

Improving the Performance of Dynamical Simulations Via Multiple Right-Hand Sides

Xing Liu Edmond Chow

School of Computational Science and Engineering
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, 30332, USA
xing.liu@gatech.edu, echow@cc.gatech.edu

Karthikeyan Vaidyanathan Mikhail Smelyanskiy

Parallel Computing Lab
Intel Corporation
Santa Clara, California, 95054, USA
{karthikeyan.vaidyanathan, mikhail.smelyanskiy}@intel.com

Abstract—This paper presents an algorithmic approach for improving the performance of many types of stochastic dynamical simulations. The approach is to redesign existing algorithms that use sparse matrix-vector products (SPMV) with single vectors to instead use a more efficient kernel, the generalized SPMV (GSPMV), which computes with multiple vectors simultaneously. In this paper, we show how to redesign a dynamical simulation to exploit GSPMV in way that is not initially obvious because only one vector is available at a time. We study the performance of GSPMV as a function of the number of vectors, and demonstrate the use of GSPMV in the Stokesian dynamics method for the simulation of the motion of macromolecules in the cell. Specifically, for our application, we find that with modern multicore Intel microprocessors in clusters of up to 64 nodes, we can typically multiply by 8 to 16 vectors in only twice the time required to multiply by a single vector. After redesigning the Stokesian dynamics algorithm to exploit GSPMV, we measure a 30 percent speedup in performance in single-node, data parallel simulations.

Index Terms—sparse matrix-vector product (SPMV); iterative methods; block Krylov subspace methods; SIMD (SSE/AVX); Stokesian dynamics; Brownian dynamics; biological macromolecular simulation

I. INTRODUCTION

The performance of large-scale scientific computations largely depends on the choice of numerical algorithms and the efficiency of these algorithms on high-performance hardware. Thus, much research has been devoted to developing new algorithms or adapting existing ones to have better performance characteristics. Common strategies include reducing communication and synchronization requirements in the parallel context, as well as adapting algorithms rich in parallel work for specific architectures such as GPUs. These new algorithms usually come with tradeoffs, such as increased complexity, and possibly poorer numerical properties. The overall goal, however, is to reduce the throughput time of a scientific application. In this paper, we present an approach for speeding up certain types of dynamical simulation codes by redesigning their algorithms to use multiple right-hand sides. These new algorithms can exploit sparse matrix-vector products with multiple vectors, which run much more efficiently than sparse matrix-vector products with a single vector.

The sparse matrix-vector product (SPMV) is a common kernel in numerical simulation codes. The performance of SPMV on various architectures has been studied over many dozens of papers and many techniques, such as ordering and blocking, have been suggested for improving perfor-

mance [38], [29], [36]. In recent years, numerous methods have been invented to improve SPMV by reducing its bandwidth requirements, such as Compressed Sparse Blocks [8] and Bitmasked Register Blocks [7]. Although many optimizations have been intensively studied, SPMV is generally known for poor performance on modern CPUs. Studies have shown the best performance to be about 30% of peak CPU flop rates [38], [14], [24].

The performance of a very similar kernel has a much higher flop rate. Gropp, Kaushik, Keyes, and Smith [16] observed that a “generalized” SPMV (GSPMV), which multiplies a sparse matrix by a *block* of vectors *simultaneously*, can be performed in little more time than a traditional SPMV with a single vector. This result is easily seen from the fact that the memory bandwidth cost of accessing the matrix in DRAM is amortized over many vectors. In 1999, when the paper appeared, the rule of thumb was that one could multiply by *four* vectors in about 1.5 times the time needed to multiply by a single vector. Today, given the well-known growing imbalance between memory access and computation rates, the incremental cost of additional vectors is much smaller. Just like the fact that flops are becoming “free,” additional vectors for SPMV are also becoming free.

That paper by Gropp et al. [16] was likely the first to promote the use of algorithms that can make use of multiple vectors with each SPMV. Although no specific algorithms were identified in that paper, there are obvious applications where multiple vectors can be exploited. For example, in a finite element analysis where solutions for multiple load vectors, or more generally, “multiple right-hand sides” are desired, it is natural to use a block iterative solver, where each iteration involves an SPMV with a block of vectors. Such iterative methods have been avoided because of numerical issues that can arise [27], but these methods can be expected to gain more attention with the increasing performance advantage of GSPMV over single-vector SPMV. The importance of block iterative solvers is increasing as well, given the increasing number of applications in which multiple right-hand sides occur, for example, in applications of uncertainty quantification, where solutions for multiple perturbed right-hand sides are desired.

In the above applications, the use of GSPMV with a block iterative solver is natural because all the right-hand side vectors are available at the same time. It is not clear, however, how to use GSPMV when the right-hand sides are only available sequentially, i.e., one after another. This is the

common situation in dynamical simulations where, at each time step, a single right-hand side system is solved, and the solution of this system must be computed before the system at the next time step can be constructed.

In this paper, we present and test a novel algorithm that can exploit GSPMV for many types of dynamical simulations, even though the right-hand sides are only available sequentially. The main idea is to set up and solve an auxiliary system of equations with multiple right-hand sides; the solution to this auxiliary system provides good initial guesses for the original systems to be solved iteratively at each time step. Solving the auxiliary system is extra work, but it can be done very efficiently using GSPMV, and it is piggy-backed onto a solve which must be performed anyway, leading to an overall reduced computation time. The algorithm can be regarded as an instance of a technique or approach that is applicable to other situations.

In addition to presenting the algorithm above, another purpose of this paper is to show experimentally and with some simple analyses the performance advantages of GSPMV compared to SPMV. We discuss how to implement GSPMV efficiently with the SIMD (e.g., SSE/AVX) capabilities of general-purpose processors. Moreover, we study the performance of GSPMV in the distributed memory case.

We have implemented our algorithm in a Stokesian dynamics code which is being developed with the long term goal of simulating the motion of proteins and other macromolecules in their cellular environment. Section II provides background on the Stokesian dynamics method. Our algorithm is presented in Section III. Section IV presents a performance analysis of GSPMV, in particular for matrices arising in our application. In Section V, we test our approach by demonstrating the use of GSPMV in a Stokesian dynamics code. The results here are suggestive of results of applying our approach to other types of dynamical simulations. Section VI concludes the paper. Like [16], we hope that this paper will encourage exploration into developing algorithms that can use efficient kernels that operate on multiple vectors simultaneously.

II. BACKGROUND ON STOKESIAN DYNAMICS

A. General Principles

The algorithmic work in this paper is motivated by and studied in the context of the Stokesian dynamics (SD) method [4], [5]. We use this method for the simulation of the motion of biological macromolecules in solvent, but SD may be used in many applications. In SD simulations, the particles may be colloids, polymers, proteins, or other macromolecules in environments where the inertial forces are much smaller than the inter-particle forces, i.e., the particle Reynolds number is small. Of scientific and engineering interest are the macroscopic properties of the particle motion, such as average diffusion constants, that arise from the microscopic motions of the particles. We are interested in large-scale SD simulations, for example, those involving upwards of one million particles.

In SD simulations, the macromolecules are modeled as spherical particles of possibly varying radii. At each time step, like in other particle simulation methods, forces on the particles are computed and then the particle positions

are updated. For macromolecules in solvent, it is important to accurately model the hydrodynamic forces, that is, the forces mediated by the solvent on one particle due to the motion of other particles. Hydrodynamic forces are long range, varying as $1/r$, where r is the inter-particle separation. Particles that are nearly touching, however, also experience a strongly repulsive, short-range hydrodynamic force, called the “lubrication” force. SD accurately models both long- and short-range hydrodynamic forces. This is in contrast to the well-known Brownian dynamics (BD) method [11] which cannot accurately model short-range forces, and has thus been used only to study relatively dilute systems. SD, however, is able to study more closely-packed, high volume fraction systems, such as the crowded macromolecular environment of the cell [1]. This capability, however, makes SD much more computationally demanding than BD.

Biological molecules also experience stochastic forces corresponding to random collisions with molecules of the solvent. Since the particles of the solvent are not modeled explicitly, a Gaussian noise vector with a configuration-dependent correlation is used to simulate this “Brownian” force in SD (as in BD). This adds a significant complexity to a SD simulation. In addition, other forces can be incorporated, such as bonded forces for simulating long-chain molecules as a bonded chain of particles.

