Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures

Abtin Rahimian*, Ilya Lashuk*, Shravan K. Veerapaneni†, Aparna Chandramowlishwaran*
Dhairya Malhotra*, Logan Moon*, Rahul Sampath‡, Aashay Shringarpure*,
Jeffrey Vetter‡, Richard Vuduc*, Denis Zorin†, and George Biros*

* Georgia Institute of Technology, Atlanta, GA 30332
† New York University, New York, NY 10002
‡ Oak Ridge National Laboratory, Oak Ridge, TN 37831
rahimian@gatech.edu, ilashuk@cc.gatech.edu, shravan@cims.nyu.edu,
aparna@cc.gatech.edu, dhairya.malhotra88@gmail.com, lmoon3@gatech.edu,
rahul.sampath@gmail.com, aashay.shringarpure@gmail.com,
vetter@computer.org, richie@cc.gatech.edu, dzorin@cs.nyu.edu, gbiros@acm.org

***Abstract*—We present MoBo ("Moving Boundaries"), a fast, petaflop-scalable algorithm for Stokesian particulate flows. Our target application is the direct simulation of blood, which we model as a mixture of a Stokesian fluid (plasma) and red blood cells (RBCs). We present simulations with up to 200 million *deformable* RBCs. This problem corresponds to 66 billion unknowns in space, while typical simulations require $\mathcal{O}$(100–1000) time steps. By contrast, prior literature on methods of comparable modeling fidelity have scaled to $\mathcal{O}$(1,000–10,000) cells; thus, MoBo improves on prior art by four orders of magnitude in terms of the number of cells.**

Directly simulating blood is a highly challenging multiscale, multiphysics problem. Compared to other simulations that use rigid particles, our approach has three distinct characteristics: (1) we faithfully represent the physics of RBCs by using nonlinear solid mechanics to capture the deformations of each cell; (2) we accurately resolve the long-range, N-body, hydrodynamic interactions between RBCs (caused by the surrounding plasma); and (3) we allow for the highly non-uniform distribution of RBCs in space.

We designed MoBo to support parallelism at all levels, including inter-node distributed memory parallelism, intra-node shared memory parallelism, data parallelism (vectorization), and fine-grained multithreading for GPUs. We present results within node for single-core, single-socket, GPU, and hybrid multicore CPU-GPU; CPU-GPUs on the Lincoln cluster; and on 200,000 cores of the Oak Ridge National Laboratory's Jaguar PF system. For the latter, we achieve 0.7 Petaflops/s of sustained performance. Thus, beyond its scientific significance, MoBo shows how heterogeneous architectures and programming models can be applied to the petascale.

I. Introduction

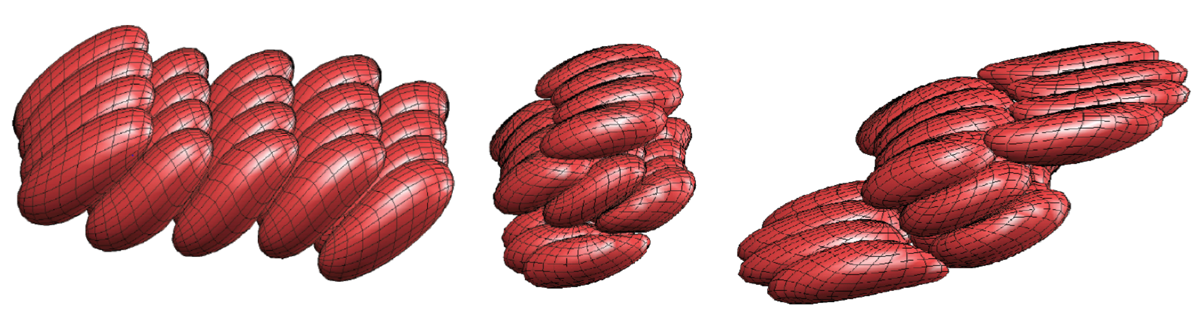
Clinical needs in thrombosis risk assessment, anti-coagulation therapy, and stroke research depend heavily on a significantly improved understanding of the microcirculation of blood and platelet rheology. Toward this end, we present a new computational infrastructure, called MoBo, that enables the *direct numerical simulation of several microliters of blood*, which consists of hundreds of millions of cells, at new levels of physical fidelity. (Figure 1) MoBo comprises two key algorithmic components: **(1)** scalable integral equation solvers for Stokesian flows with dynamic interfaces; and **(2)** scalable fast multipole algorithms. In terms of size alone, MoBo's overall simulation capability represents an advance that is orders of magnitude beyond prior work.

Challenges in Direct Numerical Simulation of Blood. The direct numerical simulation of blood is a multiscale, multiphysics problem of unprecedented scale. Just one microliter, or one drop, of blood can contain millions of red blood cells (**RBCs**). In addition, the surrounding plasma has its own charactersitics, the physics of which also require resolution. Unlike other RBC simulations that use rigid particles, we aim to better represent the RBC physics by (a) using nonlinear solid mechanics to capture the deformations of each RBC; (b) more accurately resolving the long-range, N-body-type hydrodynamic interactions between RBCs, caused by the surrounding plasma; and (c) allow for the non-uniform distribution of particles in space, which may span seven orders of magnitude in length scales. Finally, we must ensure that the volume and surface area of each RBC are conserved. As a result, resolving the dynamics of RBC deformation results in hundreds to thousands of discretization points per cell, and, thus, billions or trillions of unknowns for an entire blood flow simulation.

To our knowledge, for simulations with the same capabilities in resolving the physics of blood flow across several time and length scales, the largest systems previously considered use (a) 1,200 cells [41], in highly accurate models (integral equations) like ours; and (b) 14,000 cells [9], using Lattice Boltzmann methods. The latter has been parallelized on up to 64K cores. Other methods that treat RBCs as rigid bodies have scaled to a large number of cells, but the physics remain a very crude approximation of the actual blood flow. Red blood cells are not rigid, Figure 1.[1]

Our Approach. To address the challenges associated with

[1]For example, 65% volume-fraction suspension of rigid spheres cannot flow, whereas blood continues flowing even when the volume-fraction in the plasma reaches 95% [3].



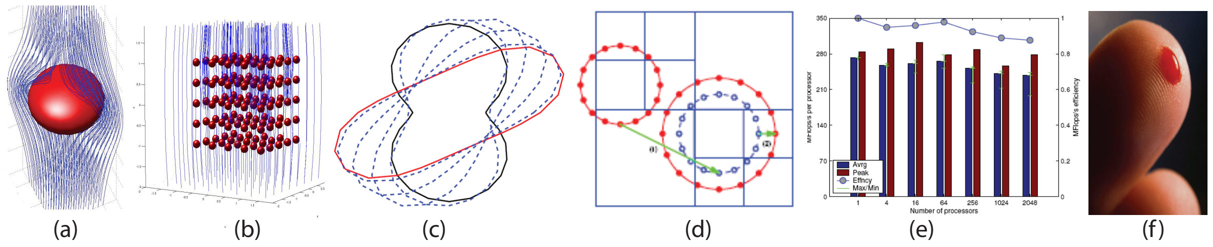


Fig. 1: COMPUTATIONAL INFRASTRUCTURE FOR PARTICULATE FLOWS. *We have developed computational tools for the efficient solution of boundary integral equations: (a) surface representations and quadratures for singular integrals; (b) many-particle solvers; (c) nonlinear solvers for deformable red blood cells; and (d,e) parallel, kernel-independent, tree-based, fast summation methods. The advantage of boundary integral methods is that only the particle boundary is discretized. This is crucial for decreasing the number of degrees of freedom for large-scale, high-accuracy simulations and eliminates the need for computationally intensive and difficult-to-parallelize 3D meshing techniques for deformable boundaries. With an appropriate fast-summation scheme, boundary integral methods result in optimal algorithmic complexity. Our tools enable parallel, arbitrary-accuracy, adaptive, and efficient simulation of deformable particles suspended in Stokesian flows. The target application is the direct numerical simulation of* $\mathcal{O}(10^{-3})$–$\mathcal{O}(1)$ *microliters of blood flow (one can think of a blood drop as roughly one microliter) in small vessels.*

the direct numerical simulation of blood flow, we have designed MoBo to have the following basic components:

- We use an integro-differential formulation in which we couple a boundary integral formulation for the Stokes equations (plasma) with a Helfrich free energy term for the vesicle's membrane elasticity.
- We use spherical harmonics representations for the deformation of RBCs.
- We use the fast multipole method to resolve long-range hydrodynamic interactions between cells and plasma.
- We use hybrid parallelism at all levels to expose maximum concurrency, including distributed and shared memory parallelism, data parallelism (vectorization), and massive fine-grained multithreading via GPGPU acceleration.

