Performance Optimization of Reactive Molecular Dynamics Simulations With Dynamic Charge Distribution Models on Distributed Memory Platforms

Kurt A. O'Hearn*
ohearnku@msu.edu
Michigan State University
East Lansing, Michigan

Abdullah Alperen*
alperena@msu.edu
Michigan State University
East Lansing, Michigan

Hasan Metin Aktulga
hma@msu.edu
Michigan State University
East Lansing, Michigan

ABSTRACT

Reactive molecular dynamics (MD) simulations are important for high-fidelity simulations of large systems with chemical reactions. Iterative linear solvers used to dynamically determine atom polarizations in reactive MD models and redundancies related to bond order calculations constitute significant bottlenecks in terms of time-to-solution and the overall scalability of reactive force fields. The objective of this work is to address these bottlenecks. To accomplish this goal, several optimizations are explored including acceleration of the charge model solver through an effective preconditioning technique and a numerical method with reduced communication overheads, as well as initialization and data structure changes for bond order calculations. Detailed scalability analysis of these optimizations and their overall impact is presented. A single-allreduce pipelined non-blocking conjugate gradient (PIPECG) solver coupled with a sparse approximate inverse (SAI) based preconditioner has been observed to yield significant speedups over the baseline standard CG solver with Jacobi preconditioner. These results are significant as they can facilitate scalable simulations of large reactive systems, and presented techniques can be used in other polarizable MD models.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; • **Computing methodologies → Molecular simulation**; **Massively parallel and high-performance simulations**; • **Mathematics of computing** → *Solvers*.

KEYWORDS

reactive molecular dynamics, iterative sparse solvers, distributed preconditioners, communication hiding techniques

ACM Reference Format:
Kurt A. O'Hearn, Abdullah Alperen, and Hasan Metin Aktulga. 2019. Performance Optimization of Reactive Molecular Dynamics Simulations With Dynamic Charge Distribution Models on Distributed Memory Platforms. In *ICS '19: ACM International Conference on Supercomputing, June 26–28, 2019, Phoenix AZ*. ACM, New York, NY, USA, 10 pages. https://doi.org/doidoidoi

*Authors contributed equally

ICS '19, June 26–28, 2019, Phoenix AZ

ACM ISBN isbnisbn…$xx.yy
https://doi.org/doidoidoi

1 INTRODUCTION

Molecular dynamics (MD) simulations play increasingly important roles in diverse fields, ranging from biophysics to chemistry to materials science. Classical MD techniques rely on static bonds and fixed partial charges associated with atoms, limiting their applicability to non-reactive systems. To study phenomena involving chemical reactions, quantum mechanical (QM) methods have typically been the method of choice. QM simulations must account for electronic degrees of freedom present in the system. As such, they are typically limited to sub-nanometer length and picosecond time scales. However, long-time reactive simulations are critical for several scientific problems such as catalysis, battery interfaces, biological simulations involving water, and emerging areas like surface oxidation and chemical vapor deposition (CVD) growth. Progress on these fronts is limited because long continuous time simulations of large-scale systems are very difficult, if not impossible, to perform using purely quantum mechanical (QM) or hybrid quantum mechanical/molecular dynamics (QM/MD) simulations. The Reactive Force Field (ReaxFF) method [20], a bond order potential that bridges quantum-scale and classical MD approaches by explicitly modeling bond activity and redistribution of charges, is in principle ideally suited for this purpose. However, the computationally complex force field formulation hinders ReaxFF's scalability to large number of processing cores. In this paper, we present algorithmic and numerical techniques to address scaling bottlenecks and enable high performance ReaxFF simulations at scale.

ReaxFF is a relatively recent model (developed in early 2000s [20]) and is similar to the classical MD model in the sense that it models atomic nuclei together with their electrons as a point particle. Unlike classical MD models, ReaxFF mimics bond formation and breakage observed in QM methods by replacing the static harmonic bond models with the bond order concept, which is a quantity indicating bond strength between a pair of atoms based on the types of the atoms and the distance between them. Consequently, ReaxFF can overcome many of the limitations inherent to conventional MD. While the bond order concept dates back to 1980s and has been exploited in other force fields before (such as COMB [17] and AIREBO [18]), the distinguishing aspect of ReaxFF is the flexibility and transferability of its force field that allows ReaxFF to be applied to diverse systems of interest [6, 14, 16, 19].

ReaxFF is currently supported by major open source (PuReMD [9], LAMMPS [3], RXMD [11]) and commercial (ADF, Material Studio) software with an estimated userbase of over 1,000 groups. In this paper, we focus on the PuReMD software, as PuReMD and its LAMMPS integrations, *i.e.*, the User-ReaxC and User-ReaxC/OMP packages [3], represent the most widely used implementations of ReaxFF. PuReMD uses novel algorithms and data structures to achieve high performance while retaining a small memory footprint. An optimized neighbor generation scheme, elimination of the bond order derivatives list in bonded interactions, lookup tables to accelerate non-bonded interaction computations, and efficient iterative solvers for charge distribution are the major algorithmic innovations in PuReMD [1, 2]. PuReMD has been shown to outperform the LAMMPS/Reax package by 3-5× on various systems while using only a fraction of the memory space [1]. However, solution of large sparse linear systems required for distribution of partial charges and computations related to dynamic bonding constitute significant bottlenecks against scalability of PuReMD. More precisely, the sparse linear solves and bond order computations at halo (ghost) regions may start accounting for a significant portion of the execution time in large runs due to communication overheads and redundant computations.

In this paper, we discuss the use of a communication-hiding conjugate gradient solver to prevent the onset of collective communication overheads at scale (Section 3.1), present a preconditioning technique that significantly accelerates the sparse linear solves in charge distribution (Section 3.2), and describe a novel technique to avoid redundant bond order computations at halo regions (Section 3.3). We evaluate the performance impact of these techniques through extensive tests and demonstrate that they significantly improve the execution time and scalability of large ReaxFF simulations (Section 4). While the techniques presented are discussed in the context of the ReaxFF model, they can directly be used in other bond order potentials. The accelerated charge model solvers can also be useful for classical force fields with polarizable charge models.

2 BACKGROUND

The ReaxFF approach allows reactive phenomena to be modeled with atomistic resolution in a molecular dynamics framework. Consequently, ReaxFF can overcome many of the limitations inherent to conventional molecular simulation methods, while retaining, to a great extent, the desired scalability. Fig. 1 depicts the various ReaxFF interactions and summarizes the work flow of a simulation which includes computation of various atomic interaction functions (bonds, lone pair, over-/under-coordination, valance angles, torsions, van der Waals and Coulomb) and summing up various contributions to obtain the net force on each atom for a given time step. In terms of its implementation in PuReMD, major components of ReaxFF can be summed up as *neighbor generation*, *initialization of interaction lists*, *bonded interactions*, *charge distribution* and *non-bonded interactions*. Before presenting our performance improvement techniques, we summarize each of these major components and the general parallelization scheme in PuReMD.