B. Governing Equations

The governing equation for particle simulations with Brownian interactions is the Langevin equation,

$$M \frac{d^2 r}{dt^2} = f^H + f^B + f^P$$

where r is a $3n$ -dimensional vector containing the three components of position of n particles, M is a mass matrix, and f^H , f^B , and f^P are the hydrodynamic, Brownian, and other external or inter-particle forces, respectively. We note that in this paper, we will use an approximation that neglects the rotation of the particles. The Langevin equation is simply a modification of Newton’s equation of motion to include the stochastic term f^B . The equation also often implies that r contains a reduced number of degrees of freedom, for instance r does not contain degrees of freedom due to the solvent, as the solvent is modeled by f^B .

In SD simulations, the inertial forces are small and thus particle mass can be neglected, i.e., $M d^2 r / dt^2 = 0$. The hydrodynamic force on a particle is dependent on the positions and velocities of *all* other particles; f^H takes the form

$$f^H = R(r) \left(\frac{dr}{dt} - u^\infty \right)$$

where $R(r)$ is a “resistance” or friction matrix that depends on the particle configuration, and u^∞ is the velocity of the bulk flow at the position of the particles. In this paper, we will use $u^\infty = 0$ without loss of generality. The Brownian force f^B is a Gaussian random vector with mean zero and covariance proportional to $R(r)$. The latter requirement is due to the fluctuation-dissipation theorem of statistical physics which relates fluctuations in the Brownian force with friction. Finally, in this paper, and for our simulations, $f^P = 0$.

The matrix R (dropping the dependence on r in our notation) describes the relationship between the hydrody-

dynamic forces and the velocities of a system of particles. The exact relationship involves solving the Stokes equations for multiple particles. In the Stokesian dynamics method, by contrast, R is constructed by superimposing the analytical solutions for two spherical particles in Stokes flow to approximate the multi-particle solution. Separate analytical solutions are provided for the long- and short-range hydrodynamic interactions, resulting in

$$R = (M^\infty)^{-1} + R_{\text{lub}}$$

where the first component is a dense matrix representing the long-range hydrodynamic interactions, and the second component is a sparse matrix representing the short-range lubrication interactions [10]. Structurally, R and its two components are block matrices, with blocks of dimension 3×3 . Each block represents the interaction between two particles. The blocks in M^∞ are the Oseen or Rotne-Prager-Yamakawa tensors [30], [39]; the blocks in R_{lub} are tensors coming from lubrication theory [21], [23]. It is these tensors that make R dependent on the particle positions and radii. We further adjust R_{lub} to project out the collective motion of pairs of particles [9]. With these choices, M^∞ is symmetric positive definite and R_{lub} is symmetric positive semidefinite.

We will see in the next section that each time step involves solves with the matrix R . For small problems, a Cholesky factorization is used; for the large problems in which we are interested, iterative solution methods have been suggested [34], [33], [35]. Iterative methods involve matrix-vector multiplies with R and, for efficiency, must multiply the dense component of the matrix R using fast algorithms such as particle-mesh Ewald (PME) [33], [17], [31]. Like GSPMV, such algorithms may also exploit multiple vectors for efficiency. In this paper, we will only study the efficiency of GSPMV and leave the study of PME with multiple vectors for future work. We thus use an alternative, sparse approximation to R proposed by [34]

$$R = \mu_F I + R_{\text{lub}}$$

which is applicable when the particle interactions are dominated by lubrication forces. The term $\mu_F I$ is a ‘‘far-field effective viscosity’’ with the parameter μ_F chosen depending on the volume fraction of the particles [34]. We use a slight modification of this technique to account for different particle radii.

C. Method of Simulation

In SD simulations where $f^p = 0$, the hydrodynamic forces balance the Brownian forces. The governing equation to be solved is

$$R(r) \frac{dr}{dt} = -f^B$$

which is, notably, a differential equation of first order. Although this problem is not smooth due to time-fluctuations in f^B , a second-order integrator must be used because of the configuration dependence of R ; a first-order integrator makes a systematic error corresponding to a mean drift, $\nabla \cdot R^{-1}$, see [11], [12], [15]. (For the Oseen and Rotne-Prager-Yamakawa tensors, the gradient with respect to r is zero, making the second-order method unnecessary.) For example, the explicit midpoint method may be used, which requires

two matrix solves at each time step,

$$\text{Solve } R(r_k) u_k = -f_k^B \quad (1)$$

$$\text{Compute } r_{k+1/2} = r_k + \frac{1}{2} \Delta t u_k \quad (2)$$

$$\text{Solve } R(r_{k+1/2}) u_{k+1/2} = -f_k^B \quad (3)$$

$$\text{Compute } r_{k+1} = r_k + \Delta t u_{k+1/2} \quad (4)$$

where k is a time index and u_k represents the velocity vector at time index k . The time step size Δt is chosen such that it is larger than the Brownian relaxation time, but small enough so that particles do not overlap. For our experiments, we use a modification of the midpoint method which helps avoid particle overlaps at the intermediate configuration [2]. The most costly steps of this procedure are the solves with the resistance matrices.

Another costly component of the SD method is the computation of the Brownian force vector with the proper covariance. This vector may be computed as $f^B = Lz$ (neglecting proportionality constants) where z is a standard normal vector and where L satisfies $R = LL^T$. Since R is symmetric and positive definite, the usual approach is to compute L as the lower-triangular Cholesky factor of R . This approach, however, is impractical or at least very costly for large problems. An alternative is to compute $S(R)z$, where $S(R)$ is a shifted Chebyshev polynomial in the matrix R which approximates the square root [13]. The matrix $S(R)$ itself is never computed, and each computation of a matrix polynomial times a vector only requires matrix-vector product operations with the matrix R . This is particularly advantageous when R is sparse. We use the Chebyshev approach in our simulations.

A summary of the algorithm at each time step is shown in Alg 1. In the following, let R_k denote $R(r_k)$, and let z_k denote the standard normal vector generated for step k .

Algorithm 1: SD Algorithm for one time step.

- 1 Construct $R_k = \mu_F I + R_{\text{lub}}(r_k)$
 - 2 Compute $f_k^B = S(R_k)z_k$
 - 3 Solve $R_k u_k = -f_k^B$
 - 4 Compute $r_{k+1/2} = r_k + \frac{1}{2} \Delta t u_k$
 - 5 Solve $R_{k+1/2} u_{k+1/2} = -f_k^B$
 - 6 Update $r_{k+1} = r_k + \Delta t u_{k+1/2}$
-

Many SD implementations use a Cholesky factorization of R for computing f^B and for solving the systems in steps 3 and 5. An important advantage of this is because the Cholesky factor computed for step 2 can be reused for step 3. A further optimization which we have used, but which does not appear to have been used elsewhere, is to solve the system in step 5 using the same Cholesky factor combined with a simple iterative method, such as ‘‘iterative refinement.’’ Combined with an initial guess which is the solution from step 3, only a very small number of iterations are needed for convergence. Thus only one Cholesky factorization, rather than two, is needed per time step. Cholesky factorizations, however, are not practical for very large problems. In the remainder of this paper, we focus on the use of iterative methods and thus sparse matrix-vector multiplications to carry out SD simulations.

III. EXPLOITING MULTIPLE RIGHT-HAND SIDES

The SD method requires the solution of a sequence of related linear systems with matrices R_k which slowly evolve in time as the particles slowly evolve in time. A number of solution techniques for sequences of linear systems can take advantage of the fact that the matrices are slowly varying. The most obvious technique is to invest in constructing a preconditioner that can be reused for solving with many matrices. As the matrices evolve, the preconditioner is recomputed when the convergence rate has sufficiently degraded. A second technique is to “recycle” components of the Krylov subspace from one solve to the next [28] to reduce the number of iterations required for convergence. A third technique is to use the solution of the previous system as the initial guess for the current system being solved. This is applicable when the solution itself is a slowly varying quantity as the sequence evolves.

At each SD time step, two linear systems must be solved which have the same right-hand sides and which have matrices that are slightly perturbed from each other. The simplest technique for exploiting these properties is to use the solution of the first linear system as the initial guess for the iterative solution of the second linear system.

Different time steps, however, have completely different right-hand sides. As already mentioned, these right-hand sides are in fact random with a multivariate normal distribution. At first glance, it thus does not appear possible that an initial guess is available to aid solving the first linear system of each time step.

