SIAM J. SCI. COMPUT.
Vol. 42, No. 1, pp. C1–C22

FAST SOLVERS FOR CHARGE DISTRIBUTION MODELS ON SHARED MEMORY PLATFORMS*

KURT A. O'HEARN†, ABDULLAH ALPEREN†, AND HASAN METIN AKTULGA†

Abstract. Including atom polarizability in molecular dynamics (MD) simulations is important for high-fidelity simulations. Linear solvers for charge models that are used to dynamically determine atom polarizations constitute significant bottlenecks in terms of time-to-solution and the overall scalability of polarizable and reactive force fields. We present properly customized preconditioning techniques to accelerate the iterative solvers used for several charge models and develop their efficient shared memory parallel implementations in the open source PuReMD (Purdue Reactive Molecular Dynamics) software package. With these goals in mind, special attention has been paid to minimizing the mean combined preconditioner construction and solver time. Detailed analysis of how different preconditioning techniques affect solver convergence rate and the overall performance is presented. Incomplete LU/Cholesky and sparse approximate inverse (SAI) based schemes that produce good quality factors with a relatively low number of nonzeros have been observed to yield significant speedups over a baseline Jacobi preconditioner. These results are significant as they can enable efficient simulations of small to moderate-sized systems on multicore computers, but, more importantly, they serve as a basis for distributed memory solvers.

Key words. molecular dynamics, reactive force fields, polarizable force fields, charge distribution models, Krylov subspace solvers, preconditioning methods, incomplete LU, sparse approximate inverse

AMS subject classifications. 65F08, 65F10, 65F50

DOI. 10.1137/18M1224684

1. Introduction.

Molecular dynamics (MD) simulations are utilized across a wide range of fields, including physics, chemistry, biology, and materials science. On the one hand, the *force field*s (the set of parameterized mathematical equations describing the interactions between atoms) used in these simulations must be of high fidelity so that simulations can have scientific impact. On the other hand, the force field formulations need to be computationally inexpensive such that simulations of millions to billions of timesteps can be routinely performed, allowing scientists to examine chemical or biological phenomena that take place in the range of microseconds to milliseconds. In this regard, over the past few decades several highly successful force fields have been developed to model liquids, proteins, and other molecular materials [16, 11, 27]. The efficiency of these force fields is mainly due to the use of static bonds between atom pairs and fixed charges on each atom. While suitable for various applications, these simplifications render classical force fields unsuitable in applications where polarization effects or chemical reactions are important.

The impact of modeling charge distribution on the fidelity of MD simulations is well known [13, 29, 25, 15]. As such, to this day several charge distribution models

*Submitted to the journal's Software and High-Performance Computing section November 5, 2018; accepted for publication (in revised form) November 8, 2019; published electronically February 18, 2020. This work constitutes a significant extension of the work previously published in [23]. Specifically, new charge models (EE and ACKS2), preconditioning techniques (ILUDD, SAI), an effective dynamic refactorization scheme, and scalability enhancements (graph coloring for ILU-based method) were added. Also, a deeper performance analysis is presented.

https://doi.org/10.1137/18M1224684

Funding: This work was supported by NSF grants ACI-1566049 and OAC-1807622, and by an MSU Foundation Strategic Partnership Grant (SPG).

†Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 (ohearnku@msu.edu, alperena@msu.edu, hma@msu.edu).

have been developed for atomistic systems [17, 28, 22, 35]. While these models basically aim to find the charge distribution that minimizes the total electrostatic energy, the set of physical/chemical constraints they impose in this optimization problem is essentially what separates them. As discussed in subsection 2.1, minimization of the electrostatic energy subject to the set of constraints defined by a particular model requires the solution of large systems of linear equations. It is important to note that trying to incorporate the long-range Coulomb effects into the dynamic charge distribution models would significantly increase their computational costs, making them impractical for actual simulations. Hence, force fields utilizing dynamic charge models adopt tapering functions which limit the effect of charges only within a certain radius (which is typically 10–12 Å). As a result, linear systems arising in the formulations of dynamic charge models are sparse, and iterative solvers are used to find sufficiently accurate solutions.

Development of fast and efficient iterative solvers for dynamic charge models constitutes the main objective of this paper. Previous work on this topic has explored the use of Jacobi preconditioners and good initial guesses, both of which have indeed been shown to be highly effective [1, 2]. Nevertheless, charge distribution in actual simulations still takes up a significant amount of computation time and represents an important scalability bottleneck, as demonstrated, for instance, for the Reax Force Field (ReaxFF) [34, 33] in [1, 3]. It can be argued that dynamic charge distribution would represent an even more significant bottleneck for other force fields if it were incorporated into their formulations, because ReaxFF is known to be a complex and computationally expensive force field.