Parallelization. PuReMD uses the well-known domain decomposition technique where the input domain is divided into equal-sized

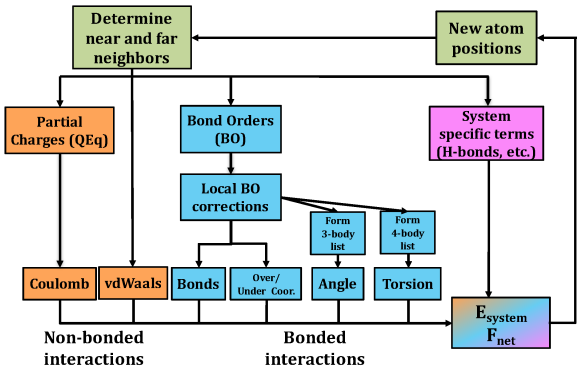


Figure 1: ReaxFF's computational workflow includes bonded, non-bonded, and system-specific interactions, each with different physical formulations and cut-offs.

subdomains according to a prescribed 3D Cartesian configuration of processes. Each process is then responsible for calculating the net force on all atoms in its subdomain and updating their positions based on the velocity Verlet integration scheme [7]. After position updates, atoms crossing process boundaries, if any, are transferred, and atom information at boundaries are exchanged locally (halo exchange) between processes that are neighbors in the 3D grid. Since interaction functionals in ReaxFF are very expensive computationally, interactions falling at process boundaries are computed on only one of the processes (as explained in [1]) to prevent redundant computations. This necessitates the exchange of force information at the end of a timestep for those atoms whose coordinates were communicated at the start of the timestep.

Neighbor Generation. In PuReMD, neighbor lists are generated using the linked cell algorithm [7] which bins atoms into small cells based on their coordinates and generates the neighbors for each atom by only scanning atoms in cells within applicable interaction cutoffs (typically 10-12Å). The neighbor list is stored by default as a half list, *i.e.*, for neighboring atoms i and j, only a single record is kept. A compact adjacency list format (similar to the compressed row format in sparse matrices) is used for storing the neighbor list. Neighbor generation is accelerated using the Verlet list method, which adds a buffer region on top of the largest interaction cutoff and updates the neighbors of each atom by scanning the Verlet list instead of the neighboring cells. This way, Verlet lists need to be formed from scratch (which we call *reneighboring* and is performed using the neighboring cell information) only occasionally. Reneighborings in ReaxFF can be delayed for a few hundred timesteps (typically 250 to 500) as each timestep in ReaxFF is only fractions of a femtosecond, *i.e.*, an order of magnitude smaller than the typical timestep length in classical MD.

Initialization of Interaction Lists. While neighbor information is needed only for non-bonded interactions (with the cutoff radii r_{nonb} typically being 10-12Å) in most classical MD models, in ReaxFF neighbor information is needed additionally for formation of the Hamiltonian matrix for the dynamic charge model (which uses a distance cutoff identical to non-bonded Coulomb interactions), for bonded interactions (with cutoff radii r_{bond} being 4-5Å), and for hydrogen bond interactions (with cutoff radii r_{hbond} being 7.5Å).

Consequently, PuReMD creates four interaction lists by scanning through the Verlet list, updating the atomic pair distances found therein and checking them against various cutoffs involved. As expected, the process of creating the interaction lists is memory-bound in terms of performance because calculations associated with creation of these lists are trivial. However, the bonds list is an exception here as it requires expensive bond order calculations between pairs of atoms that are within the prescribed r_{bond} cutoff. The memory-bound nature of this kernel and expensive bond order calculations make this routine (which we denote shortly by *init*) one of the expensive parts in PuReMD.

Bonded Interactions. Since bonds are dynamic in bond order formalism, 3-body and 4-body interactions (which involve three and four atoms, respectively, as their names indicate) also need to be discovered on-the-fly. Accurately modeling chemical reactions and avoiding discontinuities on the potential energy surface in the presence of dynamic bonds requires almost all bonded interactions (such as bond energy, 3-body valence angle energy, and 4-body torsion energy) to have significantly more complex mathematical formulations than those found in classical MD models [16, 19]. In addition, in a reactive environment, atoms often do not achieve their optimal coordination numbers; to compensate for this, ReaxFF requires additional modeling abstractions such as lone pair, over/under-coordination, and 3-body and 4-body conjugation potentials, which introduce significant computational cost to evaluation of bonded interactions. Consequently, bonded interaction calculation costs which are insignificant in classical MD models constitute a significant part of the total execution time in ReaxFF.

Charge Distribution. An important requirement for correctly modeling reactions is the charge distribution procedure, which tries to approximate the partial charges on atoms using suitable models such as the Electronegativity Equilibration Method (EEM) [12] and the Charge Equilibration Method (QEq), [15]. While the chemical intuition behind these two methods is quite different, they produce identical-looking charge distributions in practice. Since QEq produces a symmetric positive definite Hamiltonian which is easier to solve using distributed memory solvers, PuReMD uses the QEq method. In QEq, charges are determined by minimizing the electrostatic energy E_{ele}. Let $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$ denote the atomic positions in a system with n atoms, where $\mathbf{r}_i \in \mathbb{R}^3$. Atomic charges $\mathbf{q} = (q_1, q_2, \ldots, q_n)$, $q_i \in \mathbb{R}$, are thus defined by solving the following optimization problem:

$$\begin{aligned} \underset{\mathbf{q}}{\text{argmin}} \quad & E_{\text{ele}}(\mathbf{q}) = \sum_i \chi_i q_i + \frac{1}{2} \sum_{i,j} H_{ij} q_i q_j \\ \text{subject to} \quad & q_{\text{net}} = \sum_i q_i \\ \text{where} \quad & H_{ij} = \delta_{ij} \cdot J_i + (1 - \delta_{ij}) \cdot F_{ij} \\ & F_{ij} = \begin{cases} \frac{1}{\sqrt[3]{r_{ij}^3 + \gamma_{ij}^{-3}}}, & r_{ij} \le r_{\text{nonb}} \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

In Eq. (1), χ_i and J_i denote the atomic electronegativity and idempotential; δ_{ij} denotes the Kronecker delta operator; $r_{ij} = \left\| \mathbf{r}_j - \mathbf{r}_i \right\|_2$ signifies the distance between the atomic pair i and j; and $\gamma_{ij} = \sqrt{\gamma_i \cdot \gamma_j}$ denotes a pairwise tuned parameter for elements i and j for avoiding unbounded electrostatic energy at short distances. Applying the method of Lagrange multipliers to Eq. (1), the sets of linear equations below are obtained [2, 13]:

$$\begin{aligned} \sum_{i=1}^{n} H_{ki} s_i &= -\chi_k, \quad k = 1, \ldots, n \\ \sum_{i=1}^{n} H_{ki} t_i &= -1, \quad k = 1, \ldots, n. \end{aligned} \tag{2}$$

The dimension of the linear systems in Eq. (2) is equal to the number of atoms in the simulation (and can be on the order of several millions to billions). Due to the computational expense of including long-range interactions during charge distribution, QEq uses a truncated electrostatic kernel and therefore $\mathbf{H}$ is a sparse matrix. Consequently, these systems must be solved approximately using iterative methods [2]. Using solutions to Eq. (2), partial atomic charges q_i can be determined:

$$q_i = s_i - \frac{\sum_{j=1}^{n} s_j}{\sum_{j=1}^{n} t_j} \cdot t_i.$$

Nonbonded Interactions. Nonbonded interactions include van der Waals and Coulomb forces, which have functional forms of the Morse and electrostatic potentials, respectively. As these interactions are pairwise interactions, they can be calculated directly by going over the Verlet list. When advantageous performance-wise, non-bonded interactions can be approximated with very high accuracy using cubic spline interpolations over lookup tables.