We now, in fact, present a way to construct initial guesses for these systems in an efficient way. At two consecutive time steps, k and $k+1$, the linear systems to be solved are

$$R_k u_k = S(R_k) z_k \quad (5)$$

$$R_{k+1} u_{k+1} = S(R_{k+1}) z_{k+1} \quad (6)$$

where an initial guess for the second system is desired. In our approach, instead of solving the first system, the following system, which augments the first system with an additional right-hand side, is solved instead:

$$R_k [u_k \ u'_{k+1}] = S(R_k) [z_k \ z_{k+1}] \quad (7)$$

This multiple right-hand side system is solved with a block iterative method. The critical point is that this solve is expected to cost little more than the solve of the original system with a single right-hand side due to the use of GSPMV operations. Since R_{k+1} is close to R_k and $S(R_{k+1})$ is close to $S(R_k)$, the solution u'_{k+1} is an initial guess for the second system (6). The hope is that the number of iterations required to solve the second system is now reduced compared to the extra cost of constructing and solving (7) with the additional right-hand side.

The above procedure is of course extended to as many right-hand sides as is profitable. Thus the solution of one augmented system with m right-hand sides at the beginning of m time steps produces the solution for the first of these time steps and initial guesses for the following $m-1$ time steps. The parameter m may be larger or smaller depending on how R_k evolves and on the incremental cost of GSPMV for additional vectors. We refer to m as the *number of right-*

hand sides.

A summary of the algorithm for m time steps is shown in Alg 2. In the following, we call this algorithm the *Multiple Right-Hand Sides (MRHS) algorithm*. The algorithm requires a vector of initial positions, r_0 . Let $U = [u_0, \dots, u_{m-1}]$ and $Z = [z_0, \dots, z_{m-1}]$. In step 2, note that a GSPMV is also used for constructing the right-hand sides, F^B . We use k to denote the index $0, \dots, m-1$.

Algorithm 2: MRHS Algorithm for m time steps.

- 1 Construct $R_0 = \mu_F I + R_{\text{lub}}(r_0)$
 - 2 Compute $F^B = S(R_0)Z$
 - 3 Solve augmented system $R_0 U = F^B$
 - 4 Compute $r_{1/2} = r_0 + \frac{1}{2} \Delta t u_0$
 - 5 Solve $R_{1/2} u_{1/2} = -f_0^B$ using solution u_0 from step 3 as initial guess
 - 6 Update $r_1 = r_0 + \Delta t u_{1/2}$
 - 7 **for** $k \leftarrow 1$ **to** $m-1$ **do**
 - 8 Construct $R_k = \mu_F I + R_{\text{lub}}(r_k)$
 - 9 Compute $f_k^B = S(R_k)z_k$
 - 10 Solve $R_k u_k = -f_k^B$ using u_k from step 3 as initial guess
 - 11 Compute $r_{k+1/2} = r_k + \frac{1}{2} \Delta t u_k$
 - 12 Solve $R_{k+1/2} u_{k+1/2} = -f_k^B$ using solution from step 10 as initial guess
 - 13 Update $r_{k+1} = r_k + \Delta t u_{k+1/2}$
 - 14 **end**
-

IV. GENERALIZED SPARSE MATRIX-VECTOR PRODUCTS WITH MULTIPLE VECTORS

In order to fully understand the performance potential of algorithms using multiple vectors (or multiple right-hand sides), we need to better understand the performance of GSPMV. In this section we study this performance experimentally and with a simple analytical model. It is important to study optimized implementations of GSPMV because it would be these that are used in practice. We use standard performance optimizations for this purpose, but producing a general, highly-optimized implementation of GSPMV is outside the scope of this paper. In particular, we do not exploit any symmetry in the matrices.

A. Performance Optimizations for GSPMV

1) *Single-Node Optimizations*: There is a substantial literature exploring numerous optimization techniques for SPMV, i.e., the single-vector case. Vuduc [37] provides a good overview of these techniques. More recently, the performance of various SPMV algorithms has been evaluated by several groups [38], [3]. They cover a wide range of matrices, different storage formats, and various types of hardware. There also exist optimized GSPMV implementations. The approaches of these implementations are generally extensions of existing methods used for SPMV or SPMM (sparse matrix-matrix multiply). For instance, Im [18] extended register blocking and cache blocking methods to handle multiple vectors in GSPMV. Lee et al. [25] improved GSPMV by reducing its memory traffic. The method they used is an

extension of the vector blocking method, which was first used in SPMM.

We applied several well-known SPMV optimizations to GSPMV, including thread blocking and the use of SIMD. We also implemented TLB and cache blocking optimizations [26]. However, use of large pages made TLB blocking unnecessary for all but unrealistically large number of vectors. We have not used register blocking [18] due to the fact that our matrices already have natural 3×3 block structure. We store the m vectors in row-major format to take advantage of spatial locality.

While there exist a variety of sparse storage formats, in this work we focus on the widely-used Block Compressed Row Storage (BCRS) format, due to the known block structure of our matrices. Similar to the CSR format, BCRS requires three arrays: an array of non-zero blocks stored row-wise, a column-index array which stores the column index of each non-zero block, and a row pointer array, which stores beginning of each block row.

We have developed a code generator which, for a given number of vectors m , produces a fully-unrolled SIMD kernel, which we call the *basic kernel*. This kernel multiplies a small 3×3 block by a $3 \times m$ block. Multiplication of each matrix element is unrolled by m . The nine elements of a 3×3 block are stored packed in SIMD registers and SIMD shuffle operations are used to extract and replicate required values. It is also possible to use vector blocking for multiple vectors, as this was shown to result in improved register allocation and cache performance [25], [18]. However, for our datasets, increasing m resulted in *at most* a commensurate run-time increase. As a result, vector blocking would not be effective for realistic values of m .

2) *Multi-Node Optimizations*: Similar to single-node SPMV, multi-node SPMV has been well-studied in the past, for example, see [32], [6]. Our multi-node GSPMV implementation is similar to multi-node implementations of SPMV, except that it operates on a block of vectors. For a given matrix partitioning, communication volume scales proportionately with the number of vectors, m .

Strong scaling performance of GSPMV is generally limited by two factors: load imbalance and communication overhead. To address load imbalance, we used a simple, coordinate-based row-partitioning scheme. This partitioning bins each particle using a 3D grid and attempts to balance the number of non-zeros in each partition. The entire operation is inexpensive, and can be done during neighbor list construction to further amortize its overhead over several time steps. Coordinate-based partitioning resulted in communication volume and load balance comparable to that of a METIS [22] partitioning.

To reduce communication overhead, we overlap computation with communication, using nonblocking communication MPI calls. We also overlap the gather of the elements to be communicated with the multiply by the local part of the matrix. We use a small subset of threads to perform the communication and gather operations, while the remaining threads perform the compute.

B. Performance Model

1) *Single-Node Bound*: Gropp et al. [16] analyzed the benefits of multiplying a sparse matrix by multiple vectors. However, only the bandwidth-bound case was analyzed. We also analyze the compute-bound case, which can arise for large-enough m . We also slightly extend the performance model to block-structured matrices which arise in many applications including SD simulation.

We now define some quantities used frequently in this paper. For the GSPMV operation $Y = RX$, let n denote the number of rows and let n_b denote the number of block rows in the matrix R (for 3×3 blocks, $n_b = n/3$). Further, let n_{nz} denote the number of stored scalar non-zeros in the matrix and let n_{nzb} denote the number of block non-zeros in the matrix. Let s_a be the size, in bytes, of a matrix block (for 3×3 blocks, $s_a = 72$ in double precision). Let s_x be the size, in bytes, of a scalar entry of the (dense) vectors to be multiplied.

The total amount of memory traffic in bytes incurred by a GSPMV operation is

$$M_{tr}(m) = mn_b(3 + k(m))s_x + 4n_b + n_{nzb}(4 + s_a)$$

Memory traffic due to non-zeros as well BCRS indexing structures, is represented by the second and third terms in the expression. The first term represents memory traffic due to accessing X and Y : 1 read of X , 1 read of Y , one write to Y , plus $k(m)$ additional memory accesses to each element of X . The function $k(m)$ depends on matrix structure as well as machine characteristics, such as cache size. The function $k(m)$ also depends on m : as the number of vectors increases, the working set also increases and will put additional pressure on the last level cache. In this case, $k(m)$ will also increase. Cache blocking and matrix reordering techniques will reduce the value of $k(m)$. Note $k(m)$ can also be negative, which can happen for example, when both X and Y fit into last level cache and are retained there between multiple calls to GSPMV. For the matrices that are typical in our SD simulation, $k(m)$ is only a weak function of m . For example, for a typical SD matrix with 25 non-zero blocks per block row, $k(m)$ is ~ 3 for m between 1 and 42.

