Large-Scale Hydrodynamic Brownian Simulations on Multicore and Manycore Architectures

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

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

***Abstract*—Conventional Brownian dynamics (BD) simulations with hydrodynamic interactions utilize $3n \times 3n$ dense mobility matrices, where n is the number of simulated particles. This limits the size of BD simulations, particularly on accelerators with low memory capacities. In this paper, we formulate a matrix-free algorithm for BD simulations, allowing us to scale to very large numbers of particles while also being efficient for small numbers of particles. We discuss the implementation of this method for multicore and manycore architectures, as well as a hybrid implementation that splits the workload between CPUs and Intel Xeon Phi coprocessors. For 10,000 particles, the limit of the conventional algorithm on a 32 GB system, the matrix-free algorithm is 35 times faster than the conventional matrix-based algorithm. We show numerical tests for the matrix-free algorithm up to 500,000 particles. For large systems, our hybrid implementation using two Intel Xeon Phi coprocessors achieves a speedup of over 3.5x compared to the CPU-only case. Our optimizations also make the matrix-free algorithm faster than the conventional dense matrix algorithm on as few as 1000 particles.**

***Index Terms*—Brownian dynamics; particle-mesh Ewald (PME); hybrid parallelization; Intel Xeon Phi**

I. Introduction

Brownian dynamics (BD) is a computational method for simulating the motion of particles, such as macromolecules and nanoparticles, in a fluid environment. It has a myriad of applications in multiple areas including biology, biochemistry, chemical engineering and materials science [1], [2]. In BD simulations, only solute molecules are treated explicitly; fluid molecules are modeled implicitly as random forces on the solute molecules.

Hydrodynamic interactions (HI) are long-range interactions between particles where the motion of a particle through the fluid induces a force, mediated by the fluid, on all other particles. The modeling of HI is essential for correctly capturing the dynamics of particles, particularly collective motions [3], [4]. Although the modeling of HI makes BD simulations more realistic and more comparable to experiments, these interactions are completely neglected in many simulations because they are costly to compute. Efficient ways of modeling HI are arguably one of the biggest hurdles facing computational biologists striving for higher fidelity macromolecular simulations [5].

The high computational cost of modeling HI is mainly due to the explicit construction of dense hydrodynamic mobility matrices and the calculation of Brownian displacement vectors. Explicit construction of the mobility matrix requires $O(n^2)$ operations with a large constant, as well as $O(n^2)$ space, where n is the number of simulated particles. The simplest and most common technique for calculating Brownian displacements is to compute a Cholesky factorization, which scales as $O(n^3)$. Due to the expensive cost of modeling HI, conventional BD algorithms with HI are only applied to small systems with a few thousand particles.

In order to address the problems of large-scale BD simulations with HI, we use a matrix-free approach. The main idea is to avoid constructing the hydrodynamic mobility matrix, and to utilize methods for computing Brownian displacements that do not require this matrix to be available explicitly. Specifically, we replace the mobility matrix by a particle-mesh Ewald (PME) summation. The PME algorithm is well-known for computing electrostatic interactions in molecular dynamics (MD) simulations [6], [7]. It scales as $O(n \log n)$ and only requires $O(n)$ storage. To compute Brownian displacements without a matrix in this context, we use a Krylov subspace method [8].

The matrix-free approach also allows large-scale BD simulations to be accelerated on hardware that have relatively low memory capacities, such as GPUs and Intel Xeon Phi. In this paper we describe an efficient implementation of the matrix-free BD algorithm on multicore CPUs as well as a hybrid implementation using the Intel Xeon Phi coprocessor. The new implementations are capable of simulating systems with as many as 500,000 particles. The experimental results show that the matrix-free algorithm implemented on CPUs is more than 35x faster than the conventional BD algorithm in simulating large systems. The hybrid BD implementation using two Intel Xeon Phi coprocessors achieves additional speedups of over 3.5x for large simulated systems.

Related work. Existing BD codes that model HI, including BD_BOX [9] and Brownmove [10], use the conventional BD algorithm, in which the mobility matrix is explicitly constructed. There also exist BD codes optimized for GPUs [9], [11], but they are limited to approximately 3,000 particles, again due to the explicit construction of the dense mobility matrix. HI can also be modeled using a sparse matrix approximation [12], [13] when the interactions are primarily short-range; this is the approach used in LAMMPS [14].

Matrix-free approaches for particle-fluid simulations is not completely new. PME has been used to accelerate simulations in Stokesian conditions [15], [16], [17]. However, these codes use the PME summation of the Stokeslet or Oseen tensor,

rather than the Rotne-Prager-Yamakawa [18], [19] tensor widely used in BD. In our work, we use this latter tensor with PME and also specifically handle the computation of Brownian forces with PME.

PME algorithms are implemented widely in molecular dynamics codes for electrostatic interactions. Harvey and Fabritiis [20], describe an implementation of smooth PME on GPU hardware for MD. However, there are many differences between PME for MD and PME for BD that affect implementation and use.

In our previous work, we presented the use of Krylov subspace methods for computing Brownian displacements in the BD algorithm [8]. We did not use PME in that work, but the use of PME necessitates matrix-free approaches for computing Brownian displacements such as Krylov subspace methods.

II. Background: Conventional Ewald BD Algorithm

A. Brownian Dynamics with Hydrodynamic Interactions

In BD simulations, the solvent molecules are modeled as spherical particles of possibly varying radii. Like in other particle simulation methods, particle positions are propagated step by step. The BD propagation formula with HI can be expressed as [21]

$$\vec{r}(t+\Delta t) = \vec{r}(t) + M\vec{f}\Delta t + k_B T(\nabla \cdot M)\Delta t + \vec{g} \qquad (1)$$
$$\langle \vec{g} \rangle = 0, \quad \langle \vec{g}\,\vec{g}^{\,T} \rangle = 2k_B T M \Delta t.$$

Here, $\vec{r}$ is the position vector of the n particles, t is the time, Δt is the time step length, k_B is Boltzmann's constant, T is the temperature, $\vec{f}$ is the forces determined by the gradient of potential energy, and $\vec{g}$ is the Brownian displacement. The term involving $\vec{f}$ may represent van der Waals or bonded interactions, etc.

The matrix M is the mobility matrix. The (i,j)-th entry in M is a 3×3 tensor describing the interaction between particles i and j. Thus, the size of M is $3n\times 3n$. The Rotne-Prager-Yamakawa (RPY) tensor is widely used for modeling HI in BD simulations. With free boundary conditions, the entries of M using the RPY tensor are

$$M_{ij} = \frac{1}{6\pi\eta a}\left[\frac{3a}{4\|\vec{r}_{ij}\|}(I+\hat{r}_{ij}\hat{r}_{ij}^T) + \frac{a^3}{2\|\vec{r}_{ij}\|^3}(I-3\hat{r}_{ij}\hat{r}_{ij}^T)\right]$$

when $i \neq j$, and $M_{ii} = (6\pi\eta a)^{-1} I$ otherwise. In the above, $\vec{r}_{ij}$ is the vector between particles i and j, $\hat{r}_{ij}$ is the normalized vector, η is the viscosity, and a is the radii of the particles. The matrix M is symmetric positive definite for all particle configurations. As is common, we use conditions such that $\nabla \cdot M = 0$ so that the third term in Equation (1) is zero.

B. Ewald Summation of the RPY Tensor

Periodic boundary conditions are widely used in BD simulations. In this case, a particle i not only has long-range interactions with a particle j in its own simulation box, but also with all the images of j in the infinite number of replicas of the simulation box tiling all of space. In matrix terms, this means that the tensor M_{ij} describing the interaction between particles i and j is an infinite sum of terms.