In this paper, we focus on the development of effective, yet inexpensive, preconditioners to accelerate the solvers used in dynamic charge distribution models, and focus as well on high-performance implementation of the resulting solvers in the open source PuReMD (Purdue Reactive Molecular Dynamics) software package [1, 2]. Presented techniques are shared memory parallel, but an efficient shared memory parallel solver constitutes the critical building block for extending our techniques to distributed memory systems. We demonstrate through extensive numerical tests that these carefully designed efficient preconditioned solvers and their efficient implementations can significantly accelerate solvers for dynamic charge models.

2. Dynamic charge distribution models (CMs) and linear solvers.

2.1. Charge equilibration (QEq). We define and discuss the charge distribution models and their formulations as linear systems of equations, beginning with the QEq model [28]. In this model, the distribution of charges over atoms is determined by minimizing the electrostatic energy E_{ele} given the atomic positions. Let $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n)$ signify the positions of the system of n atoms, where $\mathbf{r}_i \in \mathbb{R}^3$ for $1 \leq i \leq n$. Atomic charges $\mathbf{q} = (q_1, q_2, \ldots, q_n)$, $q_i \in \mathbb{R}$, are thus defined by solving the following problem:

$$\begin{aligned} \min_{\mathbf{q}} \quad & E_{\text{ele}}(\mathbf{q}) = \sum_i \chi_i q_i + \frac{1}{2} \sum_{i,j} H_{ij} q_i q_j \\ \text{subject to} \quad & q_{\text{net}} = \sum_i q_i. \end{aligned} \tag{2.1}$$

In (2.1), we define H_{ij} as $\delta_{ij}\eta_i + (1 - \delta_{ij}) \cdot F_{i,j}$, where δ_{ij} denotes the Kronecker delta operator; χ_i and η_i denote the atomic electronegativity and idempotential; $r_{ij} = ||\mathbf{r}_j - \mathbf{r}_i||_2$ signifies the distance between the atomic pair i and j; and $\gamma_{ij} = \sqrt{\gamma_i \cdot \gamma_j}$

denotes a pairwise shielding term tuned for element types of atoms i and j to avoid unbounded electrostatic energy at short distances. Additionally, $F_{i,j}$ is defined as

$$F_{i,j} = \begin{cases} \frac{1}{\sqrt[3]{r_{ij}^3 + \gamma_{ij}^{-3}}}, & r_{ij} \leq r_{\text{nonb}} \text{ (which is typically 10–12 Å)}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

Applying the method of Lagrange multipliers to (2.1), we obtain the following sets of linear equations below [19, 2]:

$$\begin{aligned} \sum_{i=1}^{n} H_{ki} s_i &= -\chi_k, \quad k = 1, \ldots, n \\ \sum_{i=1}^{n} H_{ki} t_i &= -1, \qquad k = 1, \ldots, n. \end{aligned} \tag{2.3}$$

Here, the values s_i and t_i can be thought of as pseudocharges used during the Lagrangian method. From these, partial atomic charges q_i can be computed as follows:

$$q_i = s_i - \frac{\sum_{j=1}^{n} s_j}{\sum_{j=1}^{n} t_j} \cdot t_i. \tag{2.4}$$

We henceforth refer to the coefficient matrix in (2.3) as $\mathbf{H}_{\text{QEq}} \in \mathcal{R}^{n \times n}$. We note that $\mathbf{H}_{\text{QEq}}$ is symmetric and has been observed to be positive definite across numerous atomic systems we have worked with.

2.2. Electronegativity equalization (EE). EE is another commonly used approach for partial charge calculation; it relies on the principle that charges should be distributed to atoms in order to satisfy constraints for both net system charge and equalized atom electronegativity [18, 17]. For the latter, we represent the electronegativity of atom i as ϵ_i, and thus the EE constraint is formalized as

$$\epsilon_1 = \epsilon_2 = \cdots = \epsilon_n = \bar{\epsilon}. \tag{2.5}$$

Using the same definition of electrostatic energy (i.e., E_{ele}) and empirical parameters from (2.1) in conjunction with the constraints in (2.5), we can represent the system of linear equations to be solved in EE in block form as the following superset of those from the QEq model:

$$\begin{bmatrix} \mathbf{H}_{\text{QEq}} & \mathbf{1}_n \\ \mathbf{1}_n^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{q} \\ \bar{\epsilon} \end{bmatrix} = \begin{bmatrix} -\chi \\ q_{\text{net}} \end{bmatrix}. \tag{2.6}$$

In (2.6), q_{net} again represents the net charge of the atomic system, while the EE charge matrix $\mathbf{H}_{\text{EE}} \in \mathcal{R}^{(n+1)\times(n+1)}$ is symmetric, indefinite, and sparse.