The time for performing GSPMV with m vectors, if the operation is bandwidth-bound, is $T_{bw}(m) = M_{tr}(m)/B$, where B is the achievable machine bandwidth. The time in the compute-bound case is $T_{comp}(m) = f_a mn_{nzb}/F$, where f_a is number of flops required to multiply a block element of R by a block element of X . For example, f_a is 18 for the case of a 3×3 block. The quantity F is the achievable compute-bound performance of the basic kernel.

We approximate the performance of GSPMV by the maximum determined by the compute and bandwidth bounds, $T(m) = \max(T_{bw}(m), T_{comp}(m))$. The relative time, $r(m)$, is defined as the ratio of the time it takes to multiply by m vectors to the time it takes to multiply by one vector. Hence $r(m) = T(m)/T(1)$. Since we assume that $T(1)$ is bandwidth-bound, $r(m) = T(m)/T_{bw}(1)$. We can divide both numerator and denominator by n_b and note that n_{nzb}/n_b is an average number of non-zero blocks per block row of the matrix. The relative time becomes

$$r(m) = \frac{\max[m(3+k(m))s_x + 4 + (n_{nzb}/n_b)(4+s_a), m f_a(n_{nzb}/n_b)(B/F)]}{(3+k(1))s_x + 4 + (n_{nzb}/n_b)(4+s_a)} \quad (8)$$

For small values of m , the relative time is generally determined by the bandwidth bound. For larger m , it is possible that the compute-bound may start dominating the performance. This would be the case for large enough values of byte to flop ratio, B/F , and a large number of blocks per block row, n_{nzb}/n_b , in the matrix. It is also possible that if n_{nzb}/n_b is too low or $k(m)$ is too high, the bandwidth-bound will continue dominating the performance for all values of m . As an example of this, consider a very large diagonal matrix which does not fit into the last level of cache. Clearly, GSPMV is bandwidth-bound in this case for any value of m , since there is no reuse of any vector elements.

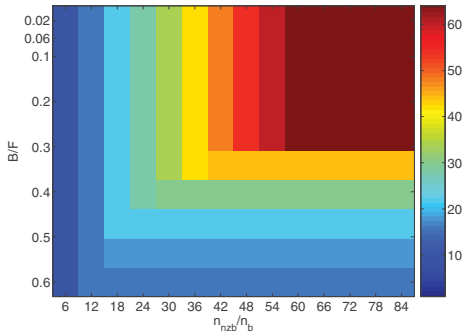


Fig. 1: Number of vectors that can be multiplied in 2 times the time needed to multiply by a single vector as a function of n_{nzb}/n_b (x-axis) and B/F (y-axis).

Using the above model, Figure 1 shows a profile of the number of vectors which can be computed in 2 times the time needed to multiply by one vector as n_{nzb}/n_b varies between 6 and 84 and as B/F varies between 0.02 and 0.6. For simplicity, $k(m)$ is optimistically assumed to be 0. The figure shows the trends, but in reality, $k(m)$ is greater than 0 and this will restrict the growth of the number of vectors that can be multiplied in a fixed amount of time. For example, as shown later in Section IV-D, for the same values of n_{nzb}/n_b and B/F , the experimentally obtained values of the number of vectors are somewhat smaller than those shown in this profile.

2) *Multi-Node Bound*: We now extend the definition of relative time to the multi-node case. For p nodes, the relative time $r(m, p)$ is the ratio of time to compute with m vectors on p nodes to the time to compute with a single vector on the same number of nodes. On a single-node GSPMV performance may be bound by bandwidth or computation; on multiple nodes, GSPMV performance can be also bound by communication, which will increase with p .

C. Experimental Setup

In this section we briefly describe the experimental setup for evaluating our GSPMV implementations. We introduce relevant hardware characteristics of the evaluated systems and present an overview of the test matrices.

1) *Single-Node Systems*: We performed single-node experiments on two modern multi-core processors: Intel® Xeon® Processor X5680, which is based on Intel® Core™ i7, and Intel® Xeon® Processor E5-2670, which is based on Sandy Bridge. In the rest of the paper, we abbreviate the first architecture as WSM (for Westmere) and the second as SNB (for Sandy Bridge).

WSM is a x86-based multi-core architecture which provides six cores on the same die running at 3.3 GHz. It features a super-scalar out-of-order micro-architecture supporting 2-way hyper-threading. In addition to scalar units, this architecture has 4-wide SIMD units that support a wide range of SIMD instructions called SSE4 [20]. Together, the six cores can deliver a peak performance of 79 Gflop/s of double-precision arithmetic. All cores share a large 12 MiB last level L3 cache. The system has three channels of DDR3 memory running at 1333 MHz, which can deliver 32 GB/s of peak bandwidth.

SNB is the latest x86-based architecture. It provides 8 cores on the same die running at 2.6 GHz. It has a 8-wide SIMD instruction set based on AVX [19]. Together, the 8 cores deliver 166 Gflops of double-precision arithmetic. All cores share a large 20 MiB last level L3 cache. The system has four channels of DDR3 which can deliver 43 GB/s of peak bandwidth.

We see that compared to WSM, SNB has 2.1 times higher compute throughput but only 1.3 times higher memory bandwidth. Effectively, compared to WSM, SNB can perform 1.6 times more operations per byte of data transferred from memory.

2) *Multi-Node Systems*: We performed multi-node experiments on a 64-node cluster. Each node consists of a dual-socket CPU with the same configuration as WSM, described in the previous section, except it runs at the lower frequency of 2.9 GHz. The nodes are connected via an InfiniBand interconnect that supports a one-way latency of 1.5 *usecs* for 4 bytes, a uni-directional bandwidth of up to 3380 MiB/s and bi-directional bandwidth of up to 6474 MiB/s. Note that in our experiments we have only used a single socket on each node.

TABLE I: Three matrices from SD.

Matrix	n	n_b	n_{nz}	n_{nzb}	n_{nzb}/n_b
mat1	0.9M	300K	15.3M	1.7M	5.6
mat2	1.2M	395K	81M	9M	24.9
mat3	1.2M	395K	162M	18M	45.3

3) *Matrix Datasets*: To study GSPMV, we used three matrices generated by our SD simulator, **mat1**, **mat2** and **mat3**. Table I summarizes their main characteristics. We changed the cutoff radius in the SD simulator to construct matrices with different values n_{nzb}/n_b .

D. Experimental Results

1) *Compute and Bandwidth Bounds*: The performance model described in Section IV-B requires B/F , which is the ratio of STREAM bandwidth to the achievable floating-

point performance of the basic kernel. Running STREAM¹ to obtain B on both architectures shows that WSM achieves 23 GB/s, while SNB achieves 33 GB/s, which is a factor of 1.5 improvement over WSM, due to the additional memory channel. To obtain F , we constructed a simple benchmark that repeatedly computed with the same block of memory. We ran this benchmark for various values of m between 1 and 64. If we exclude $m = 1$, which achieves low performance on both architectures due to low SIMD parallelism, on average this benchmark achieved 45 Gflops on WSM and 90 Gflops on SNB. The standard deviation from this average is $\sim 11\%$ for both architectures, the maximum deviation is 13% for WSM and 17% on SNB. The factor of 2 speedup of SNB over WSM is commensurate with their peak floating-point performance ratios. Note also our kernel achieved close to 70% floating-point efficiency on both architectures. The corresponding values of B/F are 0.55 and 0.37 for WSM and SNB, respectively.

TABLE II: Performance and bandwidth usage of SPMV ($m = 1$).

	mat1, WSM	mat2, WSM	mat3, SNB
GB/s	17.8	18.3	32.0
Gflops	3.6	4.2	7.4

2) *Single-Node Results*: Table II shows performance and bandwidth utilization of single-vector SPMV on both architectures and three matrices. It serves as our baseline. We can see our single vector performance is within 20% of achievable bandwidth on WSM and within 3% on SNB. The reason for such high bandwidth efficiency on SNB is its large 20 MiB last level cache which retains a large part of the X and Y vectors (example of negative $k(m)$ discussed in Section IV-B1). Note that we ran **mat1** and **mat2** matrices on WSM, while to capture the cumulative effects of increased n_{nz}/n_b and B/F on GSPMV performance, we ran **mat3** on SNB.