3 METHODS

Charge distribution is essentially a precursor to Coulomb interactions. However, the iterative method used therein is significantly more expensive than the Coulomb interaction itself. Even worse, this iterative method requires a large number of communication operations (both local & global). We note that with the exception of position updates and force exchanges needed as part of parallelization, among all kernels described above, only the charge distribution solver requires communications. Again, among all kernels, only initialization of bonded interactions list incurs redundant calculations which may hamper scalability in large simulations. In this section, we present algorithms and numerical techniques to alleviate these inefficiencies.

3.1 Reduction of Global Communication Overheads in QEq Solvers

Baseline Preconditioned CG. Previous versions of PuReMD used the conjugate gradient (CG) algorithm for solving for the charges in the QEq procedure [1]. CG is a member of the Krylov subspace methods; these methods seek a solution x to the sparse system $Ax = b$, with A being symmetric positive definite, which falls within the Krylov subspace $\kappa(A, b) = \{b, Ab, A^2b, \ldots\}$. Each iteration of CG increases the dimension of the subspace, and the procedure terminates when an acceptable solution is found. In the case of CG, storing all the vectors which define κ is not necessary due to orthogonality and A-orthogonality of the conjugate vectors. Computing additional conjugate vectors with the prescribed A-orthogonality obviously requires sparse matrix vector multiplications (SpMVs), which in

the parallel context of PuReMD entails two local communications over the 3D process torus – a forward communication to distribute the input vectors and a backward communication to accumulate partial results into the output vector (note that in PuReMD the Hamiltonian is stored as a half matrix to leverage symmetries). Also, the orthonormalizations require calculation of inner products and vector norms. In a parallel setting, these operations entail two *all-reduce* operations which are quite expensive global communications in large scale runs.

In PuReMD, the standard CG solver is improved by: i) producing good initial guesses, ii) solving both systems in Eq. (2) through simultaneous iteration, and iii) utilizing a Jacobi (diagonal) preconditioner. The initial guesses in PuReMD are based on the observation that atomic positions (and hence charges) change only slightly from timestep to timestep; as such PuReMD uses cubic and quadratic extrapolations to solutions of Eq. (2) from previous timesteps. The Jacobi preconditioning idea stems from the fact that the Hamiltonian $\mathbf{H}$ carries a heavy diagonal. Both techniques are inexpensive, yet quite effective; together they substantially improve the convergence rate of CG for QEq, but as we demonstrate in Section 4, this basic solver still requires tens of iterations per timestep, hampering the overall scalability.

Preconditioned Pipelined CG. The scalability of Krylov subspace methods like CG is hampered by global communications during the calculation of inner products and vector norms. To combat this scalabilty issue, the pipelined CG (PIPECG) algorithm [8] aims to achieve lower communication latency by reducing the number of these communications to only one non-blocking global reduction per iteration. PIPECG essentially rearranges the algebraic formulation of CG in order to achieve this goal of one communication per step. As a trade-off for this, PIPECG ends up performing more computation (dense vector-dense vectors operations) per step and also exhibits slightly worse convergence properties than CG (i.e., stagnation at tolerances beginning at around 10^{-11}). Algorithm 1 highlights the advantages and disadvantages of PIPECG by showing that the single global communication (lines 20 and 21) can be overlapped with the preconditioner application and SpMV on subsequent lines, effectively hiding some of the communication latency – specifically, an `MPI_IAllreduce` call can be started at these lines and waited on until the completion of the SpMV. Additionally, the increased number of vector operations are shown in lines 17 through 19. Because of these advantages, we implement PIPECG and analyze its impact on performance in the following section.

Algorithm 1 Preconditioned Pipelined Conjugate Gradient

1: **function** PIPECG(H, x_0, b, M, τ, m_{iters})
2: $u \leftarrow Hx_0$ ▷ SpMV
3: $r \leftarrow b - u$
4: $u \leftarrow Mr$ ▷ Apply Prec.
5: $w \leftarrow Hu$ ▷ SpMV
6: $\delta \leftarrow w^T u$, $\gamma_{\text{new}} \leftarrow r^T u$
7: $\kappa \leftarrow \sqrt{u^T u}$, $b_{\text{norm}} \leftarrow \sqrt{b^T b}$ ▷ 1 Global Redux
8: $m \leftarrow Mw$ ▷ Apply Prec.
9: $n \leftarrow Hm$ ▷ SpMV
10: $x \leftarrow x_0$, $i \leftarrow 0$
11: **while** $\frac{\kappa}{b_{\text{norm}}} \geq \tau$ And $i \leq m_{\text{iters}}$ **do**
12: **if** $i > 0$ **then**
13: $\beta \leftarrow \frac{\gamma_{\text{new}}}{\gamma_{\text{old}}}$, $\alpha \leftarrow \frac{\gamma_{\text{new}}}{\delta - \beta/\alpha \cdot \gamma_{\text{new}}}$
14: **else**
15: $\beta \leftarrow 0$, $\alpha \leftarrow \frac{\gamma_{\text{new}}}{\delta}$
16: **end if**
17: $z \leftarrow n + \beta z$, $q \leftarrow m + \beta q$, $p \leftarrow u + \beta p$
18: $d \leftarrow w + \beta d$, $x \leftarrow x + \alpha p$, $u \leftarrow u - \alpha q$
19: $w \leftarrow w - \alpha z$, $r \leftarrow r - \alpha d$
20: $\gamma_{\text{old}} \leftarrow \gamma_{\text{new}}$, $\delta \leftarrow w^T u$
21: $\gamma_{\text{new}} \leftarrow r^T u$, $\kappa \leftarrow \sqrt{u^T u}$ ▷ 1 Global Redux
22: $m \leftarrow Mw$ ▷ Apply Prec.
23: $n \leftarrow Hm$ ▷ SpMV
24: $i \leftarrow i + 1$
25: **end while**
26: **return** x
27: **end function**

3.2 Acceleration of QEq through preconditioning

Despite reducing global communication overheads, the PIPECG solver for QEq that is described above still has excessive computation and communication costs compared to the rest of the operations that must be performed in a simulation step. Therefore, to accelerate the convergence of the PIPECG algorithm, we present a novel distributed memory preconditioning scheme based on the sparse approximate inverse (SAI) technique.

3.2.1 SAI Preconditioning. Given a linear system $\mathbf{Hx} = \mathbf{b}$, sparse approximate inverse (SAI) preconditioning aims to find a matrix $\mathbf{M}$ that serves as a good approximation to $\mathbf{H}^{-1}$ by selectively computing the entries of $\mathbf{H}^{-1}$. Using $\mathbf{M}$ as a preconditioner (the transformed system is $\mathbf{MHx} = \mathbf{Mb}$ in case of left preconditioning and $\mathbf{HMu} = \mathbf{b}$ where $\mathbf{x} = \mathbf{Mu}$ in case of right preconditioning), ideally the preconditioned system is expected to converge faster than the original linear system. To satisfy that, the cost of computing and applying the preconditioner should be marginal as construction of $\mathbf{M}$ is non-trivial and SAI preconditioning introduces one extra SpMV per solver iteration. Considering that the cost of SAI preconditioner construction and application are directly proportional to the number of non-zeros in $\mathbf{M}$, $\mathbf{M}$ should be chosen to be even sparser than the linear system itself $\mathbf{H}$ (note that $\mathbf{H}^{-1}$ is actually a dense matrix).