In MoBo, we employ an arsenal of celebrated methods in mathematics and computer science: Fourier and Legendre transforms, adaptive fast multipole methods, antialiasing, remeshing, Galerkin projections, multi-step time methods, fast spherical harmonics rotations, spectral quadratures for smooth and weakly singular integrals, preconditioned Krylov linear solvers, and dense linear algebra operations.

Roughly speaking, our overall formulation can be summarized as follows. Every cell is represented by a number of points; in our examples, we use either 84 or 312 points. The motion of each point $\mathbf{x}$ is governed by

$$\begin{aligned}\frac{\partial \mathbf{x}}{\partial t} &= \mathbf{v}(\mathbf{x}),\\ \mathbf{v}(\mathbf{x}) &= \mathbf{v}_{local} + \mathbf{v}_{global} + \mathbf{v}_{background}.\end{aligned} \tag{1}$$

Here, $\mathbf{v}$ is the velocity of the point, which we decompose into three components: $\mathbf{v}_{local}$, $\mathbf{v}_{global}$, and $\mathbf{v}_{background}$. The "*local*" velocity accounts for the interactions between the specific point in the RBC under consideration and all of the other points within that same RBC. The "*global*" velocity accounts for all of the interactions occurring across all of the RBCs in the simulation. The "*background*" velocity is the imposed flow field. One can view MoBo as a particle method in which we compute the particle velocity and then update the particle position.[2] This work builds on our previous work on massively parallel tree-data structures [30], [27], parallel and kernel independent fast multipole methods [37], [18], [8], and fast solvers for particulate flows [33], [25], [34].

Contributions. The details of the mathematical apparatus that is required to compute $\mathbf{v}_{local}$ and $\mathbf{v}_{global}$ have been described previously [25]. In this paper, we focus on the parallelization and performance analysis for the computation of $\mathbf{v}_{local}$ and $\mathbf{v}_{global}$. The former requires 9 different kernels. The latter requires a fast summation scheme for the Stokes kernel. In summary,

[2]This description is convenient, but can be misleading. The derivation of the equation of motion has nothing to do with particles. In fact, the points we use to represent the cells change throughout the simulation due to the reparameterization of the cells' surfaces.

- We present a hybrid implementation of 9 computational kernels that MoBo uses for the computation of $\mathbf{v}_{local}$ and $\mathbf{v}_{global}$. The kernels are multithreadead and distributed on both GPU and CPUs, which share the workload concurrently, thereby delivering excellent per-node performance.
- The most intensive kernels in our computation have been designed for locality, accuracy, and computational efficiency by capitalizing on highly optimized BLAS3 (GEMM) operations.
- We further improve the performance our SC'09 FMM algorithm [18]. These improvements include SSE vectorization, multithreading via OpenMP, and simultaneous asynchronous GPU acceleration. To our knowledge, our FMM code remains the most scalable and generally-applicable code available to date.
- We present single-node analysis for computations of $\mathbf{v}_{local}$ and $\mathbf{v}_{global}$ on AMD, Intel, and NVIDIA platforms.
- We present weak and strong scaling results on the Jaguar PF system at Oak Ridge National Laboratory (ORNL).

Performance Highlights. We achieve 780 Tflop/s of sustained performance on the 196,608 cores of the AMD Istanbul-based Kraken system (4 Gflop/s per core), with 160$\times$ speedup on strong scaling when moving from 48 to 24,576 cores (512$\times$); and 64% efficiency on weak scaling. On other platforms, we demonstrate up to 18 Gflop/s per core of sustained performance on the Intel Nehalem-EP; and up to 175 Gflop/s per NVIDIA T10P-based GPU.

For our largest run, we solved a problem involving 8,000 RBCs per MPI process, on 32,768 MPI processes for a total of 196,608 cores. We discretized in space using 84 points per RBC. This set of parameters results in a total of 262,144,000 red blood cells (50 drops of blood) and 66 billion unknowns per time step.

Limitations. Despite its capabilities, our method has several limitations. First, it is restricted to low Reynolds numbers and, therefore, cannot accurately approximate flows in large arteries. Secondly, the discretization of the RBCs is not adaptive; that is to say, all of the RBCs are approximated using the same number of points. Thirdly, we currently do not support arbitrary confined boundaries. Although we have previously discussed the underlying algorithms [25], the parallelization of such algorithms remains a challenge. Finally, the memory requirements of the method grow with the cube of the number of points per RBC. Alternatives exist, but the implications on performance are not yet well understood.

Related work on parallel algorithms for particulate flows and blood. Here, we focus on algorithms for particulate flows. Despite the engineering and scientific developments in understanding the complex behavior of particulate flows [24], it is only recently that algorithmic advances have allowed accurate 3D simulations of Stokes flows with hundreds of deformable particles using boundary integrals on a single CPU. Attempts to parallelize integral equation solvers have been restricted to spatially uniform particle distributions and have not scaled to a large number of cores. The main challenges are achieving *end-to-end concurrency* (i.e., parallelizing all algorithmic components to circumvent Amdahl's law), reducing the number of unknowns, especially for concentrated suspensions, and parallelization of the elliptic problem associated with the background flow.

Impressive simulations have been conducted using *fictitious/immersed boundary-like methods* [1], [3], [12], [15], [19], [32]. However, there is limited work ([14], [20], [28], [38]) in efficiently parallelizing these methods and no scalings on thousand-core machines. In blood rheology simulations, large-number-of-particles simulations include [22], in which a $50\mu\text{m}^2 \times 500\mu$m-capillary blood flow with 300 thousand rigid particles was modeled, and [11] in which a *dissipative particle dynamics* method was used to model a few thousand deformable RBCs. In [29], a method based on *lattice-Boltzmann methods* was used for particle simulations but the time-stepping was explicit and limited to rigid particles, with a few exceptions (e.g., [9]); however, Lattice Boltzmann methods require extremely small time steps due to numerical stiffness. Finally, *moving-mesh finite-element methods* [31] have been used but suffer from parallel scalability issues in the case of large 3D deformations [2], [6].

A different class of methods is based on *boundary integral equation* formulations. *Such formulations are ideal for Stokesian particulate flows since they discretize only the moving surfaces, which is much easier than discretizing the moving volume* [2]. Indeed, this approach has been successful in accurately modeling large numbers of deformable particles [5], [17], [24], [23], [40], [39]. In [41], 1200 deformable drops were modeled using an algorithmically scalable method. That implementation was sequential and required 120 CPU hours. Overall, limited work exists in parallelizing such methods. An exception is the parallel Stokes solver [21], but it did not use a fast-summation acceleration.

II. Formulation and Algorithms for Particulate Flows

Notation. Before we proceed with describing the kernels of MoBo let us introduce some notation. We use MATLAB's notation for linear algebra operations. We use upper-case letters for matrices and lower-case letters for vectors. We use upper-case bold-face letters for discretizations of infinite dimensional integral and differential operators, and lower-case bold-face letters to denote vectors and points in $\mathbb{R}^3$.

Mathematical formulation. The equations governing the dynamics of particulate flows in the free space $\mathbb{R}^3$ are the Stokes equation for the plasma and a differential equation for the forces on the membrane of an RBC. All of the RBCs are assumed to be filled with a Stokesian fluid that is the same as the surrounding fluid (henceforth "*fluid*").[3]

[3]The more general case in which the inside and outside fluid are different does not introduce any complications either mathematically or algorithmically. See [25].

Symbol	Definition
m	Number of points used to discretize the RBC surface
q	Spherical harmonics expansion order
n	Number of Red Blood Cells
p	Number of processors
$\mathbf{v}$	Velocity of the points
$\mathbf{F}$	DFT matrix operator
$\mathbf{P}_k$	k^{th} order Discrete Legendre transform
$\mathbf{S}, \mathbf{S}^{-1}$	Forward/inverse spherical harmonics transform

TABLE I: Index of frequently used symbols and operators.