Ewald summation is the standard procedure for computing infinite sums, by reformulating the sum into two rapidly converging sums, one in real space and one in reciprocal (Fourier) space. The Ewald sum has the form

$$M = M^{real} + M^{recip} + M^{self}$$

where the first term is the real-space sum, the second-term is the reciprocal-space sum, and the third term is a constant. For the RPY tensor, Beenakker [22] derived the formulas

$$\begin{aligned} M_{ij}^{real} &= \sum_{\vec{l}} M_\alpha^{(1)}(\vec{r}_{ij} + \vec{l}L) \\ M_{ij}^{recip} &= \frac{1}{L^3}\sum_{\vec{k}\neq 0} \exp(-i\vec{k}\cdot\vec{r}_{ij}) M_\alpha^{(2)}(\vec{k}) \qquad (2) \\ M_{ij}^{self} &= M_\alpha^{(0)}\delta_{ij} \end{aligned}$$

where L is the width of the simulation box and δ_{ij} is the Kronecker delta. The real-space sum is over all replicas $\vec{l}$ of the simulation box ($\vec{l}$ is a vector from the simulation box to one of its replicas). The reciprocal-space sum is over all Fourier lattice vectors $\vec{k} \neq 0$. The functions $M_\alpha^{(1)}(\vec{r})$ and $M_\alpha^{(2)}(\vec{k})$ derived by Beenakker [22] are designed to decay quickly with $\|\vec{r}\|$ and $\|\vec{k}\|$, respectively.

These functions are parameterized by α; if α is large, then the real-space sum converges faster than the reciprocal-space sum, and vice-versa. Thus α tunes the amount of work between the real-space and reciprocal-space sums and can be chosen to reduce computation time depending on the relative cost of performing the two summations.

C. Computing Brownian Displacements

From Equation (1), the Brownian displacement $\vec{g}$ is a random vector from a multivariate Gaussian distribution with mean zero and covariance $2k_BTM\Delta t$. This covariance is required by the fluctuation-dissipation theorem, relating stochastic forces with friction. The standard way to calculate $\vec{g}$ is by

$$\vec{g} = \sqrt{2k_BT\Delta t}\, S\, \vec{z}$$

where S is the lower-triangular Cholesky factor of M.

D. Ewald BD Algorithm

The BD algorithm with HI using Ewald summation and Cholesky factorization is presented in Algorithm 1. An observation used in BD simulations is that the mobility matrix changes slowly over time steps, meaning it is possible to use the same matrix for many time steps. Let λ_{RPY} be the update interval for the mobility matrix, whose value usually ranges from 10 to 100. In Algorithm 1, the mobility matrix and the Cholesky factorization are constructed only every λ_{RPY} time steps, and λ_{RPY} Brownian displacement vectors are generated together. Algorithm 1, called the *Ewald BD algorithm* is the baseline algorithm we use for comparisons.

III. Matrix-Free BD Algorithm

A. Particle-Mesh Ewald for the RPY Tensor

In this section, we derive the PME algorithm for the RPY tensor. To the best of our knowledge, the use of PME for the RPY tensor is new.

Algorithm 1: Ewald BD algorithm for m time steps. $\vec{r}_k$ denotes the position vector at time step k.

```
k ← 0
n ← m/λ_RPY
for i ← 1 to n do
    Construct dense matrix M_0 = M(r_k) using Ewald sums
    Compute Cholesky factorization M_0 = S S^T
    Generate λ_RPY random vectors Z = [z_1, ..., z_λRPY]
    Compute D = [d_1, ..., d_λRPY] = sqrt(2 k_B T Δt) S Z
    for j ← 1 to λ_PRY do
        Compute f(r_k)
        Update r_{k+1} = r_k + M_0 f(r_k) Δt + d_j
        k ← k + 1
    end
end
```

The PME method is a fast method for computing Ewald summations. The main idea is to use FFTs, rather than standard discrete transforms, to sum the reciprocal-space part of the Ewald summation. Since the particles are not regularly spaced, particle forces are first interpolated onto a regular mesh. The computed velocities on the regular mesh are then interpolated back onto the original particle locations.

The RPY operator acting on a vector of forces $\vec{f}$ can be written as

$$M\vec{f} = \underbrace{M^{real}\vec{f}}_{\vec{u}^{real}} + \underbrace{M^{recip}\vec{f}}_{\vec{u}^{recip}} + \underbrace{M^{self}\vec{f}}_{\vec{u}^{self}}$$

where the right-hand side consists of the real-space term, the reciprocal-space term, and a self term. The M^{self} operator is a constant tensor scaling, which is cheap and easy to implement in parallel; it will not be discussed further in this paper. In PME, the Ewald parameter α is chosen so that the real-space term can be computed using interactions between particles within a small cutoff distance r_{max}. The M^{real} operator can then be regarded as a sparse matrix with nonzeros M^{real}_{ij} when particles i and j are separated by not more than r_{max}. Since we will need to apply M^{real} multiple times to different vectors, it is advantageous to store it as a sparse matrix and perform the computation of the real-space term u^{real} as a sparse matrix-vector product (SpMV). We still call our approach "matrix-free" although we do construct this one sparse matrix.

Utilizing Equation (2), the reciprocal-space term $M^{recip}\vec{f}$ is

$$\vec{u}^{recip}_j = \frac{1}{L^3}\sum_{\vec{k}\neq 0}\sum_{i=1}^{n}\exp(-i\vec{k}\cdot\vec{r}_{ij})M^{(2)}_\alpha(\vec{k})\vec{f}_i$$

where $\vec{f}_i$ is the force on the i-th particle. By a simple manipulation,

$$\vec{u}^{recip}_j = \frac{1}{L^3}\sum_{\vec{k}\neq 0}\exp(i\vec{k}\cdot\vec{r}_j)M^{(2)}_\alpha(\vec{k})\sum_{i=1}^{n}\exp(-i\vec{k}\cdot\vec{r}_i)\vec{f}_i$$

where $\vec{r}_i$ and $\vec{r}_j$ are the positions of the i-th and j-th particles, respectively. Let us denote

$$\vec{f}^{recip}(\vec{k}) = \sum_{i=1}^{n}\exp(-i\vec{k}\cdot\vec{r}_i)\vec{f}_i$$

which can be regarded as the Fourier transform of the forces $\vec{f}$. We are now able to reuse $\vec{f}^{recip}(\vec{k})$ for computing $\vec{u}^{recip}_j$ for all particles j.

The main benefit of PME over Ewald is to sum complex exponentials using the FFT. This cannot be achieved as written above because the $\vec{r}_i$ are not equally spaced. Therefore, the PME method first spreads the forces (the spreading operation is the transpose of interpolation) onto a regular mesh in order to compute $\vec{f}^{recip}(\vec{k})$ via the FFT. Then, after multiplying $\vec{f}^{recip}(\vec{k})$ by $M^{(2)}_\alpha(\vec{k})$, the velocity is

$$\vec{u}^{recip}_j = \frac{1}{L^3}\sum_{\vec{k}\neq 0}\exp(i\vec{k}\cdot\vec{r}_j)M^{(2)}_\alpha(\vec{k})\vec{f}^{recip}(\vec{k}). \tag{3}$$

The sum over lattice vectors $\vec{k}$ corresponds to an inverse Fourier transform. Again, the result is computed on the regular mesh. The velocities computed on the mesh are finally interpolated onto the locations of the particles.