For the QEq problem, we choose the Frobenius norm minimization variant of left SAI preconditioning [4], which tries to find an $\mathbf{M}$ such that $||\mathbf{I} - \mathbf{MH}||_F$ is minimized where $||.||_F$ denotes the Frobenius norm of a matrix. In PuReMD, $\mathbf{H}$ is stored in the compressed sparse row (CSR) format, which actually makes constructing $\mathbf{M}$ as a right preconditioner more straightforward and affordable. Note that since $\mathbf{H}$ is symmetric, the left SAI preconditioner can then easily be inferred because left and right preconditioners of SAI are transposes of each other [5]. Let $\mathbf{e}_j$ be the j-th column of the $n \times n$ identity matrix and let $\mathbf{m}_j$ be the j-th column of $\mathbf{M}$. Then we have

$$\min_{\mathbf{M} \in \mathbb{R}^{n \times n}} ||\mathbf{I} - \mathbf{HM}||_F^2 \quad = \sum_{j=1}^{n} \min_{\mathbf{m_j} \in \mathbb{R}^{n \times 1}} \left\|\mathbf{e}_j - \mathbf{Hm}_j\right\|_2^2 .$$

Finding $\mathbf{m}_j$ for each $\left\|\mathbf{e_j} - \mathbf{Hm_j}\right\|_2^2$ corresponds to solving a set of independent least squares problems, and as such $\mathbf{M}$ can be constructed column by column through a series of *QR* decompositions.

These decompositions are performed using optimized LAPACK libraries (e.g., the Intel Math Kernel Library). Note that solving each least squares problem can be quite expensive considering the dimensions of **H**. Leveraging the fact that **M** will be much sparser than **H**, we reduce the computational costs of the least squares problems significantly by building dense vectors $\hat{\mathbf{m}}_j$ and $\hat{\mathbf{e}}_j$, and dense matrix $\hat{\mathbf{H}}_j$. This densification process essentially amounts to eliminating rows and columns that are entirely composed of zeros, as those rows and columns do not contribute to $\mathbf{Hm_j}$ (see Sect. 2.1 of [5] for further details).

While the densification step significantly reduces the computation cost of **M**, it alone is not sufficient to make our SAI preconditioner useful in practice. *The key insight that makes SAI useful for ReaxFF in practice is the same as the one we use for solver initial guesses: because an atomic system evolves slowly, the preconditioner* **M** *computed at a certain timestep can be reused effectively for several subsequent steps.* In fact, as we demonstrate in Section 4, the SAI preconditioner can be useful for hundreds of steps. As such in the accelerated QEq solver, we reconstruct **M** only occasionally and amortize the SAI preconditioner computation costs over several steps.

Determining the sparsity pattern of **M**, however, remains a challenge. The inverse of a sparse matrix is generally a dense matrix; for the inverses of our QEq matrices ($\mathbf{H}^{-1}$), we have observed that the magnitude of many entries is small. While it is desirable to select a sparsity pattern that captures the positions of the numerically large entries of $\mathbf{H}^{-1}$, these positions cannot be known *a priori*. Several works have suggested approaches for choosing a sparsity pattern [10], but these are neither sparser than **H** nor practical to compute in our case. During experiments with several molecular systems, we observed a strong correlation between the positions of the numerically large entries of **H** and $\mathbf{H}^{-1}$. Therefore, we hypothesized that positions of numerically large entries in **H** are a good candidate for the sparsity pattern of **M**. Subsequent experiments confirmed that including only τ percent of the numerically large entries of **H** indeed yields promising results. We remark, however, that finding the ideal τ value is non-trivial. On one hand, larger τ values (15%-20%) substantially improve the CG convergence but computation and application of **M** becomes expensive. On the other hand, smaller τ values make computation and application of **M** more affordable at the expense of smaller improvements in solver convergence. The longevity of **M** (discussed in the previous paragraph), the number of processes used, and the parallel efficiency are certainly other important considerations.

We observe that in general τ being 10% to 15% (depending on the molecular system) leads to decent improvements in total QEq solve time. While we empirically determine the ideal value of τ for experiments presented in this paper, note that MD simulations typically last for millions to billions of steps. *As such, an auto-tuning mechanism that empirically determines the ideal τ value and preconditioner reconstruction frequency on-the-fly for a given atomic system and architecture, while performing the MD simulation itself, is certainly feasible and is planned as future work.*

3.2.2 Parallelization of the SAI Preconditioner. After describing how we "engineer" a custom SAI preconditioner that is practical for dynamic charge models, we next detail its parallelization. While this topic is arguably well documented in the literature, our contribution here is the development of an SAI implementation that leverages problem-specific characteristics of ReaxFF for high efficiency and scalability, so that the resulting solver can perform significantly better than PuReMD's existing QEq solver that already delivers decent results with its simple Jacobi preconditioning scheme.

Our discussion focuses on parallel construction of the SAI preconditioner **M** as this step is highly non-trivial; application of **M**, which needs to be performed at each iteration of PIPECG, is trivial as it simply requires a parallel SpMV. Construction of **M** includes two main subtasks: i) pattern selection, and ii) setup and solution of the least squares problems. As both subtasks are intimately tied to the structure of the Hamiltonian **H**, we begin with its description.

Structure of the Hamiltonian. In the Hamiltonian **H**, a matrix entry (i, j) represents the charge affinity relation between atoms i and j. While the original charge solver in PuReMD adopts a half matrix for reducing storage and SpMV computation costs, in the SAI preconditioned solver we store the full **H** matrix in distributed memory (hence redundantly storing the symmetric entries twice), as this simplifies the setup of the least squares problems (subtask 2). With the full storage scheme, the **H** matrix is effectively 1D block partitioned because each process knows all neighbors (*i.e.*, all non-zeros in a row) of its local atoms (*i.e.*, all rows it owns).

Locally each process stores its submatrix in a structure that facilitates the construction of **M**, as well as parallel SpMVs (needed in CG/PIPECG). As depicted in Fig. 2, lower indices are reserved for local atoms. In PuReMD, local communications between neighboring processes follows a three-stage communication scheme that respects the 3D torus topology: first, atom information is exchanged in $-x$ and $+x$ directions; then in $-y$ and $+y$ directions, and finally in $-z$ and $+z$ directions [1]. Atoms received after each stage are indexed contiguously in the atom list as well as the local Hamiltonian, and messages to be transmitted in the subsequent step are augmented with the received atom information as necessary. This communication scheme yields the indexing and organization shown in Fig. 2.

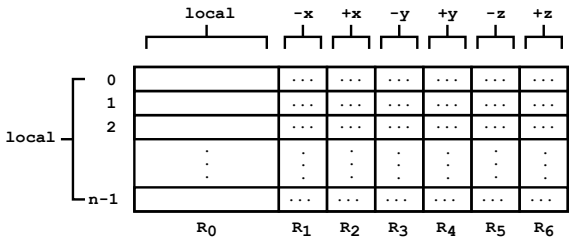


Figure 2: Structure of the local charge matrix where region R_0 corresponds to the interactions between local atoms, R_1 corresponds to the interactions between local atoms and the atoms imported from $-x$ direction, etc.