Figure 2(a) shows the achieved (red solid curve) versus predicted (green solid curve) relative time, r , for **mat2** on WSM, as m varies from 1 to 42. As described in Section IV-A achieved performance is the maximum of compute and bandwidth bounds. These two bounds are represented by dotted and dashed curves in the figure. The results show that our predicted relative time closely matches the trend in achieved relative time. A similar match between predicted and achieved relative times was observed for the other two matrices (not shown here for brevity).

Figure 2(b) shows the relative time as a function of m for all three test matrices. The red curve at the top represents the relative time for **mat1** on WSM. We see that for this matrix, we can compute 8 vectors in 2 times the time of a single vector. This is the smallest number of vectors, compared to the other two matrices, when run on the same hardware. This is not surprising because, as Table I shows, **mat1** has very small n_{nz}/n_b . As a result, it is bandwidth-bound for any number of vectors. The blue curve in the middle shows the relative time for matrix **mat2** on WSM. We see that for this

¹Nontemporal stores have been suppressed in the STREAM measurements and the bandwidth numbers reported have been scaled appropriately by 4/3 to account for the write-allocate transfer.

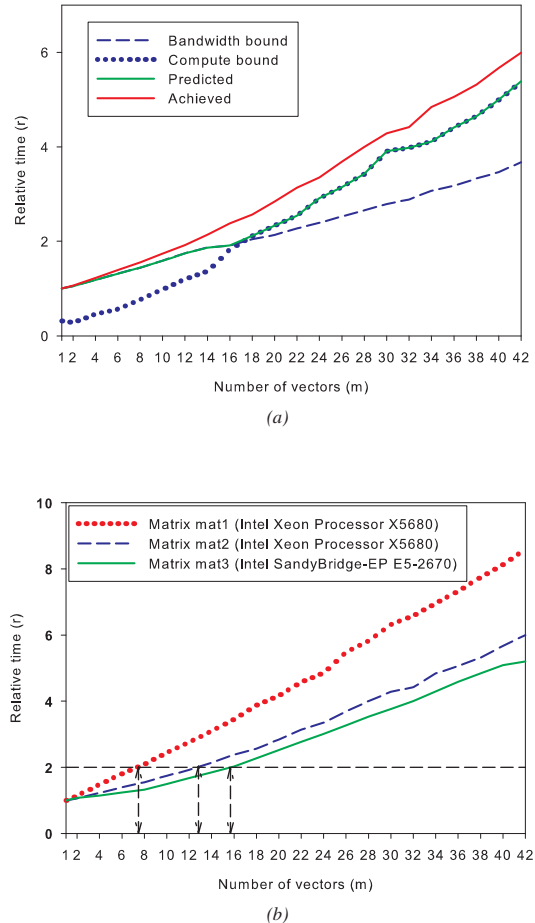


Fig. 2: Relative time, r , as a function of m . (a) correlation between performance model and achieved performance for **mat2** on WSM, (b) $r(m)$ for three matrices.

matrix, we can multiply as many as 12 vectors in 2 times the time needed to multiply a single vector: 4 more vectors compared to **mat1**. This is due to the fact that **mat2** has larger n_{nz}/n_b , compared to **mat1**. Finally, the bottom green curve shows relative time for matrix **mat3** on SNB. Note this matrix has the highest n_{nz}/n_b , compared to the other two matrices, **mat1** and **mat2**. Moreover, SNB has higher B/F , compared to WSM. As a result, we see that in this configuration we can multiply as many as 16 vectors in 2 times the time needed to multiply one vector.

3) *Multi-Node Results*: We describe the performance of GSPMV on multiple nodes using two matrices **mat1** and **mat2**. Figure 3 shows the relative time as m varies from 1 to 32 and number of the nodes is increased from 1 to 64. As defined earlier, for a given number of nodes, the *relative time* is the ratio of time required to multiply by m vectors to the time required to multiply by a single vector on the same number of nodes.

For small numbers of nodes, e.g., 4 and 16, the relative time curves are *somewhat higher* but similar to the case for a single node. The slight increase may be attributed to the

TABLE III: GSPMV communication time fractions for **mat1** matrix. The communication time is significantly higher than the computation time for 32 and 64 nodes. This is not surprising given **mat1**'s low n_{nz}/n_b of only 5.6.

	Number of vectors, m		
	1	8	32
32 nodes	88%	76%	52%
64 nodes	97%	90%	67%

cost of gathering remote vector values. For large numbers of nodes, e.g., 64, the relative time curves are *lower* than for the single node case. This is because communication costs dominate for the case of large numbers of nodes (as shown in Table III). Therefore, the additional compute required as the number of vectors increases does not significantly affect the overall time of GSPMV. In addition, the communication time of GSPMV on large numbers of nodes is mainly consumed by message-passing latency. For a given number of nodes, the time increases very slowly with increasing numbers of vectors. This leads to lower values of relative time for large numbers of nodes.

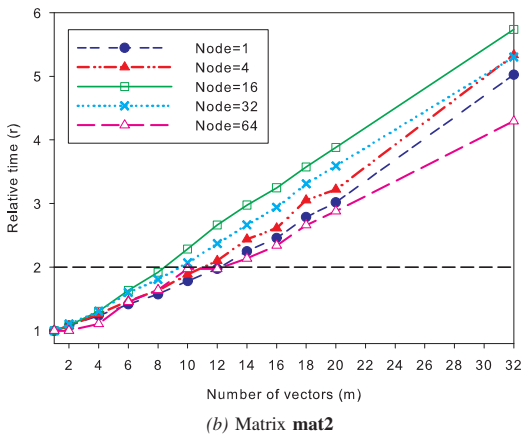
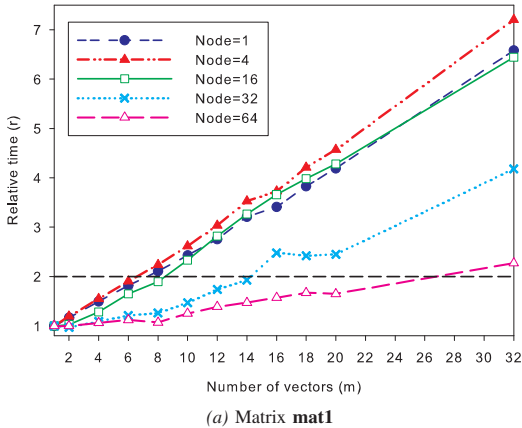


Fig. 3: Relative time for GSPMV using matrix (a) **mat1** and (b) **mat2** as a function of m for various number of nodes up to 64.

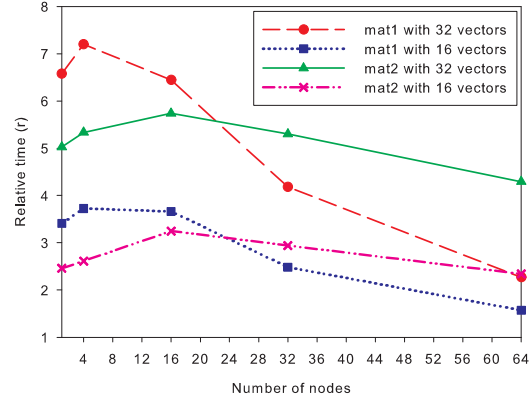


Fig. 4: Relative time for GSPMV as a function of number of nodes.

In summary, Figure 4 shows the trend in relative time as a function of the number of nodes. As explained above, the relative time increases slightly and then decreases. These results show preliminarily that the use of GSPMV is particularly effective when using large numbers of nodes. Further experiments, however, are needed to test other types of matrices and other partitioning schemes, as well as potentially other implementations.

V. STOKESIAN DYNAMICS RESULTS

In this section, we test the performance of the multiple right-hand side algorithm (Algorithm 2) in a SD application. Indeed, our motivation to improve the performance of SD led to the approach proposed in this paper. Demonstrating the algorithm in the context of an actual application is important because we are then using a sequence of matrices with an application-determined variation, rather than an artificial sequence of matrices which may be parameterized to vary faster or slower.