We now introduce the interpolation method. We use cardinal B-spline interpolation, as used in smooth PME (SPME) [7]. We found the SPME approach to be more accurate than the original PME approach [6] with Lagrangian interpolation, while negligibly increasing computational cost. In contrast to SPME with a scalar kernel such as for electrostatics, SPME with a 3×3 tensor kernel means that spreading is applied to 3×1 quantities. Let the force at particle i be denoted by $\vec{f}_i = [f^x_i, f^y_i, f^z_i]$. These forces are spread onto a $K\times K\times K$ mesh,

$$F^\theta(k_1,k_2,k_3) = \sum_{i=1}^{N}\vec{f}^\theta_i\, W_p(u^x_i-k_1)W_p(u^y_i-k_2)W_p(u^z_i-k_3) \tag{4}$$

where θ can be x, y or z, and u^x, u^y and u^z are scaled fractional coordinates of the particles, i. e., for a given particle located at (r^x, r^y, r^z), we have $u^\theta = r^\theta K/L$ for a cubical $L\times L\times L$ simulation box. The functions W_p are cardinal B-splines of order p (piecewise polynomials of degree $p-1$). For simplicity of notation, we assume that the W_p functions "wrap around" the periodic boundaries. Importantly, the W_p have compact support: they are nonzero over an interval of length p. Thus Equation (4) shows that each $\vec{f}^\theta_i$ is spread onto p^3 points of the mesh around the i-th particle. Figure 1 illustrates the spreading of a force onto a 2D mesh.

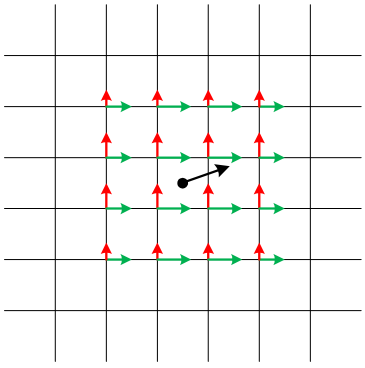


Figure 1: Spreading a force onto a 2D mesh using B-splines of order 4. The black dot represents a particle, and the black arrow represents a force on the particle. The spread forces on the mesh are represented by the red and green arrows.

After spreading the forces into the regular mesh, we can apply the FFT. The approximation to the θ component of

$\vec{f}^{recip}(\vec{k})$ on the mesh is $b_1(k_1)b_2(k_2)b_3(k_3)\mathcal{F}[F^\theta(k_1,k_2,k_3)]$ where $\mathcal{F}$ denotes the 3D FFT, and where the b functions are complex scalars which can be interpreted as coefficients of interpolating complex exponentials with B-splines (unnecessary when PME Lagrangian interpolation is used). The b functions are fixed given the interpolation scheme and have negligible computational cost. In practice, they are absorbed into the influence function to be described next, and we will not further discuss them in this paper.

Referring to Equation (3), the next step is to multiply the Fourier transform of the forces by $M_\alpha^{(2)}(\vec{k})$. Assuming uniform particle radii, $M_\alpha^{(2)}(\vec{k})$ derived in [22] is

$$\begin{aligned} M_\alpha^{(2)}(\vec{k}) =& (I-\hat{k}\hat{k}^T)m_\alpha(\|\vec{k}\|) \\ m_\alpha(\|\vec{k}\|) =& (a-\tfrac{1}{3}a^3\|\vec{k}\|^2)(1+\tfrac{1}{4}\alpha^{-2}\|\vec{k}\|^2+\tfrac{1}{8}\alpha^{-4}\|\vec{k}\|^4) \\ & \times 6\pi\|\vec{k}\|^{-2}\exp(-\tfrac{1}{4}\alpha^{-2}\|\vec{k}\|^2) \end{aligned} \quad (5)$$

where $\hat{k}$ is the normalization of $\vec{k}$, a is the radius of the particles, $(I-\hat{k}\hat{k}^T)$ is a 3×3 tensor, and $m_\alpha(\|\vec{k}\|)$ is a scalar function. Computationally, $M_\alpha^{(2)}(\vec{k})$ is a 3×3 symmetric matrix defined at each of the $K\times K\times K$ mesh points. We refer to this structure as the *influence function*, $I(k_1,k_2,k_3)$. Defining $C^\theta(k_1,k_2,k_3)=\mathcal{F}[F^\theta(k_1,k_2,k_3)]$, applying the influence function means computing

$$D^\theta(k_1,k_2,k_3)=I(k_1,k_2,k_3)\cdot C^\theta(k_1,k_2,k_3). \quad (6)$$

The velocities on the mesh can be computed as

$$U^\theta(k_1,k_2,k_3)=\mathcal{F}^{-1}[D^\theta(k_1,k_2,k_3)]$$

where $\mathcal{F}^{-1}$ denotes the 3D inverse FFT. The particle velocities $\vec{u}^{recip}$ can be computed by interpolating U^θ onto the locations of the particles, which is the reverse process of spreading.

In summary, each PME operation effectively multiplies the mobility matrix by a given vector of forces $\vec{f}$ for the particles located at $\vec{r}$. In the following, we will use $\vec{u}=PME(\vec{f})$ to denote this operation, given some particle configuration $\vec{r}$. The creation of a PME operator in software includes the construction of the sparse matrix M^{real} and other preprocessing steps needed for applying M^{recip}.

B. Computing Brownian Displacements with PME

The canonical method of computing Brownian displacements, which uses Cholesky factorization, requires M to be available as a matrix. With PME, such an explicit form for M is not available. We recently introduced the use of Krylov subspace methods for computing Brownian displacements [8], [23]. These methods only require the ability to perform matrix-vector products with M. Thus this method is suitable for computing Brownian displacements in the PME context.

Since in BD we can use the same mobility matrix for several time steps, we use a *block* Krylov subspace method to compute Brownian displacement vectors for multiple time steps simultaneously. This has the benefit of (a) fewer total number of iterations are required for convergence than the single vector Krylov method, leading to lower computational cost per vector [8], and (b) the SpMV operation for computing the real-space term is applied to a block of vectors, which is more efficient than single vector SpMV [24].

Other methods for computing Brownian displacements in matrix-free form are available, but they require eigenvalue estimates of M, e.g., [25]. Although not our emphasis, Krylov subspace methods are combined with PME for the first time in this paper. See [8] for technical details on Krylov subspace methods for computing Brownian displacements.

C. Matrix-Free BD Algorithm

The matrix-free BD algorithm is shown in Algorithm 2. Krylov(PME,Z) denotes an application of the Krylov subspace method, in which the products of the mobility matrix with a given vector $\vec{f}$ are evaluated by $PME(\vec{f})$. Since the mobility matrix changes slowly over time steps, we can use the particle configuration at the current time step to compute the Brownian displacement vectors for the following λ_{RPY} time steps. The sparse matrix M^{real} is only updated (at line 4) every λ_{RPY} time steps.

Algorithm 2: Matrix-free BD algorithm for m time steps. $\vec{r}_k$ denotes the position vector at time step k.

```
1  k ← 0
2  n ← m/λ_RPY
3  for i ← 1 to n do
4  |   Construct PME operator using r_k
5  |   Generate λ_RPY random vectors Z = [z_1, ..., z_λRPY]
6  |   Compute D = [d_1, ..., d_λRPY] = Krylov(PME, Z)
7  |   for j ← 1 to λ_PRY do
8  |   |   Compute f(r_k)
9  |   |   Update r_{k+1} = r_k + PME(f(r_k))Δt + d_j
10 |   |   k ← k + 1
11 |   end
12 end
```

IV. Efficient Implementation of PME on Multicore and Manycore Architectures

A. Reformulating the Reciprocal-Space Calculation

In molecular dynamics simulations, PME is not applied more than once for a given particle configuration. In our work, we apply PME iteratively in Krylov subspace methods to compute Brownian displacements. Thus our optimization of PME involves a setup phase where precomputation is used to speed up the actual PME computations.