Specifically, we have

$$-\Delta \mathbf{v} + \nabla p = 0 \quad \text{in } \mathbb{R}^3 \backslash \cup \gamma_k, \tag{2}$$

where $\mathbf{v}$ is the velocity of the fluid, p is the pressure, and γ_k is the interface between the k^{th} RBC and the surrounding fluid. This equation is subject to the constraints

$$[\![\mathbf{v}]\!] = 0, \quad [\![\mathbf{Tn}]\!] = \mathbf{f} \quad \text{on } \cup_k \gamma_k, \quad \text{div } \mathbf{v} = 0 \quad \text{in } \mathbb{R}^3, \tag{3}$$

where $[\![\cdot]\!]$ denotes a "jump", a difference between the inside and outside of the RBC membrane, $\mathbf{Tn}$ is the normal traction of the fluid and $\mathbf{f}$ is a force related to the elasticity of the cell membrane (i.e., the cell membrane's resistance to bending and stretching). The second equation states that the bending force of the interface is balanced by the forces of the fluid. The third equation states the incompressibility of the fluid. The first constraint merely implies that a point on the RBC's membrane moves with the same velocity as the surrounding fluid. Using this constraint, we get an equation for the evolution of the interface

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}(\mathbf{x}), \quad \text{for } \mathbf{x} \in \gamma_k, \text{ and all } k, \tag{4}$$

where $\mathbf{x}$ denotes the position vector of the points on the γ_k. As mentioned in the introduction, the velocity $\mathbf{v}$, at a point $\mathbf{x}$, can be decomposed into three components:

$$\mathbf{v}(\mathbf{x}) = \mathbf{v}_{local} + \mathbf{v}_{global} + \mathbf{v}_{background}, \tag{5}$$

As already mentioned, $\mathbf{v}_{local}$ depends on the shape of the individual RBC. However, precise expressions of the local and global velocities are beyond the scope of this paper and, due to space limitations, are omitted. See [34] for further details.

Suffice it to say that $\mathbf{v}_{local}$ at a point $\mathbf{x}$ on γ_k (the membrane of the k^{th} RBC) depends only on the shape of γ_k.

The global velocity is given by the evaluation of the Stokes kernel and has the following form:

$$\mathbf{v}_{global}(\mathbf{x}_k) = \sum_i \frac{1}{\rho_{ki}} \left(\mathbf{d}_i + \frac{(\mathbf{r}_{ki} \cdot \mathbf{d}_i)\mathbf{r}_{ki}}{\rho_{ki}^2} \right), \tag{6}$$

where $\mathbf{r}_{ki} = \mathbf{x}_k - \mathbf{x}_i$, $\rho_{ki} = |\mathbf{r}_{ik}|$, $\mathbf{d}_i$ is the given density at point $\mathbf{x}_i$, $\mathbf{r}_{ki} \cdot \mathbf{d}_i$ denotes the geometric dot product between these two vectors, and $|\cdot|$ denotes the vector norm in $\mathbb{R}^3$. The third term, $\mathbf{v}_{background}$ is the imposed background flow.

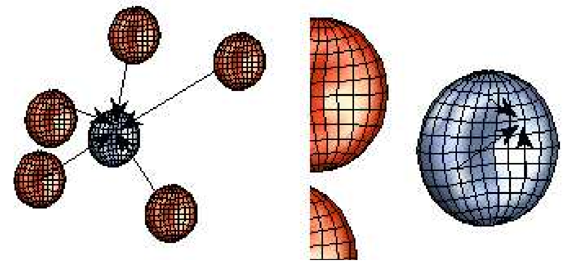


Fig. 2: AN EXAMPLE OF GLOBAL AND LOCAL INTERACTIONS. *The interaction between points on the surface of the blue membrane and points on the surfaces of the red membranes is global (left figure). The interaction between points on the surface of the same (blue) membrane is an example of local interactions (right figure).*

Representation of the surface. Let $U = \{(\theta, \phi) : \theta \in (0, \pi), \phi \in (0, 2\pi)\}$. Then, we denote a parametrization of the surface by $\mathbf{x} : U \rightarrow \mathbb{R}^3$. θ parametrizes the latitude and ϕ parametrizes the longitude. In this way, $\mathbf{x}$ can be represented in the spherical harmonics basis with spectral accuracy [34]. This spectral representation enables fast and accurate computation of high-order derivatives of $\mathbf{x}$ with respect to θ and ϕ. Such derivatives are required for the calculation of $\mathbf{v}_{local}$. (For example, $\mathbf{v}_{local}$ depends on the mean and Gaussian curvatures of the membrane, the normal vector and other geometric quantities.)

A. Overall algorithm and the main kernels

We will use a simple explicit Euler time-marching scheme. (In [34], we describe a semi-implicit second- and third- order algorithm.) Given a set of RBCs each one represented by its surface γ_k and a set of points on this surface $\mathbf{x}$ the algorithm proceeds as follows:

1) Compute $\mathbf{v}_{local}(\mathbf{x}, \gamma_k)$, for all $\mathbf{x} \in \gamma_k$ and all k;
2) Compute $\mathbf{v}_{global}$ using FMM;
3) Compute $\mathbf{v}_{background}$ using the user-supplied method;
4) Update the position using $\mathbf{x}_{new} = \mathbf{x} + \Delta t(\mathbf{v}_{local}(\mathbf{x}) + \mathbf{v}_{global}(\mathbf{x}) + \mathbf{v}_{background}(\mathbf{x}))$.

Here Δt is the time-step size in the explicit Euler scheme.

For the semi-implicit scheme described in [34], the first step is embarrassingly parallel across k (RBCs). For each cell, the complexity of computing $\mathbf{v}_{local}$ for all $\mathbf{x}$ on its surface is (q^6). The second step involves a communication-heavy all-to-all N-body calculation with the Stokes kernel. The complexity of the Stokes kernel was analyzed in [18]. We briefly summarize the main components of FMM in the next subsection.

B. Global interactions: FMM kernel

The FMM consists of two main steps, the octree construction phase and the evaluation phase. The evaluation phase consists of the computation of the far-field approximated interactions and the exact near-field direct interaction. Specifically, the evaluation phase involves the following substeps:

1) a bottom-up (post-order) tree-traversal to compute the multipole-moments approximation;

2) an arbitrary-order traversal to translate multipole-moments to local approximations;
3) a top-down (pre-order) tree traversal to accumulate all far-field interactions to the target points, the so-called "VXW-lists" calculation in FMM jargon; and
4) a near-neighbor exchange at the leafs to compute the near-range interactions, the so-called "U-list" calculation in FMM jargon.

In parallelizing FMM, the main challenges are the tree construction, and the communication involved in step (2), which in principle is local, but involves "fat" neighbors that can cause communication imbalance for highly nonuniform trees. Nevertheless, under reasonable assumptions on the distribution of RBCs, the overall complexity of the tree-construction phase is $\mathcal{O}(\frac{n}{p}\log\frac{n}{p}) + \mathcal{O}(p\log^2 p)$ and of the evaluation phase is $\mathcal{O}(\frac{n}{p}) + \mathcal{O}(\sqrt{p})$, where p is the number of MPI processes.

In [18], we proposed a novel tree-construction algorithm, a novel hypercubed-based reduction for the V-list calculation, and a hybrid GPU-MPI implementation for the U-list and V-list calculations. However, this approach left the multi-core CPUs that were driving the GPU spinning idle. In [8], we introduced a set of optimizations that can further accelerate the computations.

In section III, we report results from a different multithreading strategy. We employ a hybrid OpenMP-MPI-CUDA scheme in which we compute the dense interactions in parallel with the far-field interactions. The CPU sockets are responsible for the far-field computations (V-list) and the GPUs are responsible for the direct interactions (U-list). In addition, we have introduced several optimizations that are specific to our Kernel Independent FMM [37]

We use Streaming SIMD Extensions (SSE) technology, available in a number of modern CPUs, to speed up the floating point computations. In a nutshell, using SSE allows to perform basic arithmetic operations on small vectors of floating point numbers. Specifically, vectors can consist of either 4 single-precision floating point numbers or 2 double-precision floating point numbers (in any case vectors are 128 bit long). Arithmetic operations on different vector entries are performed by the CPU in parallel, thus speeding up the computation (in an ideal scenario) by a factor of 4 or 2, depending on the precision. We use SSE to accelerate the particle-to-particle interactions (specifically, the evaluation of the Stokes kernel). We replicate the data associated with each target point (e.g., x-coordinate) into 4 entries of an SSE vector, and load another SSE vector with x-coordinates of 4 different source points. Then we apply SSE vector subtraction to evaluate 4 differences in parallel, then we square these 4 differences in parallel and so forth. Eventually we obtain 4 potentials in an SSE register and sum them up. We use this approach for the source-to upward equivalent densities.