A. Simulation Setup

Our test system is a collection of 300,000 spheres of various radii in a simulation box with periodic boundary conditions. The spheres represent proteins in a distribution of sizes that matches the distribution of sizes of proteins in the cytoplasm of *E. coli* [1] (see Table IV). The volume occupancy of molecules in the *E. coli* cytoplasm may be as high as 40 percent. Volume occupancy significantly affects the convergence behavior of the iterative algorithms used in SD. Systems with high volume occupancies tend to have pairs of particles which are extremely close to each other, resulting in ill-conditioning of the resistance matrix. Since convergence behavior is a critical factor in the performance of the MRHS algorithm, we test a range of volume occupancies: 10%, 30% and 50%.

The time step length for the simulations is 2 ps. This is the maximum time step size that can be used while avoiding particle overlaps in the simulation. Use of a smaller time step decreases the overall simulation rate. For computing the Brownian forces to a given accuracy, we have set the maximum order of the Chebyshev polynomial to 30 (i.e., 30

TABLE IV: Distribution of particle radii.

Particle radius (Å)	Distribution (%)
115.24	2.43
85.23	3.16
66.49	6.55
49.16	0.97
45.43	0.49
43.06	3.64
42.48	2.91
39.16	2.67
36.76	8.01
35.94	8.01
31.71	10.92
27.77	25.97
25.75	8.25
24.01	9.95
21.42	6.07

sparse matrix-vector multiplies to compute the Chebyshev polynomial of a matrix times a vector).

Our SD code was written in standard C99, and was compiled with Intel ICC 11.0 using `-O3` optimization. All the experiments were carried out on a dual-socket quad-core (Intel Xeon E5530) server with 12 GB RAM using OpenMP or multicore BLAS parallelization. We do not currently have a distributed memory SD simulation code. Such a code would be very complex, needing new algorithms for parallelization and load balancing which we are also developing. In any case, the performance results for GSPMV on shared memory and distributed systems, as was shown in Section IV, are qualitatively similar, and thus we expect similar conclusions for distributed memory machines.

B. Experimental Results

1) *Accuracy of the Initial Guesses:* As described in Section III, the MRHS algorithm processes chunks of m time steps together. At the beginning of every m time steps, one augmented system with m right-hand sides is solved to provide the solution for the first time step and initial guesses for the following $m - 1$ time steps. The effectiveness of these initial guesses depend critically on how quickly the resistance matrix R changes as the time steps progress. To obtain some quantitative insight, Figure 5 shows the norm of the difference between the solution and the initial guess for several time steps. An important observation is that the discrepancy between the initial guesses and the solutions appear to increase as the square root of time. This result is consistent with the fact that the particle configurations due to Brownian motion also diverge as the square root of time. This is a very positive result because it implies that changes in the matrix R with respect to an instance at a given point in time actually *slow down* over time. This suggests the possibility that using a large number of right-hand sides may be profitable in the MRHS algorithm.

It is, of course, more relevant to measure the actual number of iterations required for convergence as the number of time steps increases, while using initial guesses constructed using the system at the first time step. The results are shown in Figure 6, where indeed, the number of iterations appear to grow slowly over time. In these tests, the conjugate gradient (CG) method was used and the iterations were stopped when

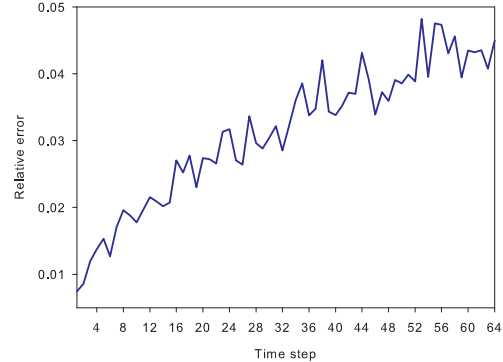


Fig. 5: The relative error $\|(u_k - u'_k)\|_2 / \|u_k\|_2$, where u_k and u'_k are the solution and initial guess at time step k , respectively. The system at the first time step is used for generating the initial guesses. The plot shows a square-root-like behavior which mimics the displacement of a Brownian system over time (the constant of proportionality of relative time divided by the square root of the time step number is approximately 0.006). This result is for a system with 3,000 particles and 50% volume occupancy.

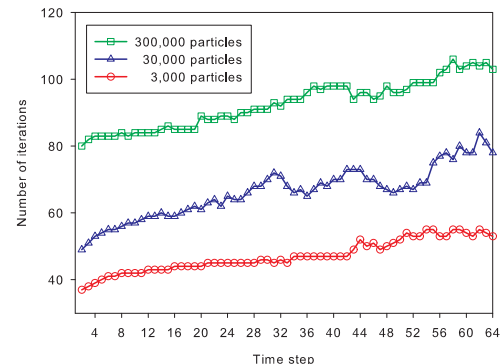


Fig. 6: Number of iterations for convergence vs. time step, with initial guesses. The volume occupancy is 50% for the 3 simulation systems.

the residual norm became less than 10^{-6} times the norm of the right-hand side.

Table V shows the number of iterations required for convergence for particle systems with different volume occupancies. For higher volume fractions, the degradation in performance is faster than for lower volume fractions. The table also shows that the number of iterations is reduced by 30% to 40% when initial guesses are used.

2) *Simulation Timings:* We now turn to timings of the SD simulation itself. Tables VII and VI show average timings for one time step for SD using the MRHS algorithm and for SD using the original algorithm without initial guesses. The MRHS algorithm used 16 right-hand sides. The tables show the compute time for major components of the simulations. These are: computing Brownian forces with using Chebyshev polynomial approximations using multiple vectors (*Cheb vectors*, step 2 in Algorithm 2); solving the auxiliary system for the initial guesses (*Calc guesses*, step 3 in Algorithm 2),

TABLE V: Number of iterations with and without initial guesses. The table shows the results for 300,000 particle systems with 10%, 30% and 50% volume occupancy.

Step	with guesses			without guesses		
	0.1	0.3	0.5	0.1	0.3	0.5
2	8	12	80	16	30	162
4	8	13	83	16	30	161
6	8	13	83	16	30	162
8	9	14	84	16	30	163
10	9	14	84	16	30	162
12	9	14	84	16	30	162
14	9	14	85	16	30	163
16	9	14	85	16	30	163
18	9	14	85	16	30	162
20	9	14	89	16	30	163
22	9	14	88	16	30	163
24	9	15	89	16	30	163

TABLE VI: Breakdown of timings (in seconds) for one time step for simulations with varying problem sizes. The volume occupancy of systems is 50%. Note that Chebyshev with multiple vectors and solves with multiple right-hand sides are amortized over many time steps and are not required in the original algorithm (marked by -).

	MRHS algorithm			Original algorithm		
	3000	30000	300000	3000	30000	300000
Cheb vectors	0.025	0.20	1.75	-	-	-
Calc guesses	0.076	0.71	9.66	-	-	-
Cheb single	0.005	0.08	0.84	0.005	0.08	0.84
1st solve	0.007	0.15	2.34	0.014	0.30	4.62
2nd solve	0.003	0.08	1.80	0.004	0.11	2.24
Average	0.021	0.36	5.46	0.023	0.49	7.70

which is only required in the MRHS algorithm; and the two solves with single right-hand sides (*1st solve* and *2nd solve*, steps 10 and 12 in Algorithm 2); as well as Chebyshev with single vector (*Cheb single*, step 9 in Algorithm 2), which are used in both the MRHS algorithm and the original algorithm. Note that in *both* algorithms, in each timestep, the solution of the first solve is used as the initial guess for the second solve.

The results show that, for most cases, the operations for Chebyshev with multiple vectors and the solves with multiple right-hand sides are very efficient. The operations with a block of 16 vectors, for example, are efficient because they are implemented with GSPMV. On the other hand, very large m can be used due to the slow degradation of convergence using these initial guesses. The average simulation time per time step is presented in the last row of Table VI and Table VII, which show that the simulations with the MRHS algorithm are 10% to 30% faster than those with the original algorithm.

3) *Analytic Model and Discussions*: An important question for the MRHS algorithm is how many right-hand sides should be used to minimize the average time for one simulation step. It can be shown that the best performance is achieved roughly when GSPMV switches from being bandwidth-bound to being compute-bound. The details are as follows.

As seen in Figure 6 and Table V, the number of iterations increases slowly over time. We assume it is constant in the following analysis. Let N denote the number of iterations for the 1st solve without an initial guess. Let N_1 and N_2 denote that number for the 1st solve and the 2nd solve respectively, both with an initial guess. Typically, $N > N_1 > N_2$.