2.3. Atom-condensed Kohn–Sham approximated to second order. The recently proposed ACKS2 model [35] can be regarded as an extension of the EE model. Essentially, ACKS2 was developed with the aim of correcting issues with accurately modeling dipole polarizability and charges during bond formation/dissociation via additional empirically fitted parameters. A block matrix representation of the linear system arising in the ACKS2 model is as follows:

$$\begin{bmatrix} \mathbf{H}_{\text{QEq}} & \mathbf{I}_n & \mathbf{1}_n & \mathbf{0}_n \\ \mathbf{I}_n & \mathbf{X} & \mathbf{0}_n^T & \mathbf{1}_n \\ \mathbf{1}_n^T & \mathbf{0}_n^T & 0 & 0 \\ \mathbf{0}_n^T & \mathbf{1}_n^T & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{q} \\ \mathbf{U} \\ \mu_{\text{mol}} \\ \lambda_U \end{bmatrix} = \begin{bmatrix} -\chi \\ \mathbf{0} \\ q_{\text{net}} \\ 0 \end{bmatrix}. \tag{2.7}$$

In (2.7), $\mathbf{U}$ contains Kohn–Sham potential coefficients, μ_{mol} and λ_U are Lagrangian multipliers from the underlying optimization problem, and X contains terms for the linear response kernel of the Kohn–Sham potential. Similarly to the EE model, the ACKS2 matrix $\mathbf{H}_{\text{ACKS2}} \in \mathcal{R}^{(2n+2)\times(2n+2)}$ is symmetric, indefinite, and sparse.

In Table 6 of Appendix A, we present the empirically fitted values utilized for studying the computational behavior of the charge models discussed in this section. In addition to the parameters mentioned in subsection 2.1, σ_i and Λ are ACKS2-specific parameters which control entries in the linear response kernel X, with the former being tuned for thresholding entries based on element types and the latter being used as a global bond softness parameter.

3. Preconditioning techniques for linear solvers. Two commonly used techniques for accelerating the convergence of iterative solvers are good initial guesses and preconditioning; the latter essentially amounts to leveraging better spectral properties of a transformed linear system. In the context of charge distribution models, the good initial guesses approach was previously explored in the form of spline extrapolations for QEq [2], which capitalized on the key observation that in a molecular simulation environment, atomic positions change slowly due to short timestep lengths. Thus, extrapolations from solutions to the linear systems in previous timesteps are likely to provide good initial guesses.

While initial guesses are easy to apply and can be very effective, a good preconditioner is crucial to ensure fast convergence. Aktulga et al. have also proposed an incomplete Cholesky based preconditioner for the QEq model, albeit in a purely sequential context [2]. Due to the difficulties in attaining a scalable implementation of the incomplete Cholesky technique, actual simulations carried out on distributed memory systems (such as those with PuReMD and LAMMPS (Large-Scale Atomic/Molecular Massively Parallel Simulator) [26]) still rely on the simple Jacobi preconditioning technique in the QEq solver. An illustration of the performance impact of this situation is that 60% or more of the total runtime in large-scale ReaxFF simulations is spent in the QEq kernel, as the cost of internode communications in the QEq solver dominates the execution time [3].

When one considers the EE and ACKS2 models, the need for efficient solvers is exacerbated. As shown in Table 7 of Appendix A, for the systems considered in this study (this fact seems to hold true in general), condition numbers of the coefficient matrices for the EE model tend to be slightly worse than those of the QEq model. For ACKS2 (where we only had empirically fitted force field parameters for the water system), the condition number is significantly worse than that for either QEq or EE.

Motivated by the factors above, identifying effective and efficient preconditioning techniques for the charge distribution models constitutes the main focus of this paper. One issue that prevents well-known preconditioners from being directly useful in this context is the fast execution requirement of MD simulations. The key enabler for our use of high-quality preconditioners is the same observation that allows good initial guesses: leveraging the slowly evolving characteristic of a molecular simulation, we amortize the cost of a preconditioner construction by reusing the same preconditioner over several timesteps. However, as we discuss below and demonstrate through numerical experiments in section 4, there still exist the following important trade-offs that must be considered in building a practical solver:

- *Effectiveness*: How much a preconditioner can increase the convergence rate of the solver.
- *Cost*: Time to compute and apply a preconditioner.