Finally, we use point-to-point interactions without precomputation in many parts of the algorithm for improve float-to-memory access rations and improved overall performance.

a) Repartitioning of Red Blood Cells: Partitioning RBCs among MPI processes is done using the FMM underlying data structure. This is necessary because after a few time steps the paritioning of RBCs does not match the optimal partitioning for the FMM and this results to excessive communication. The parallel FMM code that we are using [36], requires particles to be Morton-sorted not only locally on each MPI task but also across MPI tasks. That is, in order to apply FMM for computations, we first have to redistribute ("scatter") the points between tasks so that points become Morton-sorted, then evaluate the potentials and finally scatter the potentials back to the original layout of the points. When using multiple (say, tens of thousands) MPI tasks, the cost of these two scatters can become prohibitive, unless special measures are taken. We periodically re-distribute the RBCs between MPI tasks, so that the overall distribution of points is "close" to being Morton sorted, and thus the scatters are relatively inexpensive. Specifically, we use the partitioning of the space between MPI tasks produced by the last call to FMM. For each RBC, we determine preliminary "target MPI task" for each point of the RBC. Then we decide the final MPI task for the RBC by the voting procedure. For the actual data transfer we employ `MPI_Alltoallv()` call.

C. Local interactions: RBC physics kernels

The computation of $\mathbf{v}_{local}$ consists of several kernels. In the following, we discuss 9 kernels in which the majority of the computation takes place:

b) The spherical harmonics transform kernel: The spherical harmonics transform may be expressed in terms of matrix operations [7]. For a spherical harmonic expansion of order q, there are $2q$ points in the east-west direction and $q+1$ points on the north-south direction. Let X, Y, and Z denote matrices, each of size $2q \times (q+1)$, that hold the x-, y-, and z-coordinate components of the grid points, respectively. The points are stored in a "latitude-major order". Then, the k^{th} order spherical harmonic coefficients of X is given by

$$\hat{X}_k = \mathbf{P}_k \mathbf{W} (\mathbf{F}X)_k^T, \quad k = 0, \ldots, 2q, \tag{7}$$

where $\mathbf{F} \in \mathbb{R}^{2q\times 2q}$ denotes the discrete Fourier transform, $\mathbf{W} \in \mathbb{R}^{(q+1)\times(q+1)}$ is a diagonal matrix holding the Gaussian quadrature weights, $\mathbf{P}_k \in \mathbb{R}^{k\times(q+1)}$ is the k^{th} order associated Legendre transform and $\hat{X}_k \in \mathbb{R}^{k\times 1}$ is the vector of spherical harmonic coefficients of the k^{th} order. The same formula is also true for Y and Z. Considering the fact that q is rather small compared to the number of RBCs, it is best to perform both Fourier and Legendre transforms as matrix multiplications [10]. This also motivates the data structure for our implementation.

The inverse of spherical harmonics transform is given by

$$X = \mathbf{F}^T [\mathbf{P}_1^T \hat{X}_1 \; \ldots \; \mathbf{P}_{2q}^T \hat{X}_{2q}]^T. \tag{8}$$

Hereinafter, we formally denote the forward and inverse spherical transforms by $\mathbf{S}$ and $\mathbf{S}^{-1}$. When we have n surfaces, the complexity of the forward and inverse spherical harmonics transform is $\mathcal{O}(nq^3)$ and the depth is $\log q$.

In order to accelerate spherical harmonics transform, we represent the transform as a sequence of multiplications of *real* matrices, that is we use real DFT and not the complex one; we use BLAS or CUDA-BLAS for all the matrix multiplications; and we use column-wise (Fortran-style) storage for all the matrices.

The input data corresponding to different RBCs is packed into a $2q \times (q+1)n$ matrix. Each row of this matrix corresponds to a particular latitude across all RBC. Each column corresponds to a particular longitude on a particular RBC. Columns corresponding to the same RBC are grouped together (columns related to the first RBC are followed by the columns related to the second RBC, etc.)

We start the transform by multiplying the input matrix from the left by the DFT matrix, thus applying DFT to each column independently. Then we transpose the resulting matrix. Note that now each column of the transposed matrix corresponds to a particular frequency and one of the two possible functions (sine or cosine). We then treat each column of the transposed matrix as a $(q+1) \times n$ matrix (stored column-wise). Rows of this matrix correspond to different longitudes across all RBCs, and columns correspond to different RBCps. We then multiply this matrix from the left by an appropriate Legendre transform matrix, as given in Equation (7).

Note that in case of GPU computations (CUDA-BLAS), we perform both matrix multiplications and the transpose on GPU.

All the data is stored in a one dimensional array and depending on the size parameter passed to the BLAS kernels, its content can be interpreted as matrices of different sizes. Using Matlab's notation, we store the coordinates array for the k^{th} RBC as $C_k = [X(:)^T \; Y(:)^T \; Z(:)^T]$, and for all RBCs as $C = [C_1 \; \dots \; C_n]$. Using this structure, data can be streamed to BLAS subroutines for spherical harmonics transform and moving of the pole.

c) Pole rotation kernel for weakly-singular quadratures.: Given the surface C_k $(k = 1, \dots, n)$ and a target point (x, y, z) on the same surface, there exist a linear transformation $\mathbf{R} \in \mathbb{R}^{m \times m}$ such that the pole for the surface $\bar{C} = \mathbf{R}C$ is at (x, y, z). Note that the transformation $\mathbf{R}$ depends on the parametrization of the surface and the target point, but it has no dependency on the geometry of the surface. An example of this transformation is given in Figure 3. Let $\mathbf{R}_1, \dots, \mathbf{R}_{q+1}$ be the transformations with the target point as the grid points on the $\phi = 0$ meridian, then the transformation for other points on the θ_k latitude is a permutation of $\mathbf{R}_k$ [13].

In our simulation, we need to perform this rotation operation for all of the points on the surface of a RBC. It is clear that the complexity of a single rotation of the pole is $\mathcal{O}(q^4)$ and the memory requirement is $\mathcal{O}(q^5)$. There is another algorithm for the rotation of the pole that is based on the spherical harmonics expansion of the surface [13], and [35]. This algorithm reduces the complexity to $\mathcal{O}(q^5)$ and the memory requirements to $\mathcal{O}(q^4)$. Nonetheless, it is more involved and we refer the interested reader to the given references for a detailed explanation. It suffices to say that the second algorithm exposes more parallelism, but it involves multiple small matrix-matrix multiplication and it is inferior to the direct algorithm for problems of moderate size. Table II summarizes the result of our comparison between these algorithms.

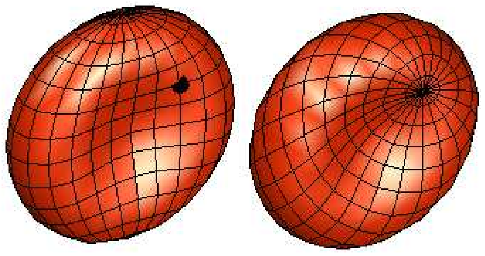


Fig. 3: THE TYPICAL BICONCAVE SHAPE. *The plot on the right is the same surface as in the one in the left, but the pole is moved to the point marked by the circle in the left figure.*

	Direct (cublas)		Rotation via spherical harmonics	
n	$q=6$	12	6	12
8	21.25	169.17	0.84	29.15
64	23.48	258.38	6.15	225.85
512	57.75	1360.07	47.85	1795.29
1024	98.02	2597.93	96.20	3589.00

TABLE II: The execution time (ms) on a Tesla GPU to move the pole to all points on the surface for different algorithms.

d) The kernel for the weakly-singular Stokes integral: The Stokes integral is singular at all points on the surface except for the pole. One method to evaluate the Stokes' integral at a point on the surface is to move the pole to that target point and evaluate the integral for that particular point [35]. In Algorithm 1, we outline the evaluation of Stokes integral. For n surfaces, the work is $\mathcal{O}(nq^6)$ and the depth is $\mathcal{O}(\log q)$.