In this section, we reformulate the interpolation and spreading operations in the reciprocal-space calculation of PME as sparse matrix-vector products. Also, for our tensor kernel, applying the influence function can be regarded as a matrix-vector product for a block diagonal matrix with 3×3 blocks. The breakdown of the reciprocal-space calculation into these kernels also allows us to implement an efficient PME code in a relatively portable way.

Define the $n\times K^3$ interpolation matrix P as

$$P(i,\ k_1K^2+k_2K+k_3)=W_p(u_i^x-k_1)W_p(u_i^y-k_2)W_p(u_i^z-k_3) \quad (7)$$

which is a sparse matrix with p^3 nonzeros per row (we have assumed 1-based indexing for i and 0-based indexing for k_1,

k_2, k_3). The spreading of forces can then be expressed as

$$[F^x, F^y, F^z] = P^T \times [\vec{f}^x, \vec{f}^y, \vec{f}^z] \quad (8)$$

and, similarly, the interpolation of velocities is

$$[(\vec{u}^{recip})^x, (\vec{u}^{recip})^y, (\vec{u}^{recip})^z] = P \times [U^x, U^y, U^z]. \quad (9)$$

After the reformulation, the reciprocal-space calculation of PME can be performed in six steps:

(1) *Constructing P*: Precomputation of the interpolation matrix P (Equation (7)).

(2) *Spreading*: Spreading of the forces onto the mesh array F^θ (Equation (8)). This has been reformulated as a sparse matrix-vector product.

(3) *Forward FFT*: Applying 3D fast Fourier transform to compute $C^\theta = \mathcal{F}[F^\theta]$.

(4) *Influence function*: Multiplying C^θ by the influence function I (Equation (6)).

(5) *Inverse FFT*: Applying 3D inverse fast Fourier transform $U^\theta = \mathcal{F}^{-1}[D^\theta]$.

(6) *Interpolation*: Interpolating the velocities on the locations of the particles (Equation (9)).

There are two main benefits of this reformulation. First, our matrix-free BD algorithm uses the Krylov subspace method to compute Brownian displacements, which requires computing PME multiple times at each simulation step. Since P only depends on the positions of the particles, with this reformulation we only need to precompute P once at the beginning of each simulation step when the PME operator is constructed (line 4 in Algorithm 2), and reuse it for all the PME computations within the step. This significantly reduces the computational cost. In addition, this reformulation transforms spreading and interpolation into SpMV operations for which high-performance implementations are available, including on accelerators.

B. Optimizing the Reciprocal-Space Calculation

1) Constructing P: The precomputation of the interpolation matrix P is performed in parallel, with P partitioned into row blocks, one for each thread. SIMD instructions are used by each thread to compute multiple rows concurrently in a row block. It is natural to store P in Compressed Sparse Row (CSR) format. However, the row pointers are not necessary since all rows of P have the same number of nonzeros (each force is spread onto the same number of FFT mesh points).

2) Spreading: The spreading step multiplies the transpose of P by the vector of forces. Since P is stored in CSR format, different threads will try to update the same memory locations in the result. To alleviate this contention, one might explicitly transpose P and store it in Compressed Sparse Column (CSC) format. However, CSC format can be inefficient for storing P since P is typically "short-and-fat" and contains many empty columns (corresponding to mesh points that receive no spreading contributions from particles).

In order to efficiently parallelize the P^T operation, we partition the mesh points into square blocks with dimensions no less than $p \times p$, where p is the interpolation order. Those blocks are then partitioned into groups such that there are no two blocks in one group that are adjacent to each other. We call such a group an *independent set*. There are eight independent sets in a 3D mesh. By observing that the particles from different blocks in the same independent set own distinct columns of P, the forces for those particles can be spread in parallel without write contention. Figure 2 illustrates independent sets in a 2D mesh, in which different independent sets are shown in different colors. In our parallel spreading implementation, the particles are first mapped into the blocks. The parallel SpMV is then performed in eight stages, and each stage only multiplies the rows of P that are associated with the particles from one independent set.

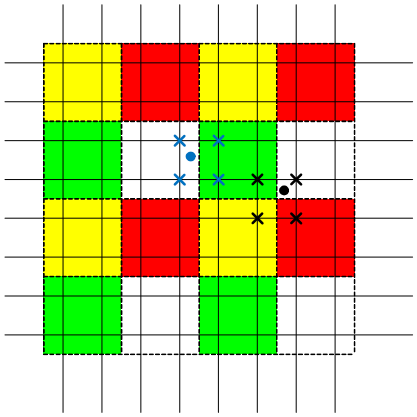


Figure 2: Independent sets for $p = 2$ in a 2D mesh. The 4 independent sets (8 in 3D) are marked in different colors. Two particles (dots) from different blocks in the same independent set cannot spread to the same mesh points (crosses).

Note that for a given set of particle positions at a time step, some mesh points may have no contribution from particles. Thus we explicitly set the result F^θ to zero before beginning the spreading operation.

3) 3D FFTs: We use the Intel MKL to perform 3D FFTs. The library contains in-place real-to-complex forward FFT and complex-to-real inverse FFT routines. This halves the memory and bandwidth requirements compared to the case if only complex-to-complex routines were available. Similarly, only half of the influence function is needed, which can be stored in an array of size $K \times K \times (K/2+1)$, where each array entry itself is a symmetric 3×3 matrix.

4) Applying the Influence Function: The influence function I only depends on the simulation box size L, the mesh size K, and the interpolation order p; it can thus be precomputed and used for any particle configuration. However, the memory required for explicitly storing I is approximately $6 \times 8 \times K^3/2$ bytes, which is 3 times larger than the storage requirement for F^θ, making this approach impractical for Intel Xeon Phi and other accelerators that have limited memory. On the other hand, constructing I on-the-fly, every time it is needed, requires evaluating exponential functions, which are costly to compute.

We observe from Equation (5), however, that the influence function is the product of a 3×3 tensor and a relatively expensive scalar function, m_α. We precompute and store this scalar function rather than the 3×3 tensor, giving a savings of a factor of 6. When applying the influence function, the quantity $(I - \hat{k}\hat{k}^T)$, which only depends on the lattice vector, can be constructed without memory accesses. Applying the influence function is memory bandwidth bound, due to the

small number of flops executed compared to data transferred for reading C^θ and writing D^θ.

5) Interpolation: In our reformulation, interpolating the velocities from the mesh points is performed by a sparse matrix-vector product. For this, we have implemented a parallel SpMV kernel for multicore processors. The optimized SpMV kernel in the CSR format on Intel Xeon Phi can be found in our previous work [26].

C. Computation of Real-Space Terms

The real-space sum of the PME method is performed by interacting pairs of particles within a short cutoff radius, depending on the Ewald parameter α. Since this operation must be repeated in the BD algorithm, we store the real-space sum operator, M^{real}, as a sparse matrix. To construct this sparse matrix, only the RPY tensors between particles within a short distance need to be evaluated, which we compute efficiently in linear time using Verlet cell lists [27].