TABLE VII: Breakdown of timings (in seconds) for one time step for simulations with varying volume occupancy. The results are for systems with 300,000 particles. Note that Chebyshev with multiple vectors and solves with multiple right-hand sides are amortized over many time steps and are not required in the original algorithm (marked by -).

	MRHS algorithm			Original algorithm		
	0.1	0.3	0.5	0.1	0.3	0.5
Cheb vectors	1.09	1.34	1.75	-	-	-
Calc guesses	0.58	1.47	9.66	-	-	-
Cheb single	0.40	0.56	0.84	0.40	0.56	0.84
1st solve	0.12	0.25	2.34	0.22	0.61	4.62
2nd solve	0.08	0.15	1.80	0.08	0.15	2.24
Average	0.66	1.07	5.46	0.70	1.32	7.70

Supposing m right-hand sides are used, the average time for one simulation step with the MRHS algorithm can be expressed as

$$\begin{aligned}
 T_{mrhs}(m) = & \frac{1}{m} \left[\underbrace{N T(m)}_{\text{Calc guesses}} + \underbrace{C_{max} T(m)}_{\text{Cheb vectors}} \right. \\
 & + \underbrace{(m-1) N_1 T(1)}_{\text{1st solve with an initial guess}} \\
 & \left. + \underbrace{m N_2 T(1)}_{\text{2nd solve}} + \underbrace{(m-1) C_{max} T(1)}_{\text{Cheb single}} \right] \quad (9)
 \end{aligned}$$

where $T(m)$ is the time for GSPMV with m vectors, $T(1)$ is the time for SPMV (with a single vector), and C_{max} is the maximum order of the Chebyshev polynomial. The purpose of our analysis is to find the value of m which minimizes T_{mrhs} . We denote this value by $m_{optimal}$.

Recall the analysis in Section IV, where the performance of GSPMV is modeled as $T(m) = \max(T_{bw}(m), T_{comp}(m))$. For small values of m , GSPMV is bandwidth-bound, where $T(m)$ is equal to $T_{bw}(m)$. As m increases, there are two cases: if $k(m)$ is very large or (n_{nzb}/n_b) is small, the bandwidth bound will continue to dominate, and $T(m)$ is still determined by $T_{bw}(m)$; otherwise, the compute bound starts to dominate, and at some value of m (denoted by m_s), GSPMV switches from being bandwidth-bound to being compute-bound. In this case, $T(m)$ is equal to $T_{comp}(m)$.

In our SD simulations, most systems are in the second case, thus $T(1)$ and $T(m)$ can be expressed as

$$\begin{aligned}
 T(1) &= (n_b (3 + k(1)) s_x + 4n_b + n_{nzb} (4 + s_a)) / B \\
 T(m) &= \begin{cases} (m n_b (3 + k(m)) s_x + 4n_b \\ \quad + n_{nzb} (4 + s_a)) / B & \text{if } m < m_s \\ f_a m n_{nzb} / F & \text{if } m \geq m_s \end{cases} \quad (10)
 \end{aligned}$$

Expanding $T(1)$ and $T(m)$ in equation (9), when $m < m_s$, T_{mrhs} can be expressed as a function of $k(m)$ and m

$$T_{mrhs}(m < m_s) = (3 + k(m))P + \frac{1}{m} Q + R \quad (11)$$

TABLE VIII: m_s and $m_{optimal}$ for different systems.

Problem size	Volume occupancy	m_s	$m_{optimal}$
3,000	50%	5	4
30,000	50%	12	10
300,000	10%	15	12
300,000	30%	13	10
300,000	50%	12	10

where P , Q and R are all constants,

$$P = \frac{(N + N_2 + C_{max}) s_x n_b}{B}$$

$$Q = \frac{N - N_1}{B} [(4 n_b + n_{nzb} (4 + s_a)) - (N_1 + N_2 + C_{max}) (3 + k(1)) s_x n_b]$$

$$R = \frac{N_1 + N_2 + C_{max}}{B} [n_b (3 + k(1)) s_x + 4 n_b + n_{nzb} (4 + s_a)]$$

Typically in SD, n_{nzb} is large, and hence $Q > 0$. When $k(m)$ is small and changes very slowly with m , which is our case as mentioned earlier, the expression (11) is a decreasing function of m .

On the other hand, when $m \geq m_s$,

$$T_{mrhs}(m \geq m_s) = S + W - \frac{S}{m} \quad (12)$$

where

$$S = (N_1 + N_2 + C_{max}) [n_b (3 + k(1)) s_x + 4 n_b + n_{nzb} (4 + s_a)]$$

$$W = \frac{f_a n_{nzb} (N + N_2 + C_{max})}{F}$$

which is an increasing function of m (F is almost constant when GSPMV is compute-bound).

Putting these expressions together, we conclude that the best simulation performance is achieved when m is near m_s , i.e., when GSPMV switches from being bandwidth-bound to being compute-bound.

We evaluate our analysis by running simulations on various test problems. For each simulation, experiments were performed with different numbers of right-hand sides to determine the values of $m_{optimal}$. GSPMV was also run on these test problems to determine m_s .

Figure 7 displays the achieved (red solid curve) versus predicted (green solid curve) average simulation time per time step (T_{mrhs}) for a system with 300,000 particles and 50% volume occupancy. The predicted simulation time is the maximum of the bandwidth-bound and compute-bound estimates of T_{mrhs} . As seen in the figure, the achieved T_{mrhs} first decreases as m increases and starts to increase when m is equal to $m_{optimal}$, which matches the trend of the predicted simulation time. Table VIII compares $m_{optimal}$ and m_s for 5 different simulations, showing that they are indeed very close. The slight differences can be explained by the fact that N_1 is actually increasing in our simulations, although very slowly. These results corroborate our analysis.

Finally, we show some results that investigate the speedup of the MRHS algorithm as we increase the number of threads in a shared-memory computation. Figure 8(a) shows the computation time of GSPMV for different numbers of threads. For 8 threads, the ratio B/F is smaller than for 2 or 4 threads. As a result, the speedup with 8 threads shown in

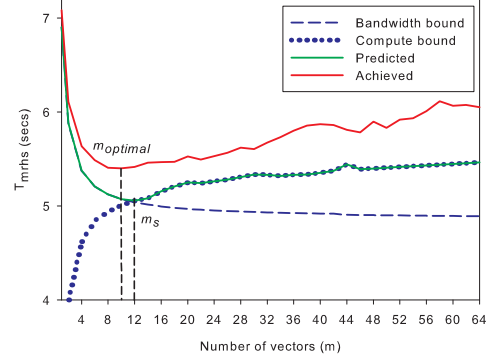


Fig. 7: Predicted and achieved average simulation time per time step vs. m . The result is for a system with 300,000 particles and 50% volume occupancy. Equations (11) and (12) were used to calculate the compute-bound and bandwidth-bound estimates with the following parameters: $N = 162$, $N_1 = 80$, $N_2 = 63$, $C_{max} = 30$, $B = 19.4GB/s$ (STREAM bandwidth). F and $k(m)$ are measured values.

Figure 8(b) is larger than that with fewer threads. This result demonstrates the potential of using the MRHS algorithm with large manycore nodes.

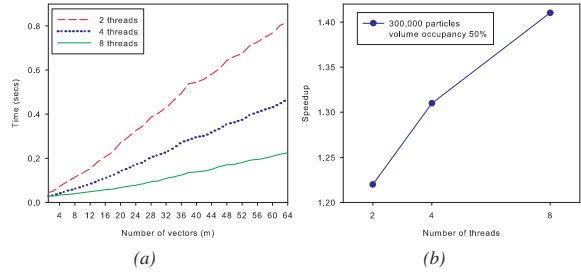


Fig. 8: (a) Performance of GSPMV vs. number of threads. (b) Speedup over the original algorithm vs. number of threads. The results are for a system with 300,000 particles and 50% volume occupancy.

VI. CONCLUSION

In this paper, we presented an algorithm for improving the performance of Stokesian dynamics simulations. We redesigned the existing algorithm which used SPMV with single vectors to instead use the more efficient GSPMV. The main idea of the new MRHS algorithm is to solve an auxiliary system with multiple right-hand sides; the solution to this auxiliary system helps solve the original systems by providing good initial guesses. The approach of the algorithm can be extended to other types of dynamical simulations.