Algorithm 1 Evaluation of singular Stokes integral. $\mathcal{O}(q^6)$
D is the input density and U is the evaluated potential.

```
for $k = 0$ to $q + 1$ do
    for $l = 0$ to $2q$ do
        $\mathbf{R} \leftarrow$ Permute $\mathbf{R}_k$ for the target $\phi_l$
        $\bar{C} = \mathbf{R}C, \bar{D} = \mathbf{R}D$,
        Evaluate $U_{kl}$ (by direct Stokes kernel)
    end for
end for
```

After the multiplications by the rotation matrices $\mathbf{R}$, the most costly kernel in our simulation is the direct Stokes evaluation kernel. For the CPU code, this kernel is implemented as SSE instructions.

e) The kernel for surface differentiation: The differentiation with respect to the ϕ parameter is a straightforward calculation using the DFT. But, differentiation with respect to θ needs extra care. With an abuse of notation, let $\mathbf{H}_k = d\mathbf{P}_k/d\theta$ and $\mathbf{W}_k = d^2\mathbf{P}_k/d\theta^2$. Then, to differentiate a function $G \in$

$\mathbb{R}^{2q\times(q+1)}$ on the surface with respect to the parameter θ we have

(i) $\hat{G} = \mathbf{S}G$,
(ii) $dG/d\theta = \mathbf{F}^T[\mathbf{H}_1^T\hat{G}_1 \ \ldots \ \mathbf{H}_{2q}^T\hat{G}_{2q}]^T$,
(iii) $d^2G/d\theta^2 = \mathbf{F}^T[\mathbf{W}_1^T\hat{G}_1 \ \ldots \ \mathbf{W}_{2q}^T\hat{G}_{2q}]^T$.

The complexity of these steps is the same as spherical harmonics transform and is $\mathcal{O}(q^3)$. With different matrices, the same kernel is used to evaluate the inverse spherical harmonics and all the derivatives.

f) Other computation kernels: From the programming prospective, several other utility kernels are required to facilitate computations on the surfaces, for instance, the computation of geometric properties of the RBC. These kernels include the geometric cross ($\mathbf{a}\times\mathbf{b}$), and dot ($\mathbf{a}\cdot\mathbf{b}$, $\mathbf{a},\mathbf{b}\in\mathbb{R}^3$) products, matrix transpose, scaling of vectors, etc.

g) The kernel for the FMM correction: The FMM algorithm indiscriminately calculates all the pairwise interactions between the source and target points. When we use FMM to compute $\mathbf{v}_{global}$, we also compute the interaction between the points that belong to the same RBC. But, these interactions need to be evaluated as *local* interactions. Therefore, we need calculate and subtract these erroneous terms form $\mathbf{v}_{global}$. The direct Stokes kernel, is used to evaluate the correction.

h) The kernel for surface reparametrization: In a typical simulation, the RBCs go thorough great distortion and the quality of the grid on their surface diminishes very fast. In Figure 4, we give an example of such distortion. In [35], we proposed a *tangential correction* algorithm to compensate for the distortions and maintain the grid quality. The reparametrization of the surface, involves calculation of the normal vector to the surface, mapping the surface to the spherical harmonics domain, filtering the high frequencies, and restricting the correction to the tangential direction on the surface (to keep the shape of the RBC intact). The complexity of this kernel is $\mathcal{O}(nq^3)$.

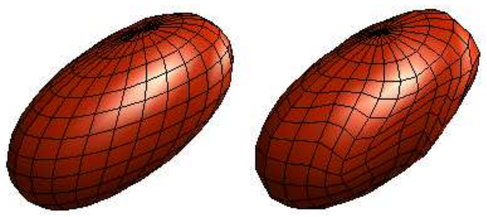


Fig. 4: Reparameterization of RBC. *The comparison between the quality of the grid for two simulation with and without the reparametrization, i.e, the redistribution of points on surface of an RBC in order to improve the numerical stability of the time-stepping scheme. Without such reparametrization, the distribution of pints on the surface becomes distorted as we march in time and the accuracy of the simulation is quickly lost.*

III. Scalability Results

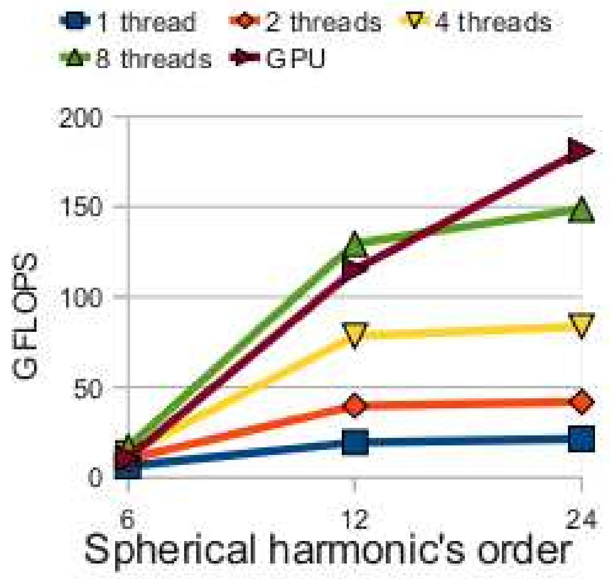
In this section, we describe the results from numerical experiments we conducted to investigate the performance characteristics of MoBo and the parallel scalability of our implementation across different architectures. Below, we summarize the different aspects of our numerical tests.[4]

Platforms and architectures: The large scale weak and strong scalability results have been obtained on the *Jaguar PF* platform at the National Center for Computational Sciences (UT/ORNL), a Cray XT5 system with 224,256 cores (2.6 GHz hex-core AMD opteron, 2GB/core) and a 3D-torus topology. Jaguar is ranked first in the top-500 list of supercomputers (www.top500.org) as of April of 2010. The GPU scalability results have been obtained on TeraGrid's *Lincoln* at the National Center for Supercomputing Applications (UIUC/NSF), a Dell cluster with NVIDIA Tesla S1070 accelerators, 1536 cores(Intel Harpertown/2.33 Ghz dual-socket quad-core 2GB/core), 384 GPUs (4GB/GPU), and InfiniBand (SDR) interconnect. The Nehalem tests where performed in an in-house 8-node cluster, with 16 sockets and one NVIDIA T10P-based GPU per socket. In all of experiments on Jaguar, we use one MPI process per socket and six threads per socket. Both $\mathbf{v}_{local}$ and $\mathbf{v}_{global}$ calculations have been multithreaded using OpenMP. Also, inn all of our GPU experiments, we use on MPI process per socket. We check different versions of combinations of multithreading.

Implementations and libraries: The code is written in C++ and the accelerator modules in CUDA. We use the `PETSc` [4] for profiling and certain parts of communication, and the `DENDRO` [26] package for the tree construction and repartitioning of the RBCs. The $\mathbf{v}_{global}$ module was implemented using our fast multipole implementation, which is based on the Kernel Independent Fast Multipole Method [18]. All of the kernels required for the calculation of $\mathbf{v}_{local}$ where implemented from scratch. We used the native CRAY `libsci` and `MKL BLAS` libraries on the Jaguar and the Intel boxes respectively.

Single node Experiments: To assess the performance of our code on a single node we perform various tests for the $\mathbf{v}_{local}$ and $\mathbf{v}_{global}$ calculations. The results are reported in Figures 5 for the $\mathbf{v}_{local}$ evaluation, and 6 for the $\mathbf{v}_{global}$ evaluation. Overall, we observe little difference between CPUs and GPUs although for higher resolutions, GPUs seem to outperform the x86 architectures. Recall that $\mathbf{v}_{global}$ and $\mathbf{v}_{local}$ utilize both CPU and GPUs. For example, the performance of $\mathbf{v}_{local}$ on a dual socket, dual GPU node exceeds 800 GFlops/s for $m = 12$. From Figure 6, we observe that the GPU accelerated version of FMM is roughly three times faster than the CPU-only thus, delivering a combined 60–70 Gflops/s per node for $\mathbf{v}_{global}$. The only data transfers between host and device is for the FMM evaluation in which the host collects the information from all RBCs and then invokes FMM. This is somewhat suboptimal. The solution is have a both GPU and CPU versions for all phases of the FMM and "treat" the GPU as a separate MPI process.

[4]In this submission, all of the FMM calculations for $\mathbf{v}_{global}$ were done in double precision, whereas all the calculations for $\mathbf{v}_{local}$ were done in single precision. For the final submission, both components of MoBo will support both double and single precision. With respect double precision, it is need to resolve the position of points in the RBCs for very skewed RBCs distributions.