This sparse matrix has 3×3 blocks, owing to the tensor nature of the RPY tensor. We thus store the sparse matrix in Block Compressed Sparse Row (BCSR) format. Previously, we optimized SpMV for this matrix format, using thread and cache blocking, as well as code generation to produce fully-unrolled SIMD kernels for SpMV with a block of vectors [24]. The latter is required in Algorithm 2 as multiple time steps are taken with the same mobility matrix, and thus it is possible and efficient to operate on multiple vectors simultaneously.

D. Performance Modelling and Analysis

The performance of each step of PME is modeled separately. We focus on the reciprocal-space part, since the real-space part is a straightforward sparse matrix-vector product. We also exclude the construction of P from our analysis, since it is a preprocessing step when PME is applied to multiple force vectors with the same particle configuration.

(a) *Spreading*: The total memory traffic, in bytes, incurred by spreading is

$$M_{spreading} = (3 \times 8 \times K^3) + (12 \times p^3 n) + (3 \times 8 \times p^3 n)$$

where the first term is the memory traffic due to the initialization of F^θ, the second term is the memory footprint of P which includes non-zeros and CSR column indices, and the last term represents the memory traffic due to writing the product of $P^T \vec{f}^\theta$ to F^θ. Since the spreading step is performed by SpMV, the performance is bounded by memory bandwidth. The execution time is estimated as

$$T_{spreading} = \frac{M_{spreading}}{B}$$

where B is the hardware memory bandwidth.

(b) *3D FFTs*: Each PME operation for the RPY tensor requires three forward 3D FFTs and three inverse 3D FFTs with dimensions $K \times K \times K$. We model the execution time as

$$T_{FFT} = 3 \times 2.5K^3 \log_2(K^3)/P_{FFT}(K)$$

$$T_{IFFT} = 3 \times 2.5K^3 \log_2(K^3)/P_{IFFT}(K)$$

where the numerators are the number of flops required for radix-2 3D FFTs, and where the denominators $P_{FFT}(K)$ and $P_{IFFT}(K)$ are the achievable peak flop rates of forward and inverse 3D FFTs, respectively.

(c) *Applying Influence Function*: Our implementation of applying the influence function constructs stores only one word per 3×3 tensor. Thus, the memory traffic due to accessing I is only $8 \times K^3/2$. As discussed in Section IV-B, the performance of this step is memory bandwidth bound. Therefore, the execution time can be expressed as

$$T_{influence} = \frac{(8 \times K^3/2) + (2 \times 3 \times 16 \times K^3/2)}{B}$$

where the second term in the numerator is the total memory footprint of C^θ and D^θ.

(d) *Interpolation*: The memory traffic incurred by interpolation is similar to that of spreading except that interpolation does not require initializing U^θ. Its execution time can be expressed as

$$T_{interpolation} = \frac{(12 \times p^3 n) + (3 \times 8 \times p^3 n)}{B}.$$

The overall performance model of the reciprocal-space PME calculation is the sum of the terms above,

$$\begin{aligned} T_{PME} = & \frac{7.5K^3 \log_2(K^3)}{P_{FFT}(K)} + \frac{7.5K^3 \log_2(K^3)}{P_{IFFT}(K)} \\ & + \frac{72p^3 n + 76K^3}{B}. \end{aligned} \tag{10}$$

The memory requirement in bytes for this part of PME can be expressed as

$$M_{PME} = (3 \times 8 \times K^3) + (12 \times p^3 n) + (8 \times K^3/2) \tag{11}$$

where the first term is the storage for F^θ and U^θ (or C^θ and D^θ), the second term represents the memory footprint of P, and the last term is the storage for the influence function. Since the number of mesh points, K^3, is generally chosen to be proportional to the number of particles n (assuming fixed volume fraction), the reciprocal-space part of PME scales as $O(n \log n)$ and requires $O(n)$ storage.

E. Hybrid Implementation on Intel Xeon Phi

With the advance of acceleration hardware such as GPUs and Intel Xeon Phi, it is important to study the coupling of general-purpose processors with accelerators to solve various computational problems. In this section, we describe a hybrid implementation of the matrix-free BD method using CPUs and Intel Xeon Phi.

In the PME method, the real-space terms and the reciprocal-space terms can be computed concurrently. It is natural to offload one of these for computation on accelerators. It is preferable to offload the reciprocal-space calculation, since it consists of regularly structured calculations, e.g., FFTs suitable for wide-SIMD operations, which also demand high memory bandwidth.

To balance the workload between CPUs and Intel Xeon Phi, the Ewald parameter α is tuned so that one real-space calculation on the CPU and one reciprocal-space calculation on the accelerator consume approximately equal amounts of execution time. To predict the execution time, we use the performance models presented in Section IV-D. This works for computing the PME operation in line 9 of Algorithm 2.

The PME operation shown in line 6, however, involves a block of vectors. Here, the real-space part is performed very efficiently with SpMV on a block of vectors. There is no library function, however, for 3D FFTs for blocks of vectors. The reciprocal-space part thus effectively has higher workload when PME is applied on a block of vectors. (The Ewald parameter α may be increased to balance the workload for this case, but practically α is limited if sparsity and scalable storage is to be maintained for the real-space part.) The solution we adopt for the PME operation in line 6 of Algorithm 2 is to also assign to CPUs some reciprocal-space calculations. A static partitioning of the reciprocal-space calculations is performed to achieve load balance between CPUs and multiple Xeon Phi coprocessors.

V. Experimental Results

A. Test-Bed and Simulation Setup

Our experimental test-bed is a dual socket Intel Xeon X5680 (Westmere-EP) system with two Intel Xeon Phi coprocessors (KNC) mounted on PCI-e slots. The key architectural parameters are listed in Table I.

Table I: Architectural parameters of systems used in performance evaluation.

	2X Intel X5680	Intel Xeon Phi
Microarchitecture	Westmere-EP	MIC
Frequency (GHz)	3.33	1.09
Sockets/Cores/Threads	2/12/24	1/61/244
L1/L2/L3 cache (KB)	64/256/12288	32/512/-
SIMD width (DP, SP)	2-way, 4-way	8-way, 16-way
GFlop/s (DP, SP)	160, 320	1074, 2148
Memory (GB)	24	8
STREAM bandwidth (GB/s)	44	150

Our BD implementations were compiled with Intel ICC 14.0. Intel MKL 11.0 was used to optimize the FFT, BLAS and LAPACK operations in the implementations, including DGEMM, DGEMV, Cholesky factorization, and forward/inverse FFTs.

A monodisperse suspension model of n particles with various volume fractions was used to evaluate the accuracy and the performance of the BD algorithms. For simplicity, van der Waals or electrostatic interactions were not included in the model. To prevent particle overlap, a repulsive harmonic potential between particles was used. The repulsion force between particles i and j both with radius a is given by

$$\vec{f}_{ij}^{repl} = \begin{cases} 125(\|\vec{r}_{ij}\| - 2a)\hat{r}_{ij} & \text{if } \|\vec{r}_{ij}\| \leq 2a \\ 0 & \text{if } \|\vec{r}_{ij}\| > 2a \end{cases}$$

where $\vec{r}_{ij}$ is the vector between particle i and j, and $\hat{r}_{ij}$ is the normalized vector. The repulsion forces were efficiently evaluated using Verlet cell lists [27].

In BD simulations, the translational diffusion coefficients of particles can be estimated by

$$D(\tau) = \frac{1}{6\tau}\left\langle((\vec{r}(t+\tau) - \vec{r}(t))^2\right\rangle \qquad (12)$$

where the angle brackets indicate an average over configurations separated by a time interval τ and $\vec{r}$ is the position vector of the particles. For a given BD algorithm, its accuracy can be evaluated by comparing the diffusion coefficients obtained from simulation with theoretical values, values obtained from experiments, or simply values from a known, separately validated simulation.

