.
[19] C. PESKIN, *Numerical analysis of blood flow in the heart*, Journal of Computational Physics, 25 (1977), pp. 220–252.
[20] M. I. SHAMOS AND D. HOEY, *Closest-point problems*, in Proceedings of the 16th Annual Symposium on Foundations of Computer Science, IEEE, Oct. 1975, pp. 151–162.
[21] J. R. SHEWCHUK, *Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator*, in First Workshop on Applied Computational Geometry, Association for Computing Machinery, May 1996, pp. 124–133.
[22] P. SU AND R. DRYSDALE, *A comparison of sequential Delaunay triangulation algorithms*, Comput. Geom. Theory Appl., 7 (1997), pp. 361–386.
[23] T. TEZDUYAR, *Team for advanced flow simulation and modeling: Fluid-object interactions.* `http://www.mems.rice.edu/TAFSM/PROJ/FOI/`.
[24] T. E. TEZDUYAR, *Stabilized finite element formulations for incompressible flow computations*, Advances in Applied Mechanics, 28 (1991), pp. 1–44.