- *Longevity*: Number of subsequent timesteps in which a preconditioner can be used effectively without having to recompute it.
- *Parallelizability*: How scalable a preconditioner is in a parallel environment.

3.1. Approaches. In the subsections that follow, we describe the preconditioning techniques explored, which we refer to in the remainder of the paper by the bold notation shown in parentheses. The symbol $\mathbf{H}$ represents the generic coefficient matrix of the sparse linear system to be solved for a specified charge model (QEq, EE, or ACKS2).

3.1.1. Jacobi (Jacobi). A simple approach is to form a preconditioning matrix $\mathbf{P}^{-1}$ by taking the multiplicative inverses of the diagonal entries of $\mathbf{H}$. The ill-defined cases of zero entries on the diagonals of $\mathbf{H}_{\text{EE}}$ and $\mathbf{H}_{\text{ACKS2}}$ matrices can be treated by simply assuming that those zeros are ones for the Jacobi preconditioner.

Despite being an inexpensive and easily parallelizable preconditioner, the Jacobi preconditioner yields limited improvements in the solver convergence rate, because $\mathbf{P}^{-1}$ is a poor approximation to $\mathbf{H}^{-1}$ for the charge models considered due to the presence of a large number of off-diagonal entries (which is on the order of a few hundred per row) with not-so-insignificant nonzero values. Nevertheless, given its simplicity, Jacobi preconditioned solver performance serves as our baseline for measuring the improvements obtained by the other preconditioning techniques explored.

3.1.2. Incomplete LU (ILU) based techniques. The idea behind ILU preconditioning is to compute and leverage an approximate $\mathbf{LU}$ decomposition $\mathbf{H} \approx \mathbf{L}'\mathbf{U}'$, where $\mathbf{L}'$ and $\mathbf{U}'$ are approximate lower and upper triangular factors, respectively. ILU techniques are known to be among the most effective preconditioners [31], but an important question is how to determine the factors $\mathbf{L}'$ and $\mathbf{U}'$, i.e., which sparsity pattern to select so that the factors are computationally inexpensive to compute and apply, yet are effective as a preconditioner. A common choice for the sparsity pattern is the 0-fill-in approach, $\mathbf{ILU}(0)$, where $\mathbf{L}' + \mathbf{U}'$ has the same sparsity pattern as $\mathbf{H}$ [31]. Another effective choice utilizes the thresholding approach, $\mathbf{ILUT}(t)$, where the sparsity patterns in $\mathbf{L}'$ and $\mathbf{U}'$ are a subset of the 0-fill-in approach in which entries smaller than a prescribed threshold t are dropped during factorization [30]. Note that since the matrices in the QEq model are symmetric positive-definite, ILU preconditioning essentially amounts to computing an incomplete Cholesky ($\mathbf{IC}$) factorization with $\mathbf{L}' = (\mathbf{U}')^T$.

Sparsification strategies for factor computation. Plain $\mathbf{ILU}(0)$ and $\mathbf{ILUTP}(t)$ preconditioning turn out to be too costly to be useful in practice for our purposes. As such, we investigated a number of techniques for reducing the computational costs associated with construction and application of ILU based preconditioning techniques and improved their scaling. In particular, we developed the following two custom sparsification schemes for matrices arising in dynamic charge models:

- **Numeric dual drop strategy:** In this scheme, we start with a 0-fill-in preconditioner (i.e., $\mathbf{ILU}(0)$ or $\mathbf{IC}(0)$), but as in $\mathbf{ILUTP}(t)$, we apply a numerical threshold to drop small entries in the incomplete factors. Specifically, after a row in a factor is computed, the 1-norm of the row is computed, followed by a thresholding operation where 0-*fill-in* entries that are less than a predetermined threshold t times the 1-norm of that row are discarded. This hybrid scheme allows us to easily determine the location of the nonzeros (by virtue of not allowing fill-ins) and obtain sparser factors that contain only

the most significant entries (as in an **ILUTP** factorization). As such, it is an inexpensive, yet effective and easy-to-implement, preconditioner for the *spd* (symmetric positive definite) matrices of QEq, which we refer to as IC with dual drop (**ICDD(t)**), where the parameter t denotes the numerical threshold to be used. In the cases of EE and ACKS2 models, to prevent numerical breakdowns during factorization, we replace zeros on the diagonal with ones and perform an ILU decomposition because EE and ACKS2 matrices are not spd. We denote this scheme as ILU with dual drop (**ILUDD(t)**).