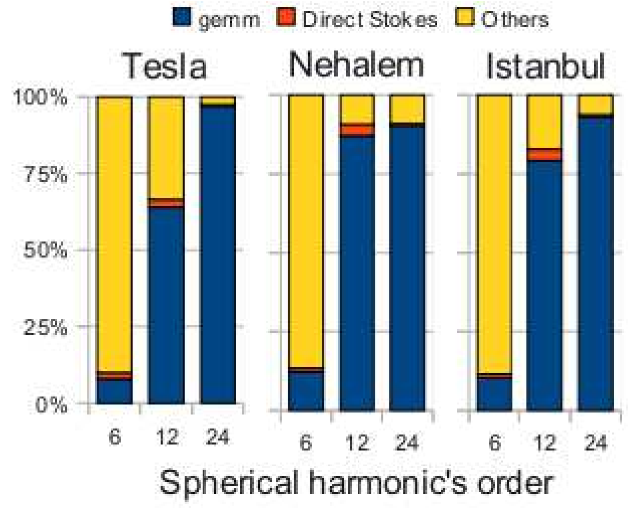


Fig. 5: SINGLE NODE FOR LOCAL INTERACTIONS. *The first figure shows the sustained FLOPS per second of the local kernel, on a Nehalem with different number of threads and Tesla. The second figure is the work share of the major components of the local kernel. For this figure, we used 8 OpenMP threads on Nehalem and 12 OpenMP threads on Istanbul.*

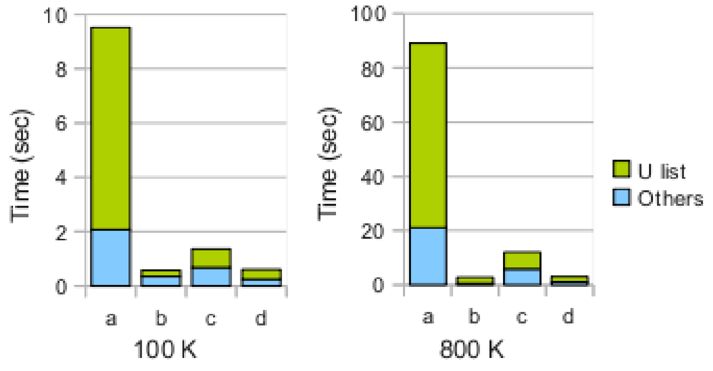
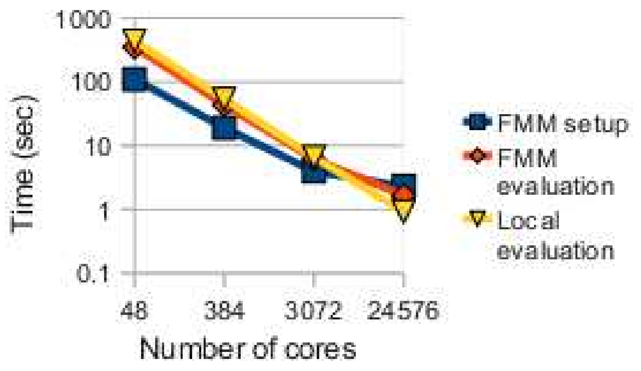


Fig. 6: SINGLE NODE RESULTS FOR THE GLOBAL INTERACTIONS. *In this figure we present results for the FMM code for various hybrid architecture setups: (a) One thread CPU only without SSE, (b) GPU only for both direct and near evaluations , (c) Four threads on CPU with SSE, (d) Four threads on the CPU with SSE for the V-list and asynchronous evaluation of U-list on the GPU. On both Istanbul and Nehalem architectures we observe 1.2 (m=6)–1.7(m=12) GFlops/s per core, for the overall FMM evaluation phase.*



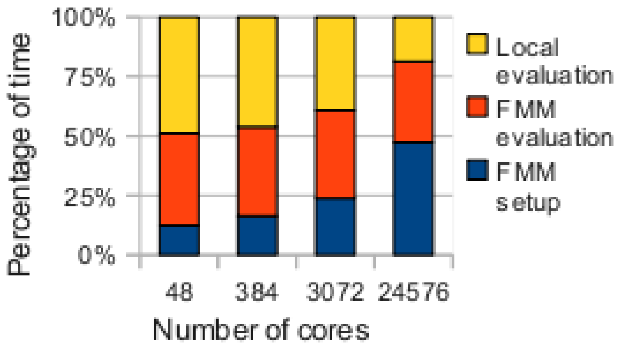


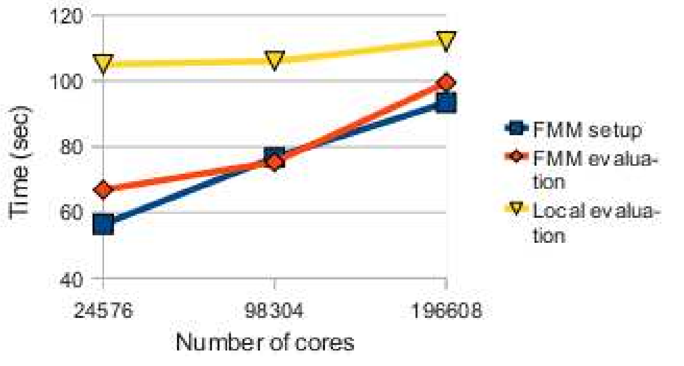
Fig. 7: STRONG SCALINGS ON JAGUAR PF. *The strong scalability result for 262144 vesicles, and total number of 22M grid points. There are 6 cores (and 6 OpenMP threads) per MPI process. The finest level of the octree is 9 and the coarsest is 3.*

MPI, strong scalability tests on Jaguar: The strong scalability results are reported in Figure 7. The problem size is 300,000 RBCs with 84 points per RBC, which corresponds to 25,000,000 unknowns. The strong scalability results demonstrate excellent speed up resulting in an unprecedented five seconds per time-step on 24,576 cores.

A. Jaguar

We get excellent speed up and we require less than 10 seconds per time step for 300,000 RBCs.

MPI, weak scalability tests on Jaguar: The weak scalability results are reported in Figure 8. The problem size (number of RBCs) per core is kept fixed to 8000 RBCs, again with 84 points per RBC for the line-distribution on the Poiseuille flow. We can observe the the calculation of $\mathbf{v}_{local}$ remains almost constant, whereas the cost of the tree-setup and $\mathbf{v}_{global}$ increase. This is due to several reasons. As we increase the problem size, the tree gets deeper and the cost per core increases. Also for such



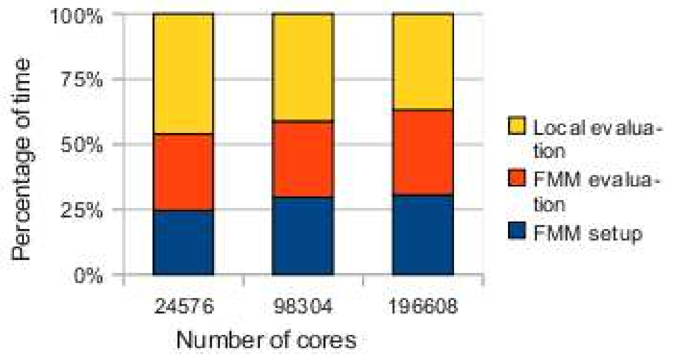


Fig. 8: WEAK SCALINGS ON JAGUAR PF. *The weak scalability of the simulation to 196,608 cores. We have chosen a line distribution of 8,000 RBCs with 84 points per RBC. We use one MPI process per socket and all of the six OpenMP threads in each MPI process. The finest to coarsest octree levels range form 24 to 4. In the largest, simulation, there are 200 million red blood cells and 66 billion unknowns. These results represent the average timings of four explicit Euler time steps.*

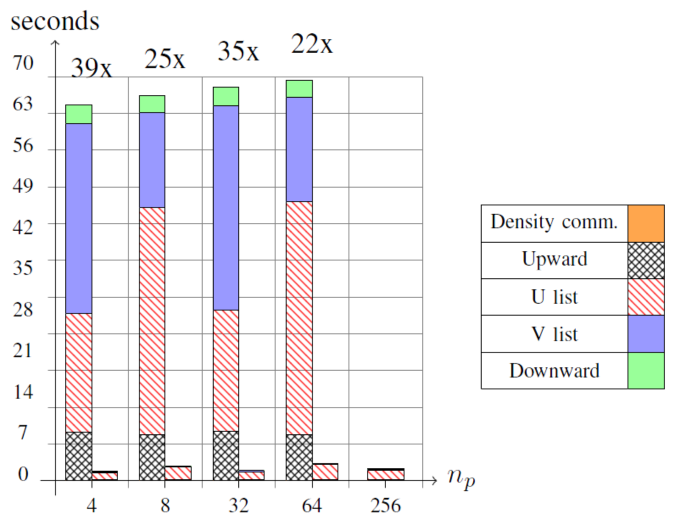


Fig. 9: GPU WEAK SCALING.. *Here we compare CPU-only with GPU/CPU configuration on up to 256 processes. For the largest run the total evaluation on 256 million points takes 2.2 secs (we did not run a CPU only example for the largest case). Throughout the computation, we maintain a 25X speed-up over a CPU-only implementation with only one thread per socket. When multithreading and vectorization is enabled, the differences become less pronounced as we can see in Figure 6. For the GPU runs, we use a shallower tree by allowing a higher number of points per box. In this way, we favor dense interactions over far-field computations. The former has a favorable computation/memory communication ratio and performs favorably on a GPU. In this examples, we used roughly 400 points per box for the GPU runs, and 100 points per box for the CPU runs. Both numbers were optimized for their respective architectures. We were able to maintain a 1.8-3 secs / evaluation for the GPU-based implementation. (This figure is reproduced from [18].)*