B. Accuracy of the Matrix-Free BD Algorithm

For a given α, the accuracy of the PME calculation is controlled by the cutoff distance r_{max}, the mesh dimension K, and the B-spline order p. Using larger r_{max}, K and/or p gives a more accurate result with a more expensive calculation. We measure the relative error of PME as

$$e_p = \frac{\|\vec{u}^{pme} - \vec{u}^{exact}\|_2}{\|\vec{u}^{exact}\|_2}$$

where $\vec{u}^{pme}$ is the result of PME, and $\vec{u}^{exact}$ is a result computed with very high accuracy, possibly by a different method.

The accuracy of the simulation also depends on the accuracy of the Krylov subspace iterations for computing the Brownian displacements. We denote by e_k the relative error tolerance used to stop the iterations.

Parameters for PME and the Krylov subspace method must be chosen to balance computational cost and accuracy. In order to choose these parameters, we performed simulations with different sets of parameters and evaluated the resulting accuracy of the diffusion coefficients obtained from these simulations. Table II shows the results. We see that the matrix-free algorithm with $e_k = 10^{-6}$ and $e_p \sim 10^{-6}$ ($e_p \sim 10^{-k}$ means we used parameters giving measured PME relative error between $10^{-(k+1)}$ and 10^{-k}) has a relative error less than 0.25%. The simulations with larger e_k and e_p also achieve good accuracy. Even with $e_k = 10^{-2}$ and $e_p \sim 10^{-3}$, the average relative error is still lower than 3%. Using larger e_k and e_p significantly reduces running time. The simulations with $e_k = 10^{-2}$ and $e_p \sim 10^{-3}$ are more than 8x faster than those with $e_k = 10^{-6}$ and $e_p \sim 10^{-6}$.

Table II: Errors (%) in diffusion coefficients obtained from simulations using the matrix-free BD algorithm with various Krylov tolerances (e_k) and various PME parameters (giving PME relative error e_p). Also shown is the execution time (seconds) per simulation step using 2 Xeon CPUs. Simulated systems were particle suspensions of 1,000 particles for various volume fractions Φ.

	$e_k = 10^{-6}$ $e_p \sim 10^{-6}$		$e_k = 10^{-2}$ $e_p \sim 10^{-6}$		$e_k = 10^{-6}$ $e_p \sim 10^{-3}$		$e_k = 10^{-2}$ $e_p \sim 10^{-3}$	
Φ	Error	Time	Error	Time	Error	Time	Error	Time
0.1	0.06	0.089	0.28	0.029	0.42	0.036	0.65	0.010
0.2	-0.09	0.102	-0.22	0.032	0.56	0.047	1.48	0.011
0.3	-0.21	0.116	-0.37	0.032	2.05	0.054	2.38	0.012
0.4	0.25	0.130	0.46	0.036	1.28	0.061	3.72	0.013
0.5	0.16	0.130	0.62	0.038	-3.69	0.062	4.27	0.013

As an example of a BD calculation, Figure 3 shows the diffusion coefficients obtained from matrix-free BD simulations of 5000 particles and various volume fractions. Simulations were performed on our test-bed (using hybrid CPU-accelerator computations) for 500,000 steps with $\lambda_{RPY} = 16$, $e_k = 10^{-2}$ and $e_p \sim 10^{-3}$, taking a total of 10 hours. The diffusion coefficients obtained from the simulations are in good agreement with theoretical values. Qualitatively, the diffusion coefficients are smaller for systems with higher volume fractions (more crowded conditions). Such long simulations also illustrate the importance of reducing wallclock time per timestep, as well as enabling larger simulations.

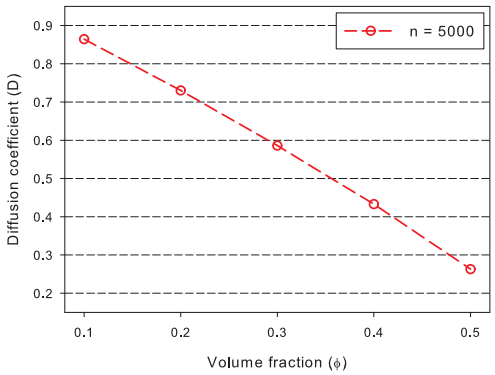


Figure 3: Diffusion coefficients (D) obtained from the simulations using the matrix-free algorithm on a configuration of 5,000 particles with various volume fractions.

C. Simulation Configurations

Table III shows the simulation configurations used in our experiments. For each configuration, the PME parameters were chosen such that execution time is minimized while keeping the PME relative error e_p less than 10^{-3}. (The procedure for choosing these parameters is beyond the scope of this paper.) The Krylov convergence tolerance $e_k = 10^{-2}$ was used in all the experiments. In a separate study, we found that if the mesh spacing L/K is fixed, then the reciprocal-space error is independent of the volume fraction. Also, the real-space error only weakly depends on the volume fraction. Therefore, for studying performance, we use a single volume fraction, which we have chosen to be 0.2.

Table III: Simulation configurations, where n is the number of particles, K is the PME FFT mesh dimension, p is the B-spline order, r_{max} is the cutoff distance used for M^{real}, α is the Ewald parameter, and e_p is the PME relative error.

Configurations	n	K	p	r_{max}	α	e_p
N100	100	32	4	6.0	0.58	7.05×10^{-4}
N500	500	32	6	6.0	0.58	9.88×10^{-4}
N1000	1,000	32	6	7.5	0.46	6.70×10^{-4}
N2000	2,000	64	6	6.0	0.58	5.88×10^{-4}
N3000	3,000	64	6	6.0	0.58	4.88×10^{-4}
N4000	4,000	64	6	6.0	0.58	8.31×10^{-4}
N5000	5,000	64	6	6.5	0.52	6.82×10^{-4}
N6000	6,000	64	6	6.5	0.52	7.92×10^{-4}
N7000	7,000	64	6	6.5	0.52	8.84×10^{-4}
N8000	8,000	64	6	7.0	0.50	8.81×10^{-4}
N10000	10,000	64	6	7.5	0.46	5.44×10^{-4}
N20000	20,000	128	6	6.0	0.58	3.24×10^{-4}
N30000	30,000	128	6	6.0	0.58	4.15×10^{-4}
N50000	50,000	128	6	6.0	0.58	9.90×10^{-4}
N80000	80,000	128	6	7.0	0.50	9.47×10^{-4}
N200000	200,000	256	4	6.0	0.58	9.86×10^{-4}
N300000	300,000	256	6	6.0	0.58	5.82×10^{-4}
N500000	500,000	256	6	7.0	0.50	3.39×10^{-4}

D. Performance of PME

In matrix-free BD, the same PME operator is applied to different vectors. This is a significant difference from molecular dynamics (or non-Brownian simulations) where a given PME operator is only applied once. Thus an important optimization in our BD case is the precomputation and reuse of the interpolation matrix P. To study the effect of this optimization, we compare the performance of simulations using precomputed P and the application of P on-the-fly (in the latter, P is not stored). The application of P on-the-fly has the advantage of lower memory bandwidth requirements since the elements of P are computed using only the particle positions.