SAI Preconditioner Setup. Given p processes and the filtering parameter τ, for each process local Hamiltonians $\mathbf{H}^{\mathbf{s}}_{\mathbf{loc}}$, with $1 \leq s \leq p$, the top τ percent numerically largest non-zeros of **H** are selected for the sparsity pattern of **M**. This problem can equivalently be stated as finding the k-th largest non-zero, namely α, in the union of all local Hamiltonians such that when entries less than α are

filtered out, only τ percentile of the non-zeros remain. However, trying to find the exact α does not lend itself to an efficient parallel algorithm, but trying to be exact is not required as τ is an empirically determined threshold. As such, we approximate τ using a sampling-based technique as shown in Algorithm 2 (see line 2). Sampled numbers from each process are collected at the root (line 5), and these numbers are processed using a Quickselect algorithm to efficiently find α (line 7). After broadcasting α, each process can obtain their local sparsity patterns $\mathbf{H^s_{sp}}$ by filtering out entries in their local $\mathbf{H^s_{loc}}$ matrices (line 10).

Algorithm 2 SAI Pattern Setup

```
1:  function SetupSAI(H, p, τ)
2:      P_local ← Sample(H, samplerate)
3:      s ← Reduce(length of P_local)
4:      if rank = root then
5:          P_global ← Gather(P_local)
6:          k ← ⌊ s·τ / 100 ⌋
7:          α ← Quickselect(P_global, k)
8:      end if
9:      Bcast(α)                          ▷ broadcast the threshold value
10:     H_sp ← FilterMatrix(H, α)
11:     return H_sp
12: end function
```

SAI Preconditioner Computation. As described in Section 3.2.1, constructing the SAI preconditioner entails solving n least squares problems where n is the total number of atoms. These problems are independent and thus can be solved in parallel by performing QR decompositions. However, the precursor to the QR decompositions is building the least squares problems which we explain through the example in Fig. 3. After determining α, the least squares problem for atom i on process s is built based on the sparsity pattern of the i-th column of $\mathbf{H^s_{sp}}$. In this example, we form the least squares problem for atom 0, and suppose that the non-zeros in column 0 of $\mathbf{H^s_{sp}}$ are in rows 0 and 3 (non-zeros in rows 4 and 7 are below α and are dropped). Consequently, columns 0 and 3 of $\mathbf{H}$ are included, but the real challenge lies in finding the subset of rows of H that are to be included in the densified least squares problem $\hat{H}_i$. Those rows are the union of j's such that at least one of the selected columns of $\hat{H}_i$ has a non-zero element at index j. Referring to the example in Fig. 3, the rows incorporated into the densified matrix will be $\{0, 3, 4, 7\} \cup \{3, 5\} = \{0, 3, 4, 5, 7\}$, which are the rows with non-zero entries found in columns 0 and 3, respectively. This process is straight-forward if all atoms were to be local, but when there is a non-local atom j in the sparsity pattern of the i-th column of $\mathbf{H^s_{sp}}$, all neighbors of j are needed; this information must be imported from another process. Fortunately, when local Hamiltonians are stored as full matrices, all neighbors of atom j can be obtained from the process that owns atom j. We describe the SAI preconditioner setup operation in Algorithm 3. To efficiently import all the neighbors of a non-local atom, a modified version of three-stage messaging is employed in the CompSAI function. All local atoms save the number of their neighbors, which is equal to the number of non-zero entries in the corresponding row of the local matrix, to a list (line 2-3). Then, every process distributes their list using staged communication (line 8) so that each atom can get

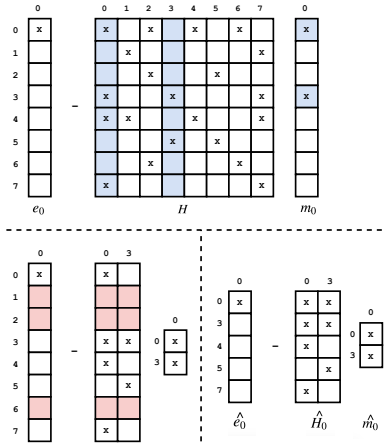


Figure 3: The global view of the process of constructing $\hat{e}_0$, $\hat{H}_0$, and $\hat{m}_0$ for an example QEq matrix $H \in \mathbb{R}^{8\times8}$ and m_0 through column selection (blue) and row eliminations (red). In the above matrices, empty spaces and x's denote zero and non-zero entries, respectively.

to learn how many non-zero entries have to be imported for each of its non-local neighbors (line 10-14). Next, the processes allocate space for those non-zero entries (line 15) and pack the non-zero entries for the local atoms that are needed by other processors (line 16-20). Then, again using the same communication scheme, each processor sends its packed entries (line 21). Finally, processors start constructing dense matrices and solving QR decompositions for each of their local atoms (line 22-25).

3.3 Optimization at the Ghost Regions

In ReaxFF, the precursor to bonded interactions is the calculation of the uncorrected bond orders (BOp), which is performed within the r_{bond} radii of each atom during initialization of interaction lists. Once the bond order list is initialized, all bonded interaction functions are calculated without any redundancies. For instance, a bond at the boundary between two processes is calculated by the owner of the atom with the smaller *global id*, or a 3-body interaction shared between 3 processes is calculated by the owner of the middle atom. In PuReMD, a process initializes all bonds that fall within its boundaries including the ghost regions to be prepared to handle all kinds of bonded interactions that it may end up being responsible for. Note that in the strong scaling regime, the redundant computations associated with doing so can be significant. For instance, when the dimensions of the subdomain of a process is equal to r_{nonb}, the three dimensional import region (which must have a thickness of r_{nonb} at least in each direction) would have a volume of roughly 26 times the volume of the process's local subdomain. However, inspecting how bonded interactions are shared between processes in PuReMD reveals that one must only extend up to 3 hops into the ghost from any local atom (as expected, this happens for 4-body bonded interactions) where a hop is defined to be a bond order with strength above a certain threshold. This suggests that there can be a significant number of redundant bond order calculations in the ghost region.

Algorithm 3 SAI Computation

```
1: function CompSAI(H, H_sp, n, N)
2:     for i ← 0 to n − 1 do
3:         row_nnz(i) ← H(i).end − H(i).start
4:     end for
5:     for i ← n to N − 1 do
6:         row_nnz(i) ← 0
7:     end for
8:     Dist(row_nnz)
9:     nnz_recv ← 0
10:    for i ← n to N − 1 do
11:        if row_nnz(i) ≠ 0 then   ▷ non-local atom i is needed by a processor
12:            nnz_recv ← nnz_recv + row_nnz(i)
13:        end if
14:    end for
15:    AllocateSpace(rows_recv, nnz_recv)
16:    for i ← 0 to n − 1 do
17:        if atom i is needed by a neighbor then
18:            rows_send.append(H(i)   ▷ non-zero entries in row_i is packed
19:        end if
20:    end for
21:    Dist(rows_send)
22:    for i ← 0 to n − 1 do
23:        ê_i, Ĥ_i, m̂_i ← Build_Dense_Matrix(H, H_sp, rows_recv, i)
24:        H_sai(i) ← QR(ê_i, Ĥ_i, m̂_i)
25:    end for
26:    return H_sai
27: end function
```

To avoid redundant BOp calculations in an effort to improve the strong scaling performance, i) we adopted a BFS-style branching with multiple sources, and ii) we implemented a scheme to automatically tighten the r_{bond} cutoff for a given input system. In our first optimization, *i.e.*, the BFS-style branching scheme, all local atoms are initially added to a queue with a hop distance of 0. Then, an atom is popped from the queue at each BFS step and its neighbors within a hop distance are inserted into the queue. The method stops when all the atoms with less than 4 hop distance are inserted into the queue. Notice that this approach can be used only when full neighbor list format is employed; otherwise, we would have to calculate the hop distances of atoms in the underlying graph of a digraph, which is identical to turning a half neighbor list into a full one. Moreover, we take advantage of symmetry to optimize the computations even further as full neighbor list is the only viable option. To realize this, we ensured that the interaction between atom i and atom j is computed by the atom that is popped from the queue first.