We presented a performance model of GSPMV and used it to explain GSPMV performance. We observe that for matrices with very small numbers of non-zeros per row, GSPMV performance is always bandwidth-bound, while for matrices with larger numbers of non-zeros per row, typical for SD and many other applications, GSPMV switches from bandwidth-bound to compute-bound behavior with increasing

numbers of vectors. In either case, it is typical to be able to multiply a sparse matrix by 8 to 16 vectors simultaneously in only twice the time required to multiply by a single vector. Similar results hold for distributed memory computations. We thus “update” the earlier result reported in [16].

We demonstrated how to exploit multiple right-hand sides in SD simulations. By using the MRHS algorithm, we measured a 30 percent speedup in performance. In addition, we used a simple model to show that the best simulation performance is achieved near the point where GSPMV switches from being bandwidth-bound to being compute-bound.

The efficiency of the MRHS algorithm depends on properties of the system being simulated and also characteristics of the hardware. With the ever-increasing gap between DRAM and processor performance, we expect that the effort of exploiting multiple right-hand sides will become even more profitable in the future.

ACKNOWLEDGMENTS

The authors would like to thank Richard Vuduc and Tadashi Ando for helpful discussions. This work was partially supported by a grant from Intel Corporation.

REFERENCES

- [1] T. Ando and J. Skolnick, “Crowding and hydrodynamic interactions likely dominate in vivo macromolecular motion,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 43, pp. 18457–18462, 2010.
- [2] A. J. Banchio and J. F. Brady, “Accelerated Stokesian dynamics: Brownian motion,” *The Journal of Chemical Physics*, vol. 118, no. 22, pp. 10323–10332, 2003.
- [3] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09, 2009, pp. 18:1–18:11.
- [4] G. Bossis and J. F. Brady, “Dynamic simulation of sheared suspensions. I. General method,” *The Journal of Chemical Physics*, vol. 80, no. 10, pp. 5141–5154, 1984.
- [5] J. F. Brady and G. Bossis, “Stokesian Dynamics,” *Annual Review of Fluid Mechanics*, vol. 20, no. 1, pp. 111–157, 1988.
- [6] A. Buluc and J. Gilbert, “Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication,” in *Proc. ICPP*, 2008.
- [7] A. Buluc, S. W. Williams, L. Oliker, and J. Demmel, “Reduced-bandwidth multithreaded algorithms for sparse-matrix vector multiplication,” in *Proc. IPDPS 2011*, 2011.
- [8] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA ’09. New York, NY, USA: ACM, 2009, pp. 233–244.
- [9] B. Cichocki, M. L. E. Jezewska, and E. Wajnryb, “Lubrication corrections for three-particle contribution to short-time self-diffusion coefficients in colloidal dispersions,” *The Journal of Chemical Physics*, vol. 111, no. 7, pp. 3265–3273, 1999.
- [10] L. Durlofsky, J. F. Brady, and G. Bossis, “Dynamic simulation of hydrodynamically interacting particles,” *Journal of Fluid Mechanics*, vol. 180, pp. 21–49, 1987.
- [11] D. L. Ermak and J. A. McCammon, “Brownian dynamics with hydrodynamic interactions,” *The Journal of Chemical Physics*, vol. 69, no. 4, pp. 1352–1360, 1978.
- [12] M. Fixman, “Simulation of polymer dynamics. I. General theory,” *The Journal of Chemical Physics*, vol. 69, no. 4, pp. 1527–1537, 1978.
- [13] —, “Construction of Langevin forces in the simulation of hydrodynamic interaction,” *Macromolecules*, vol. 19, no. 4, pp. 1204–1207, 1986.
- [14] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, “Performance evaluation of the sparse matrix-vector multiplication on modern architectures,” *J. Supercomput.*, vol. 50, pp. 36–77, October 2009.
- [15] P. S. Grassia, E. J. Hinch, and L. C. Nitsche, “Computer simulations of Brownian motion of complex systems,” *Journal of Fluid Mechanics*, vol. 282, pp. 373–403, 1995.
- [16] W. Gropp, D. Kaushik, D. Keyes, and B. Smith, “Toward realistic performance bounds for implicit CFD codes,” in *Proceedings of Parallel CFD’99*, A. Ecer, Ed. Elsevier, 1999.
- [17] E. K. Guckel, “Large scale simulation of particulate systems using the PME method,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1999.
- [18] E.-J. Im, “Optimizing the performance of sparse matrix-vector multiplication,” Ph.D. dissertation, University of California, Berkeley, Jun 2000.
- [19] “Intel Advanced Vector Extensions Programming Reference,” 2008, <http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf>.
- [20] “Intel SSE4 programming reference,” 2007, <http://www.intel.com/design/processor/manuals/253667.pdf>.
- [21] D. J. Jeffrey and Y. Onishi, “Calculation of the resistance and mobility functions for two unequal rigid spheres in low-Reynolds-number flow,” *Journal of Fluid Mechanics*, vol. 139, pp. 261–290, 1984.
- [22] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1999.
- [23] S. Kim and S. J. Karrila, *Microhydrodynamics: Principles and Selected Applications*. Boston: Butterworth-Heinemann, June 1991.
- [24] M. Krotkiewski and M. Dabrowski, “Parallel symmetric sparse matrix-vector product on scalar multi-core cpus,” *Parallel Comput.*, vol. 36, pp. 181–198, April 2010.
- [25] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, “Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply,” in *Proceedings of the 2004 International Conference on Parallel Processing*, ser. ICPP ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 169–176.
- [26] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, “When cache blocking of sparse matrix vector multiply works and why,” *Appl. Algebra Eng., Commun. Comput.*, vol. 18, pp. 297–311, May 2007.
- [27] D. P. O’Leary, “The block conjugate gradient algorithm and related methods,” *Linear Algebra and Its Applications*, vol. 29, pp. 293–322, 1980.
- [28] M. L. Parks, E. de Sturler, G. Mackey, D. D. Johnson, and S. Maiti, “Recycling Krylov Subspaces for Sequences of Linear Systems,” *SIAM J. Sci. Comput.*, vol. 28, pp. 1651–1674, Sept. 2006.
- [29] A. Pinar and M. T. Heath, “Improving performance of sparse matrix-vector multiplication,” in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing ’99. New York, NY, USA: ACM, 1999.
- [30] J. Rotne and S. Prager, “Variational Treatment of Hydrodynamic Interaction in Polymers,” *Journal of Chemical Physics*, vol. 50, no. 11, pp. 4831–4837, June 1969.
- [31] D. Saintillan, E. Darve, and E. S. G. Shaqfeh, “A smooth particle-mesh Ewald algorithm for Stokes suspension simulations: The sedimentation of fibers,” *Physics of Fluids*, vol. 17, no. 3, p. 033301, 2005.
- [32] G. Schubert, G. Hager, H. Fehske, and G. Wellein, “Parallel Sparse Matrix-Vector Multiplication as a Test Case for Hybrid MPI+OpenMP Programming,” in *IPDPS Workshops*, 2011.
- [33] A. Sierou and J. F. Brady, “Accelerated Stokesian Dynamics simulations,” *Journal of Fluid Mechanics*, vol. 448, pp. 115–146, 2001.
- [34] F. E. Torres and J. R. Gilbert, “Large-Scale Stokesian Dynamics Simulations of Non-Brownian Suspensions,” Xerox Research Centre of Canada, Tech. Rep. C9600004, 1996.
- [35] M. N. Viera, “Large scale simulation of Brownian suspensions,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2002.
- [36] R. W. Vuduc and H.-J. Moon, “Fast sparse matrix-vector multiplication by exploiting variable block structure,” in *Proc. High-Performance Computing and Communications Conf. (HPCC)*, vol. LNCS 3726. Sorrento, Italy: Springer, September 2005, pp. 807–816.
- [37] R. W. Vuduc, “Automatic performance tuning of sparse matrix kernels,” Ph.D. dissertation, University of California, Berkeley, 2003.
- [38] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC ’07. New York, NY, USA: ACM, 2007, pp. 38:1–38:12.
- [39] H. Yamakawa, “Transport Properties of Polymer Chains in Dilute Solution: Hydrodynamic Interaction,” *Journal of Chemical Physics*, vol. 53, no. 1, pp. 436–443, July 1970.