Figure 4 reports this comparison using a CPU-only implementation. In both cases, we used $\lambda_{RPY} = 16$. The number of Krylov subspace iterations for the configurations shown in the figure varies between 19 and 25, meaning a precomputation of P will be reused more than 300 times. The results show that precomputing P gives on average 1.5x speedup compared to using the on-the-fly implementation. The largest speedup is achieved by the configurations with larger values of p^3n/K^3 (*N10000*, *N80000*, *N500000*). This is because the complexity of computing P is a function of p^3n and the computational cost of the other steps is mainly dependent on K^3.

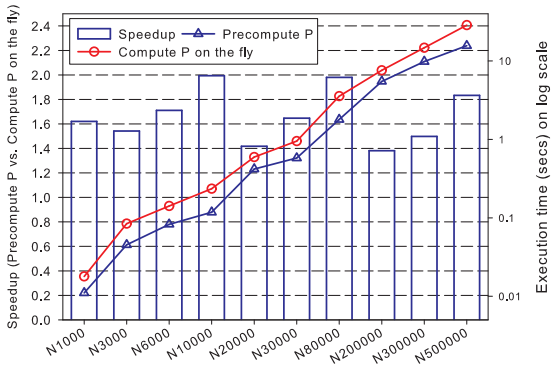
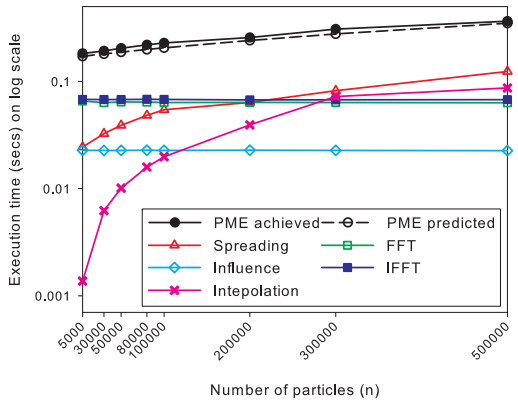


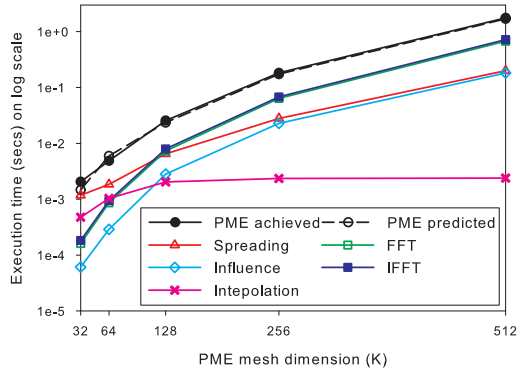
Figure 4: Performance comparison of the PME implementation (reciprocal-space part only) that precomputes P with the implementation that computes P on-the-fly.

The overall performance of the reciprocal-space part of PME is shown in Figure 5 as a function of the number of particles and of the PME mesh dimension. The break down of the time for each phase is also shown. Timings are for the CPU-only implementation. The main observation is that FFT operations generally dominate the execution time. However, the execution time of spreading and interpolation operations increases rapidly with the number of particles. These operations are memory bandwidth limited and are very costly for large numbers of particles, and can surpass the cost of the FFTs. We also observe that applying the influence function, although it is embarrassingly parallel, also becomes costly for large mesh dimensions. This is due to the high bandwidth requirements of applying the influence function. Finally, the figures also show that the achieved performance closely matches the predicted performance from the performance model, indicating that our CPU-only PME implementation is as efficient as possible.

In Figure 6 we compare the performance of the reciprocal-space part of PME on two different architectures: Westmere-EP and KNC in native mode. For small numbers of particles, KNC is only slightly faster than or even slower than Westmere-EP. This is mainly due to inefficient FFT implementations in MKL on KNC, particularly for the 3D inverse FFT (this is currently being resolved by Intel). For large numbers of particles, KNC is as much as 1.6x faster.



(a) $K = 256, p = 6, r_{max} = 5$



(b) $n = 5000, p = 6, r_{max} = 5$

Figure 5: The overall performance of the reciprocal-space part of PME and the break down of execution time for each phase as a function of the number of particles and the PME mesh dimension.

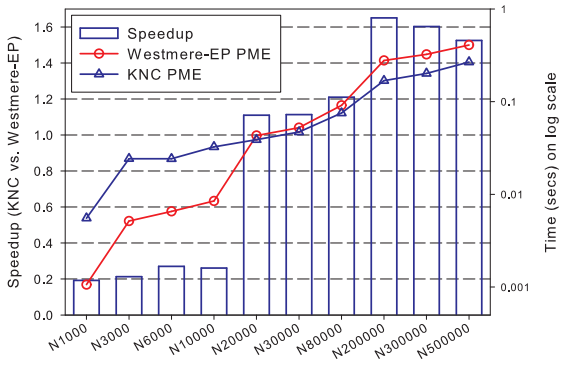
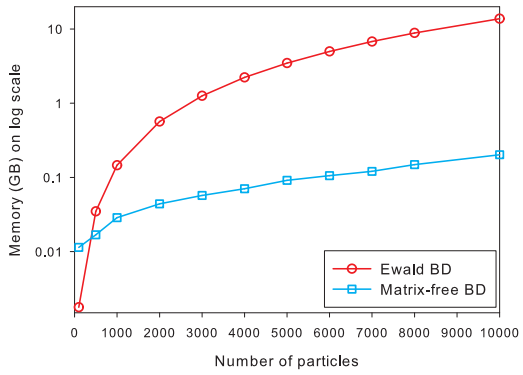


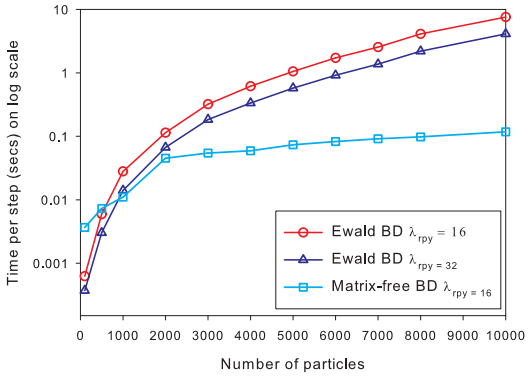
Figure 6: Performance comparison of PME on Westmere-EP with PME on KNC in native mode.

E. Performance of BD Simulations

Now we present the overall performance of simulations using the Ewald BD algorithm and the matrix-free algorithm. Figure 7 shows the results. The methods used parameters giving results of similar accuracy. As expected, the matrix-free algorithm has great advantages over the Ewald algorithm both in the terms of memory usage and execution time. For large configurations, the speedup of the matrix-free algorithm



(a) Memory



(b) Execution time

Figure 7: Comparison of the Ewald BD algorithm with the matrix-free BD algorithm on Westmere-EP as a function of the number of particles.

over the Ewald algorithm is more than 35x. (For problems of this size, the standard algorithm can be improved by using Krylov subspace methods rather than Cholesky factorizations for computing Brownian displacements.) However, the more important result is that, as shown in Figure 8, our implementation of the matrix-free algorithm is capable of performing BD simulations for as many as 500,000 particles.

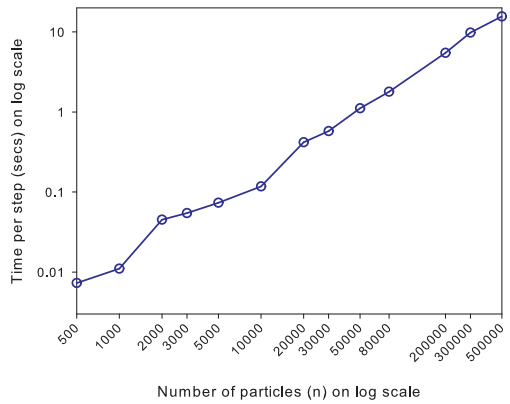


Figure 8: Performance of the matrix-free BD algorithm on Westmere-EP as a function of the number of particles.