Our second optimization, *i.e.*, automatic tightening of r_{bond}, relies on the observation that, given two atom types t_i and t_j, BOp is a monotonically decreasing function of the distance between atoms. However, r_{bond} is usually conservatively given as 4Å (or even 5Å) as part of simulation parameters. Instead of relying on this parameter, we scan all possible atom type pairs at several distances to precisely determine the distance after which the BOp value for all type pairs present in the given system are guaranteed to fall below bond order acceptance threshold.

4 NUMERICAL RESULTS

4.1 Benchmarking Systems and Hardware

Results from numerical experiments presented in the following subsections were obtained on Cori at the National Energy Research Scientific Computing Center (NERSC). Each Haswell node in this Cray XC40 system has 32 cores, on two sixteen-core Intel Xeon E5-2698v3 Haswell 2.3 GHz processors, and has 128 GB DDR4 2133 MHz ECC memory. Each core possesses a 64 KB L_1 cache (32 KB instruction, 32 KB data), a 256 KB L_2 cache, and the capability of running one or two user threads (i.e., hyperthreading). All cores on a single processor share a 40 MB L_3 cache. At the time of the experiments, Cori ran the SuSE Linux Enterprise Server version 12.3 for x86_64 architectures, linux kernel version 4.4.162-94.72-default, and glibc version 2.22-62.16.2.

The preconditioned solvers detailed in the above sections were implemented in the PuReMD ReaxFF software [1, 2]. The software was built using the Intel Compiler Collection version 18.0.1 with the `-O3 -march=native` flags and the Cray MPICH library version 7.7.3. For relevant experiments, we restricted our simulations to one process per core (no hyperthreading was utilized).

For benchmarking purposes, two molecular systems from application scenarios of reactive and polarizable force fields were selected. These systems, which were comprised of bulk water (H_2O) and amorphous silica (SiO_2), range in size from thousands to millions of atoms depending on the experiment.

To quantify the impact of optimizations presented in Section 3, we created four different versions of the PuReMD code: i) original PuReMD code as published on its website [9] (*i.e.*, CG+Jacobi [Half]), ii) PuReMD code modified to work with full neighbor lists and full Hamiltonians (*i.e.*, CG+Jacobi [Full], aims to quantify the impact of the switch to full list and matrices), iii) PuReMD with SAI preconditioning and ghost region optimizations (*i.e.*, CG+SAI(0.15) that denotes a τ value of 15%), and our final version which includes all presented optimizations (*i.e.*, PIPECG+SAI(0.15)).

4.2 Impact of Charge Solver Improvements

SAI Longevity. We start by examining for how long an SAI preconditioner can be effective, as the longevity of the preconditioner is crucial for determining how much preconditioner computation costs can be amortized, as well as for estimating the average number of solver iterations. As shown in Fig. 4, the SAI preconditioners can be effective for tens to hundreds of steps, especially for the silica system which is a solid material where atoms cannot move freely like they do in water. SAI preconditioner loses its effectiveness after some point, and the necessity of preconditioner reconstruction can be seen clearly. Overall, silica system requires fewer iterations for convergence than the water system at the same tolerances, and the gap becomes more apparent as the convergence tolerance is decreased to 10^{-10}.

Based on results shown in Fig. 4 and further empirical tests (not shown), we set the SAI reconstruction rates to 250 steps for evaluations performed in this paper. All simulations have been executed

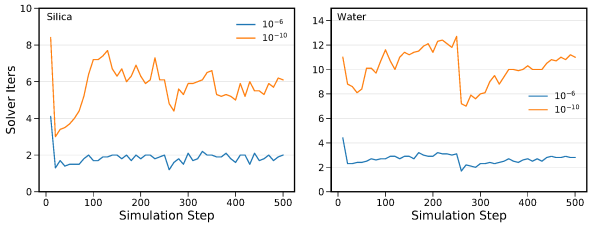


Figure 4: Longevity of the CG+SAI(0.15) preconditioned solver for the silica system with 864K atoms (left column), and bulk water system with 837K atoms (right column) using the QEq model at different solver tolerances. Preconditioners were utilized for 250 steps before being re-computed.

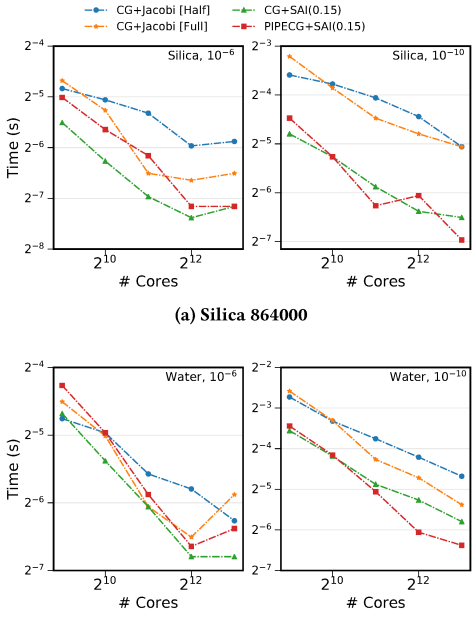
for 500 steps and averages of relevant quantities are reported, unless stated otherwise.

SAI Preconditioning Convergence Rates. In Table 1, we present the mean solver iterations for the Jacobi and SAI preconditioned QEq solvers at 10^{-6} and 10^{-10} tolerances for the 864K silica and 837K water systems using 512 processes. As expected, neither the switch to full matrix with CG, nor replacing the CG solver with PIPECG affects convergence rates. SAI preconditioning significantly improves the convergence rate of QEq compared to Jacobi preconditioning, yielding about 4 to 5 times better convergence rates for silica and water at 10^{-6} tolerance. The margin between SAI and Jacobi preconditioning gets larger as we go to 10^{-10} tolerance threshold both for silica and water systems. Other tests (not shown here) indicate that this trend continues as we move to even smaller tolerances, *e.g.*, 10^{-13}.

Table 1: Mean solver iterations for the Jacobi and SAI preconditioned QEq solvers.