non-uniform trees it is difficult to load balance for all phases of FMM. The solution to these scaling problems is to employ the hypercube-like tree-broadcast algorithm we developed in $lashuk - biros - 09$ for all of the phases of FMM. (Currently it is used only in the post-order communication phase of the evaluation phase.) Nevertheless, we achieve excellent utilization. The $\mathbf{v}_{local}$ phase sustains over 18 GFlops/s per core (single precision) and the $\mathbf{v}_{global}$ phase sustains over1.2GFlops/per core (double precision). This results in over 0.7 Petaflops sustained performance.

GPU weak scalability results for FMM on Lincoln. We report these results in Figure 9. We only report results for the uniform distribution using 1M points per GPU. We use one socket per MPI process and one GPU per socket. Its socket has 4 cores, but at this point we are not using them. The results on GPUs are excellent on up to 256 processes/GPUs. We get over a 25X per core consistently and we were evaluate to solve a 256-million particle sum in 2.3 seconds for a total of approximately 8 TFlops/s. The CPU version using floating point throughout and its single core performance is optimized using blocking and optimized BLAS libraries for the per-octant calculations.

Red Blood Cell distributions and background flow: We test a line-like distribution of cells (Figure 10) on a Poiseuille background flow.[5] Roughly speaking, such background flow corresponds to an unperturbed laminar flow in a blood vessel. The results of having a line of cells exposed to such a Poiseuille flow are easy to informally "verify" visually. Also, such a flow results in a highly non-uniform distributions of points as the simulation horizon increases.

IV. CONCLUSIONS

We have presented MoBo, the first petascale code for the direct simulation of a multiphase model of blood. MoBo exposes and exploits concurrency at all stages of a complex multiphysics, multiscale code across several parallel programming paradigms. We showed that we can efficiently scale the different parts of

[5]More precisely, this is a "*pseudo-Poiseuille flow*", since we do not impose confinement boundary conditions around the cells. Rather, we impose a free-space velocity that corresponds to a Poiseuille flow.

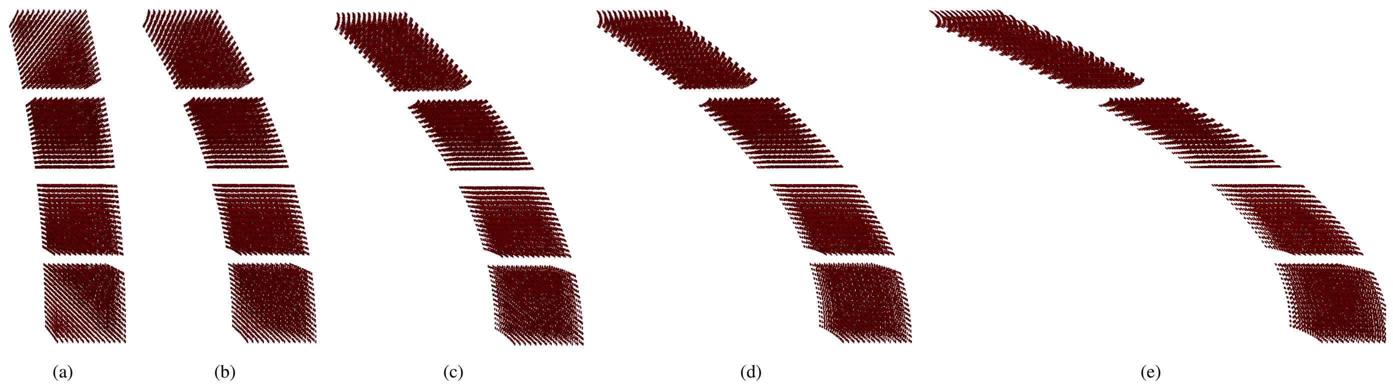
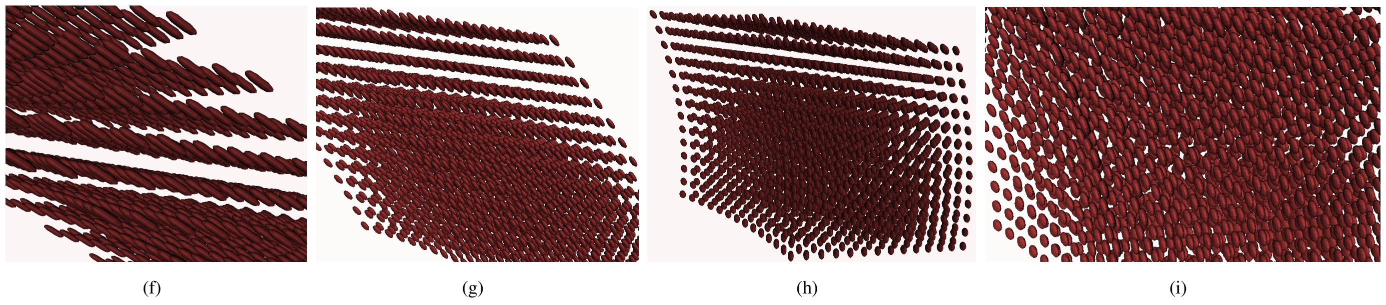



Fig. 10: SIMULATION RESULTS FOR 40,000 RBCS. *In this figure, we present results from a 40,000-RBC simulation with 84 points per RBC for a total 10,000,000 unknowns. In the top row (a–c), we can observe the alignment of the cells with the background Poiseuille flow as we advance in time. We can verify the need for non-uniform solvers. Every time step of this simulation, requires a Stokes solve at the extraordinarily complicated domain defined by the exterior and interior of the RBCs. In addition, the interfacial forces at each RBC are computed by inverting an integro-differential operator that involves fourth order derivatives. In the bottom row, we zoom in on different regions of the flow snapshot (c). We can observe the different deformations of the cells in different regions of the flow. For example, (f) and (g) depict cells from the upper tip (c) in which the shear rate is higher and the cells experience larger deformations. Subfigures (h) and (i) depict cells from the bottom left of (c), which is near the center of the Poiseuille flow and thus, the cells experience smaller deformations. The visualization was done on a single workstation using ParaView (www.paraview.org).*

the algorithm and we observe good scalability across different architectures.

For a single node performance of the $\mathbf{v}_{local}$ component we get roughly 100 GFlop/s for both CPUs and GPUs. This is not accidental. Our algorithm design and choices resulted in extensive use of GEMM routines–despite the complexity. In this way, we were able to deliver spectral accuracy while using only a small number of degrees of freedom per RBC (compare to the 1000s of degrees of freedom per RBC for dissipative particle dynamics and Lattice Boltzmann methods). For the global interaction we believe that the above 1GFlop/s *per core* for the FMM is quite remarkable given the complexity of the algorithm.

In our largest calculation on 196,608 cores, we achieved 0.7 PetaFlops for the multiphysics/multiscale problem of direct numerical simulation of blood flow. Let us emphasize that these results represent the worst case scenario with respect performance, as we use a very small number of points per cell. If we use an $m = 12$ spherical harmonics approximation for the RBCs the percentage of time spent in the $\mathbf{v}_{local}$ part of the calculation will further increase.

Taken together, MoBo opens the way for blood flow simulations of unprecedented fidelity. It enables the simulation of microfluidic and nanofluidic devices. Even the sequentially 2D version of MoBo, has already resulted in significant scientific discoveries [16].

ACKNOWLEDGMENTS.

This work was supported by the U.S. National Science Foundation grants CNS-0929947, OCI-0749285 and OCI-0749334. Computing resources on the TeraGrid systems were provided under the grants ASC070050N and MCA04N026. We would like to thank the NCCS and TeraGrid support staff, from whom we have received significant assistance.