Figure 9 compares the performance of our hybrid BD

implementation using two Intel Xeon Phi coprocessors with the CPU-only implementation. The hybrid implementation is always faster than the CPU-only implementation for all the configurations, achieving on average a speedup of 2.5x. The largest speedup is achieved by very large configurations, which is more than 3.5x. For small configurations, the advantage of the hybrid implementation over the CPU-only implementation is marginal probably because of two reasons. First, the PME implementation on KNC for small configurations is not efficient. Second, for the small configurations there is not enough work to compensate for the communication overhead of offloading.

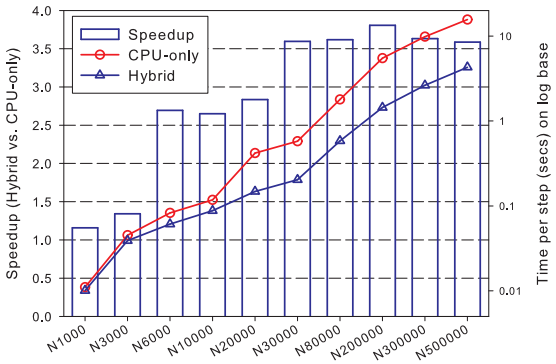


Figure 9: Performance comparison of the hybrid BD implementation with the CPU-only implementation.

VI. Conclusion

We have presented a matrix-free algorithm for Brownian dynamics simulations with hydrodynamic interactions for large-scale systems. The algorithm uses the PME method, along with a block Krylov subspace method to compute Brownian displacements in matrix-free fashion. Our software using this algorithm enables large-scale simulations with as many as 500,000 particles.

We have also described the implementation of the matrix-free algorithm on multicore and manycore processors. The PME algorithm was expressed as the application of a sequence of kernels (SpMV, FFT) to simplify efficient implementation. We have also developed a hybrid implementation of BD simulations using multiple Intel Xeon Phi coprocessors, which can be 3.5x faster than the CPU-only implementation. In future work, we will extend the functionality of the BD simulation code and use it to simulate large-scale biological systems.

Acknowledgments

This work was supported by the National Science Foundation under grant ACI-1306573.

References

[1] P. Mereghetti, R. R. Gabdoulline, and R. C. Wade, "Brownian dynamics simulation of protein solutions: structural and dynamical properties," *Biophysical Journal*, vol. 99, pp. 3782–3791, 2010.

[2] M. Długosz, J. M. Antosiewicz, and J. Trylska, "Association of aminoglycosidic antibiotics with the ribosomal A-site studied with Brownian dynamics," *Journal of Chemical Theory and Computation*, vol. 4, pp. 549–559, 2008.

[3] T. Ishikawa, G. Sekiya, Y. Imai, and T. Yamaguchi, "Hydrodynamic interactions between two swimming bacteria," *Biophysical Journal*, vol. 93, pp. 2217–2225, 2007.

[4] J. Kotar, M. Leoni, B. Bassetti, M. C. Lagomarsino, and P. Cicuta, "Hydrodynamic synchronization of colloidal oscillators," *Proc Natl Acad Sci USA*, vol. 107, pp. 7669–7673, 2010.

[5] T. Frembgen-Kesner and A. H. Elcock, "Striking effects of hydrodynamic interactions on the simulated diffusion and folding of proteins," *Journal of Chemical Theory and Computation*, vol. 5, pp. 242–256, 2009.

[6] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald – an $N\log(N)$ method for Ewald sums in large systems," *Journal of Chemical Physics*, vol. 98, pp. 10 089–10 092, 1993.

[7] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, "A smooth particle mesh Ewald method," *Journal of Chemical Physics*, vol. 103, pp. 8577–8593, 1995.

[8] T. Ando, E. Chow, Y. Saad, and J. Skolnick, "Krylov subspace methods for computing hydrodynamic interactions in Brownian dynamics simulations," *The Journal of Chemical Physics*, vol. 137, p. 064106, 2012.

[9] M. Długosz, P. Zieliński, and J. Trylska, "Brownian dynamics simulations on CPU and GPU with BD_BOX," *J. Comput. Chem.*, vol. 32, pp. 2734–2744, 2011.

[10] T. Geyer, "Many-particle Brownian and Langevin dynamics simulations with the Brownmove package," *BMC Biophysics*, vol. 4, p. 7, 2011.

[11] V. B. Putz, J. Dunkel, and J. M. Yeomans, "CUDA simulations of active dumbbell suspensions," *Chemical Physics*, vol. 375, pp. 557–567, 2010.

[12] F. E. Torres and J. R. Gilbert, "Large-Scale Stokesian Dynamics Simulations of Non-Brownian Suspensions," Xerox Research Centre of Canada, Tech. Rep., 1996.

[13] R. C. Ball and J. R. Melrose, "A simulation technique for many spheres in quasistatic motion under frame-invariant pair drag and Brownian forces," *Physica A*, vol. 247, p. 444, 1997.

[14] S. Plimpton, "Fast parallel algorithms for short-range molecular-dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[15] E. K. Guckel, "Large scale simulation of particulate systems using the PME method," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1999.

[16] A. Sierou and J. F. Brady, "Accelerated Stokesian dynamics simulations," *Journal of Fluid Mechanics*, vol. 448, pp. 115–146, 2001.

[17] D. Saintillan, E. Darve, and E. S. Shaqfeh, "A smooth particle-mesh Ewald algorithm for Stokes suspension simulations: The sedimentation of fibers," *Physics of Fluids*, vol. 17, pp. 033 301–033 301, 2005.

[18] J. Rotne and S. Prager, "Variational treatment of hydrodynamic interaction in polymers," *Journal of Chemical Physics*, vol. 50, pp. 4831–4837, 1969.

[19] H. Yamakawa, "Transport properties of polymer chains in dilute solution: Hydrodynamic interaction," *Journal of Chemical Physics*, vol. 53, pp. 436–443, 1970.

[20] M. Harvey and G. De Fabritiis, "An implementation of the smooth particle mesh Ewald method on GPU hardware," *Journal of Chemical Theory and Computation*, vol. 5, pp. 2371–2377, 2009.

[21] D. L. Ermak and J. A. McCammon, "Brownian dynamics with hydrodynamic interactions," *The Journal of Chemical Physics*, vol. 69, pp. 1352–1360, 1978.

[22] C. W. J. Beenakker, "Ewald sums of the Rotne-Prager tensor," *The Journal of Chemical Physics*, vol. 85, pp. 1581–1582, 1986.

[23] T. Ando, E. Chow, and J. Skolnick, "Dynamic simulation of concentrated macromolecular solutions with screened long-range hydrodynamic interactions: Algorithm and limitations," *The Journal of Chemical Physics*, vol. 139, p. 121922, 2013.

[24] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, "Improving the performance of dynamical simulations via multiple right-hand sides," in *Proc. 2012 IEEE Parallel & Distributed Processing Symposium*. IEEE, 2012, pp. 36–47.

[25] M. Fixman, "Construction of Langevin forces in the simulation of hydrodynamic interaction," *Macromolecules*, vol. 19, pp. 1204–1207, 1986.

[26] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. 27th International Conference on Supercomputing*. ACM, 2013, pp. 273–282.

[27] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*. Oxford: Clarendon Press, 1989.