Dataset	Tol.	Solver	Iters.
Silica	10^{-6}	CG+Jacobi [Half]	11.8
		CG+Jacobi [Full]	11.8
		CG+SAI(0.15)	1.9
		PIPECG+SAI(0.15)	1.9
	10^{-10}	CG+Jacobi [Half]	39.2
		CG+Jacobi [Full]	39.2
		CG+SAI(0.15)	5.8
		PIPECG+SAI(0.15)	5.8
Water	10^{-6}	CG+Jacobi [Half]	9.5
		CG+Jacobi [Full]	9.5
		CG+SAI(0.15)	2.7
		PIPECG+SAI(0.15)	2.7
	10^{-10}	CG+Jacobi [Half]	38.4
		CG+Jacobi [Full]	38.4
		CG+SAI(0.15)	10.2
		PIPECG+SAI(0.15)	10.2

Strong Scaling Tests. While convergence results in Table 1 are highly encouraging regarding the speedups that can be obtained through SAI preconditioning, they omit the execution time overheads associated with SAI in actual solves which include preconditioner



(a) Silica 864000

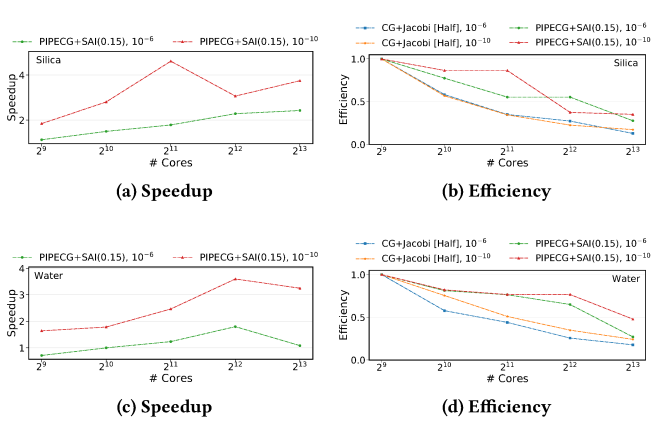
(b) Water 837120

Figure 5: Strong scaling plots for mean QEq time of four preconditioned solvers for the silica and water systems. Figures from left to right within a row show results at convergence tolerance levels of 10^{-6} and 10^{-10}.

construction and application costs. While the preconditioner construction is amortized over several steps, preconditioner application – which essentially is a parallel SpMV – must be performed at each iteration of the QEq solver and requires an extra set of local communications; therefore it can be expensive. In Fig. 5, we present the results of a strong scaling study for silica and water systems. In comparing CG+Jacobi[Half] and CG+Jacobi[Full], we observe that the switch to full neighbor lists and full charge matrices does not have a significant negative effect. In fact, at a large number of cores, we observe a positive impact from CG+Jacobi[Full], because it only needs to do a forward communication before the SpMV in CG iterations. In comparison, CG+Jacobi[Half] needs to do a forward communication before SpMV and backward communication after SpMV. In any case, neither CG+Jacobi[Half] nor CG+Jacobi[Full] exhibit good scaling beyond 2048 cores.

By switching to SAI preconditioning, CG+SAI(0.15) is able to achieve significant speedups over the Jacobi variants by virtue of reduced CG iterations. As expected, the fully optimized PIPECG+SAI(0.15) version exhibits the best performance in large core counts, while it attains similar (or sometimes slightly worse) performance to its CG variants at smaller runs. As shown in Fig. 6, we observe 0.7x to 2.4x speedup at 10^{-6} tolerances, and 1.6x to 4.6x speedup at the 10^{-10} tolerances from PIPECG+SAI(0.15) over the original QEq solver in PuReMD. Again looking at Fig. 6, it can be said that PIPECG+SAI(0.15) almost always yields better strong scaling efficiencies than the original QEq solver.

Note that the simulation volume (hence computational load) assigned to each process decreases linearly with the increase in



(a) Speedup (b) Efficiency

(c) Speedup (d) Efficiency

Figure 6: Speedups and parallel efficiencies for the strong scaling study with silica (top row) and water (bottom row). Speedups are relative to the CG+Jacobi [Half] baseline at the same number of cores.

the core count. However, the volume of the ghost region (hence local communication) decreases at a slower pace. Additionally, the overhead due to global communication operations of CG (*i.e.*, all-reduces) grows with increasing core counts. Consequently, the ratio of communication overheads to computational load on each process is maximized at high core counts, and unsurprisingly this is where the impact of SAI and PIPECG are observed most significantly.

Weak Scaling. To better analyze the impact of core count over execution time and scaling, we also conducted weak scaling experiments. As indicated earlier, in the current PuReMD implementation, the dimensions of the simulation box that can be assigned to each process must be equal to or greater than the neighbor generation cutoff (which is equal to r_{nonb} cutoff plus the Verlet buffer size). Therefore for our weak scaling experiments, we choose bulk water and silica systems with cubic shapes that roughly have the minimum possible box dimensions per process, *i.e.*, ≈13Å, and scale it up to 1,728 cores. As given in Fig. 7, PIPECG+SAI(0.15) yields speedups of between 0.9x to 2.5x for a 10^{-6} tolerance, and between 1.4x to 5.1x for a 10^{-10} tolerance over the original QEq solver. As a result of our highly optimized SAI preconditioner, SAI related overheads are minimal, allowing the reduction in iteration counts to translate to QEq speedups. Another source of performance gain with PIPECG+SAI(0.15) is the overlapping of global communications which constitutes the most significant part of the original QEq solver on 1,728 cores.

4.3 Impact of Ghost Region Optimizations

In Table 2, we quantify the impact of the ghost region optimizations. Since the implementation of the original interaction list initializations in PuReMD does not allow us to time the exact duration spent in bond order initializations, we rather compare the number of BOp evaluations in both schemes. As shown in this table, the presented optimizations significantly reduce the number of BOp evaluations by a factor of about 5 for water and about 2 for silica. This is expected as silica has longer maximum bond distance (3.5Å) and therefore 3 hops into the ghost region already covers a

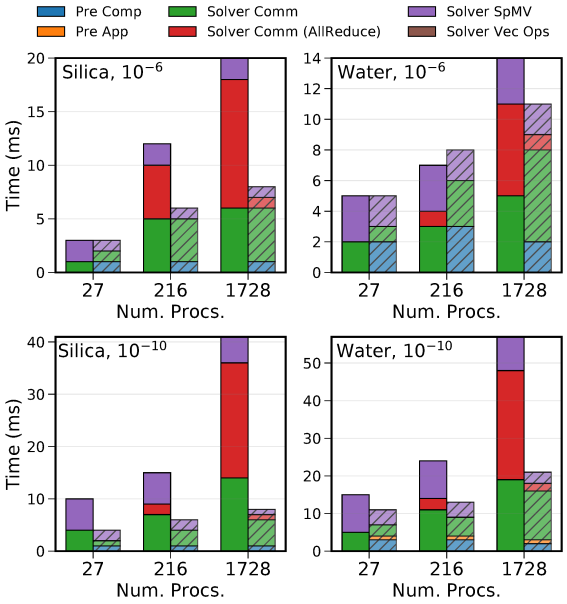


Figure 7: Weak scaling plots for QEq sub-kernels using the silica and bulk water systems at solver tolerances of 10^{-6} and 10^{-10}. Unhatched bars (left within a grouping) are from CG+Jacobi [Half] while hatched bars (right within a grouping) are from PIPECG+SAI(0.15).

significant volume. But, the maximum bond distance for water is shorter, and as such redundant calculations in a larger portion of ghost region can be avoided.

Table 2: Average number of bond order calculations per simulation step for silica and bulk water systems (6000 and 6540 atoms, respectively) using 27 cores.