REFERENCES

[1] G. Agresar, J.J. Linderman, G. Tryggvason, and K. G. Powell. An adaptive, Cartesian, front-tracking method for the motion, deformation and adhesion of circulating cells. *Journal Of Computational Physics*, 143(2):346–380, 1998.

[2] James F. Antaki, Guy E. Blelloch, Omar Ghattas, Ivan Malčević, Gary L. Miller, and Noel J. Walkington. A parallel dynamic-mesh Lagrangian method for simulation of flows with dynamic interfaces. In *Proceedings of Supercomputing 2000*, Dallas, TX, November 2000. ACM/IEEE.

[3] P. Bagchi. Mesoscale simulation of blood flow in small vessels. *Biophysical Journal*, 92(6):1858–1877, 2007.

[4] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc home page, 2001. http://www.mcs.anl.gov/petsc.

[5] Becker, L. E. and Shelley, M. J. Instability of elastic filaments in shear flow yields first-normal-stress differences. *Physical Review Letters*, 8719(19), 2001.

[6] George Biros, Lexing Ying, and Denis Zorin. A fast solver for the Stokes equations with distributed forces in complex geometries. *Journal of Computational Physics*, 193(1):317–348, 2003.

[7] John P. Boyd. *Chebyshev and Fourier Spectral Methods.* 1999.

[8] Aparna Chandramowlishwaran, Samuel Williams, Leonid Oliker, Ilya Lashuk, George Biros, and Richard Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Proceedings of IPDPS*, Atlanta, GA, 2010. IEEE Computer Society.

[9] J.R. Clausen, D.A. Reasor Jr, and C.K. Aidun. Parallel performance of a lattice-Boltzmann/finite element cellular blood flow solver on the IBM Blue Gene/P architecture. *Computer Physics Communications*, 2010.

[10] John B. Drake, Pat Worley, and Eduardo D'Azevedo. Algorithm 888: Spherical harmonic transform algorithms. *ACM Trans. Math. Softw.*, 35(3):1–23, 2008.

[11] W. Dzwinel, K. Boryczko, and D. A. Yuen. A discrete-particle model of blood dynamics in capillary vessels. *Journal Of Colloid And Interface Science*, 258(1):163–173, 2003.

[12] A. L. Fogelson and R. D. Guy. Platelet-wall interactions in continuum models of platelet thrombosis: formulation and numerical solution. *Mathematical Medicine And Biology-A Journal Of the IMA*, 21(4):293–334, 2004.

[13] Z. Gimbutas and L. Greengard. Short note: A fast and stable method for rotating spherical harmonic expansions. *J. Comput. Phys.*, 228(16):5621–5627, 2009.

[14] E. Givelberg and K. Yelick. Distributed immersed boundary simulation in titanium. *SIAM Journal On Scientific Computing*, 28(4):1361–1378, 2006.

[15] R. Glowinski, T.W. Pan, T.I. Hesla, D.D. Joseph, and J. Périaux. A fictitious domain approach to the direct numerical simulation of incompressible viscous flow pat moving rigid bodies: Application to particulate flow. *Journal of Computational Physics*, 169:363–426, 2001.

[16] Badr Kaoui, George Biros, and Chaouqi Misbah. Why do red blood cells have asymmetric shapes even in a symmetric flow? *Physical Review Letters*, 103(18):188101, 2009.

[17] Michael R. King and Daniel A. Hammer. Multiparticle adhesive dynamics: Hydrodynamic recruitment of rolling leukocytes. *PNAS*, 98(26):14919–14924, 2001.

[18] I. Lashuk, A. Chandramowlishwaran, T-A. Nguyen H. Langston, R. Sampath, A. Shringarpure, R. Vuduc, D. Zorin L. Ying, and G. Biros. A massively parallel adaptiven fast-multipole method on heterogeneous architectures. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2009. IEEE Press.

[19] Y. L. Liu and W. K. Liu. Rheology of red blood cell aggregation by computer simulation. *Journal Of Computational Physics*, 220(1):139–154, 2006.

[20] D. M. McQueen and C. S. Peskin. Shared-memory parallel vector implementation of the immersed boundary method for the computation of blood flow in the beating mammalian heart. *Journal Of Supercomputing*, 11(3):213–236, 1997.

[21] Nhan Phan-Thien, Ka Yan Lee, and David Tullock. Large scale simulation of suspensions with PVM. In *Proceedings of SC97*, The SCxy Conference series, San Jose, CA, November 1997. ACM/IEEE.

[22] I. V. Pivkin, P. D. Richardson, and G. Karniadakis. Blood flow velocity effects and role of activation delay time on growth and form of platelet thrombi. *Proceedings Of The National Academy Of Sciences Of The United States Of America*, 103(46):17164–17169, 2006.

[23] C. Pozrikidis. Numerical simulation of the flow-induced deformation of red blood cells. *Annals of Biomedical Engineering*, 31(10):1194–1205, 2003.

[24] Costas Pozrikidis. Interfacial dynamics for Stokes flow. *Journal of Computational Physics*, 169:250–301, 2001.

[25] Abtin Rahimian, Shravan K. Veerapaneni, and George Biros. A fast algorithm for simulation of locally inextensible vesicles suspended in an arbitrary 2d domain. *Journal of Computational Physics*, 2010. in press.

[26] Rahul Sampath, Santi S. Adavani, Hari Sundar, Ilya Lashuk, and George Biros. DENDRO home page, 2008.

[27] Rahul S. Sampath, Santi S. Adavani, Hari Sundar, Ilya Lashuk, and George Biros. Dendro: parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[28] K. M. Singh and J. J. R. Williams. A parallel fictitious domain multigrid preconditioner for the solution of Poisson's equation in complex geometries. *Computer Methods In Applied Mechanics And Engineering*, 194(45-47):4845–4860, 2005.

[29] C. H. Sun and L. L. Munn. Particulate nature of blood determines macroscopic rheology: A 2-D lattice Boltzmann analysis. *Biophysical Journal*, 88(3):1635–1645, 2005.

[30] Hari Sundar, Rahul Sampath, Christos Davatzikos, and George Biros. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Proceedings of SC2007*, The SCxy Conference series in high performance networking and computing, Reno, Nevada, 2007. ACM/IEEE.

[31] T. E. Tezduyar and A. Sameh. Parallel finite element computations in fluid mechanics. *Computer Methods In Applied Mechanics And Engineering*, 195(13-16):1872–1884, 2006.

[32] G. Tryggvason, B. Bunner, A. Esmaeeli, and N. Al-Rawahi. Computations of multiphase flows. In *Advances In Applied Mechanics, Vol 39*, volume 39 of *Advances In Applied Mechanics*, pages 81–120. Academic Press Inc, San Diego, 2003.

[33] Shravan K. Veerapaneni, Denis Gueyffier, George Biros, and Denis Zorin. A numerical method for simulating the dynamics of 3d axisymmetric vesicles suspended in viscous flows. *Journal of Computational Physics*, 228(19):7233–7249, 2009.

[34] Shravan K. Veerapaneni, Abtin Rahimian, , George Biros, and Denis Zorin. A fast algorithm for simulating vesicle flows in three dimensions. 2010. Submitted for publication.

[35] Shravan K. Veerapaneni, Abtin Rahimian, George Biros, and Denis Zorin. A fast algorithm for simulating vesicle flows in three dimensions. *In preparation*.

[36] Lexing Ying, George Biros, Harper Langston, and Denis Zorin. KIFMM3D: The kernel-independent fast multipole (FMM) 3D code. GPL license.

[37] Lexing Ying, George Biros, Denis Zorin, and Harper Langston. A new parallel kernel-independent fast multiple algorithm. In *Proceedings of SC03*, The SCxy Conference series, Phoenix, Arizona, November 2003. ACM/IEEE.

[38] L. T. Zhang, G. J. Wagner, and W. K. Liu. A parallelized meshfree method with boundary enrichment for large-scale CFD. *Journal Of Computational Physics*, 176(2):483–506, 2002.

[39] H. Zhao, A.H.G. Isfahani, L.N. Olson, and J.B. Freund. A Spectral Boundary Integral Method for Flowing Blood Cells. *Journal of Computational Physics*, 2010.

[40] Alexander Z. Zinchenko and Robert H. Davis. An efficient algorithm for hydrodynamical interaction of many deformable drops. *Journal of Computational Physics*, 157:539–587, 200.

[41] A.Z. Zinchenko and R. H. Davis. Large-scale simulations of concentrated emulsion flows. *Philosophical Transactions Of The Royal Society Of London Series A-Mathematical Physical And Engineering Sciences*, 361(1806):813–845, 2003.