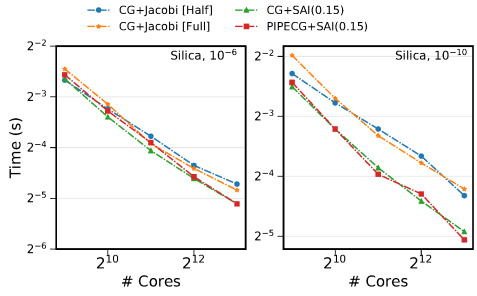
Dataset	Original Bond Init	Optimized Bond Init
Silica	2.6E5	1.1E5
Water	6.3E5	1.3E5

4.4 Overall Simulation Performance

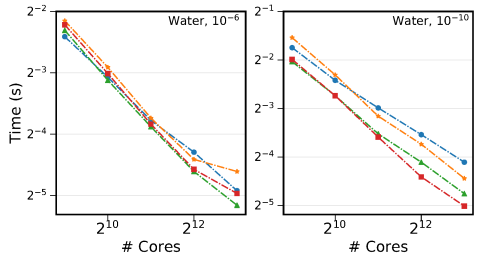
In Figs. 8 and 9, we present the strong scaling and weak scaling results for the entire PuReMD simulation, which are indeed quite similar to those presented above for QEq. Since QEq solves and initialization of interaction lists constitute a significant amount of the total running time, improvements obtained for each of these kernels carry their benefits to the overall simulation times, especially when solver tolerance is reduced to 10^{-10}. In our experiments, we observed overall speedups of 1.1x to 1.9x for strong scaling experiments, and 0.9x to 1.8x for weak scaling experiments.

5 CONCLUSIONS

Reactive MD models fill an important void in the molecular dynamics landscape, but the scalability of existing software is limited due to the onset of communication overheads and redundant calculations, specifically due to charge distribution solvers and computations related to dynamic bond order lists. In this paper, we presented



(a) Silica 864000



(b) Water 837120

Figure 8: Strong scaling plots for total simulation time of four preconditioned solvers for the silica and water systems. Figures from left to right within a row show results at convergence tolerance levels of 10^{-6} and 10^{-10}.

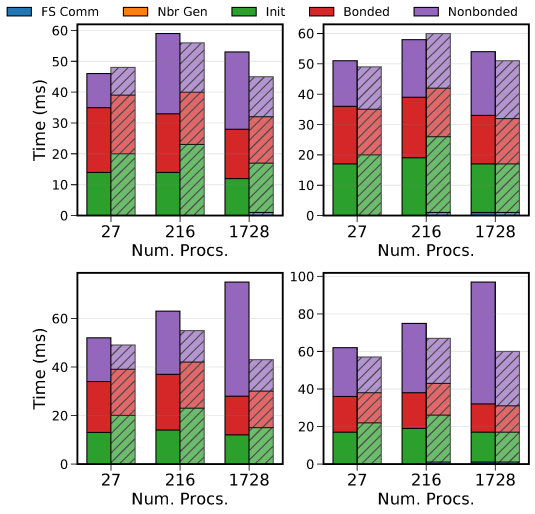


Figure 9: Weak scaling plots for main simulation subkernels using the silica (left column) and bulk water (right colum) systems at solver tolerances of 10^{-6} (top row) and 10^{-10} (bottom row). Unhatched bars (left within a grouping) are from CG+Jacobi [Half] while hatched bars (right within a grouping) are from PIPECG+SAI(0.15).

a number of novel techniques to address these bottlenecks and studied their performance impact in large-scale simulations. Our results show that the presented techniques can significantly improve the overall performance and scaling of reactive MD simulations. A number of potential performance improvement opportunities have been observed as discussed above, and will be the subject of our future work in this area. While the techniques presented in this paper are discussed in the context of the ReaxFF model, they can directly be used in other bond order potentials. The accelerated charge model solvers can also be useful for classical force fields with polarizable charge models.

ACKNOWLEDGMENTS

This research was supported through a Michigan State University Foundation Strategic Partnership Grant, NSF Grants ACI-1566049 and OAC-1807622, and the computing resources at the National Energy Research Scientific Computing Center (NERSC).

REFERENCES

[1] H. M. Aktulga et al. Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques. *Parallel Comput.*, 38(4-5):245–259, 2012.
[2] H. M. Aktulga et al. Reactive molecular dynamics: Numerical methods and algorithmic techniques. *SIAM J. Sci. Comput.*, 34(1):C1–C23, 2012.
[3] Hasan Metin Aktulga, Christopher Knight, Paul Coffman, Kurt A O'Hearn, Tzu-Ray Shan, and Wei Jiang. Optimizing the performance of reactive molecular dynamics simulations for multi-core architectures. *arXiv preprint arXiv:1706.07772*, 2017.
[4] M. Benzi and M. Tuma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics*, 30:305–340, 04 1998.
[5] M. Benzi and M. Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 19(3):968–994, 1998.
[6] Joseph C Fogarty, Hasan Metin Aktulga, Ananth Y Grama, Adri CT Van Duin, and Sagar A Pandit. A reactive molecular dynamics simulation of the silica-water interface. *The Journal of chemical physics*, 132(17):174704, 2010.
[7] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Elsevier, 2001.
[8] Pieter Ghysels and Wim Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40, 06 2013.
[9] A.Y. Grama, H. M. Aktulga, and S. B. Kylasa. PuReMD, Purdue Reactive Molecular Dynamics package, 2014. Accessed on June 8, 2016.
[10] M. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.
[11] Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashista. A scalable parallel algorithm for large-scale reactive force-field molecular dynamics simulation. *Comp Phys Comm*, 178(2):73–87, Januaury 2008.
[12] Wilfried J Mortier et al. Electronegativity-equalization method for the calculation of atomic charges in molecules. *Journal of the American Chemical Society*, 108(15):4315–4320, 1986.
[13] Aiichiro Nakano. Parallel multilevel preconditioned conjugate-gradient approach to variable-charge molecular dynamics. *Computer Physics Communications*, 104(1):59–69, 1997.
[14] Yumi Park, Hasan Metin Atkulga, Ananth Grama, and Alejandro Strachan. Strain relaxation in si/ge/si nanoscale bars from molecular dynamics simulations. *Journal of Applied Physics*, 106(3):034304, 2009.
[15] A. K. Rappe and W. A. Goddard, III. Charge equilibration for molecular dynamics simulations. *J. Phys. Chem.*, 95(8):3358–3363, 1999.
[16] Thomas P. Senftle et al. The ReaxFF reactive force-field: Development, applications and future directions. *Nature PJ Computational Materials*, 2:15011, Mar 2016.
[17] T.-R. Shan, B. D. Devine, T. W. Kemper, S. B. Sinnott, and S. R. Phillpot. Charge-optimized many-body potential for the hafnium/hafnium oxide system. *Phys. Rev. B*, 81:125328, 2010.
[18] S. J. Stuart, A. B. Tutein, and J. A. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *J. Chem. Phys.*, 112(14):6472–6486, 2000.
[19] A. C. T. van Duin et al. Reaxff: A reactive force field for hydrocarbons. *J. Phys. Chem. A*, 105:9396–9409, 2001.
[20] Adri CT Van Duin, Siddharth Dasgupta, Francois Lorant, and William A Goddard. Reaxff: a reactive force field for hydrocarbons. *The Journal of Physical Chemistry A*, 105(41):9396–9409, 2001.