- **Distance drop strategy:** A number of other works have utilized insights in the underlying application in order to select which entries to threshold [8, 24]. We choose to follow these approaches and look to the underlying physics of the charge model problem. From the definition of the coefficient matrices for the charge distribution models in (2.1), we observe that as the distance between a pair of atoms increases, the corresponding off-diagonal entry in **H** decreases proportionately to the inverse of the distance. A different way of saying this is that small off-diagonal entries in **H** are contributed by atomic pairs separated by large distances. Intuitively, these entries contribute relatively less to the preconditioner quality; thus, they are good candidates for elimination. We denote the ILU/IC factors sparsified using the distance drop technique with the suffix **DS(d)**, e.g., **ICDD(t)+DS(d)** and **ILUDD(t)+DS(d)**, where d is a real parameter in the range $(0, 1]$ that is applied as a scaling of the distance cutoff of the nonbonded interaction radius r_{nonb}. The distance drop sparsification technique can be implemented simply by constructing a sparser **H** given a prescribed parameter d, and then performing an ICDD or ILUDD factorization. As will be demonstrated through numerical experiments, distance based sparsification is highly preferred across a large majority of our benchmark cases.

Parallelization of factor computation and application. In addition to the cost and effectiveness of the incomplete factors utilized, another important consideration is exploiting high degrees of parallelism available on high-performance computing systems. Due to the need for eliminating data race conditions in a parallel ILU computation, the nonzero pattern of the sparse matrix plays an important role during both computation and application of the approximate factors. In this regard, it can be argued that the above sparsification schemes increase the degree of available parallelism by virtue of reducing the overlap between different rows of the coefficient matrices. To further improve the scalability of our ILU preconditioners, we implemented the following techniques:

- **Balanced level scheduling using graph coloring**: Fundamentally, this approach is based on the idea of level scheduling. Using the row dependency graph, groups of independent rows (i.e., levels) are determined. During the subsequent factorization, threads perform the scaling and elimination operations in parallel within a level, and computations between subsequent levels proceed in a lock-step fashion. Therefore the degree of parallelism is heavily dependent on the sparsity pattern.
  As shown in our previous work [23], a pure level scheduling approach can produce poor thread scalability in situations where there is a large number of dependencies between the rows. In order to improve the performance of level scheduling, many other works have explored the use of reorderings using

graph coloring [4, 7, 14]. For the solvers in this work, we utilize an iterative, graph coloring approach [9] to break data dependencies through an explicit permutation of $\mathbf{H}$ [21]. In this method, a permutation matrix $\mathbf{Q}$—which increases the amount of independent nodes per level—is determined by first finding an approximate coloring of the dependency graph. This permutation matrix can be applied as $\mathbf{Q}^T\mathbf{HQ}$ and subsequently factored as $\mathbf{Q}^T\mathbf{HQ} \approx \mathbf{L'_P U'_P}$ using level scheduling [20]. The graph coloring approach can be used with both the numerical and distance drop strategies (or their combination) discussed above.

- **Fine-grained ILU (FG-ILUDD(t, s))**: This iterative approach, recently proposed for computing the incomplete factors, seeks to address the issue of limited parallelism in ILU preconditioning [10]. Here, the factorization problem is posed as a constraint optimization problem where one seeks to satisfy a series of equations of the form

$$\sum_{k=1}^{\min(i,j)} L_{ik}U_{kj} = H_{ij} \tag{3.1}$$

by asynchronously updating individual nonzeros in the sparse factors. A single pass over all nonzeros in the factors is termed a "sweep"; at each sweep, the factors computed by this algorithm with fine-grained parallelism get asymptotically closer to the actual $\mathbf{L'}$ and $\mathbf{U'}$ factors. We use the parameter s to denote the number of sweeps used to compute the factors in our implementation. To achieve convergence of the factors, $\mathbf{H}$ must have a unit diagonal, which may require diagonal scaling. Additionally, we assume the convention that the factor $\mathbf{L'}$ has unit diagonal entries.
The numerical dual dropping strategy can still be applied with fine-grained ILU. In this scheme, which we refer to as **FG-ILUDD(t, s)**, after all sweeps are completed, we perform a postprocessing step where all nonzeros are scanned and those below the specified numerical threshold t times the 1-norm of that row are dropped. On the other hand, distance drop sparsification is straightforward to use here because it applies the sparsification before factorization, exactly as required by the fine-grained ILU method, and it can be highly effective because our numerical experiments show that construction of the fine-grained ILU preconditioner can be very costly despite being easily parallelizable.

3.1.3. Sparse approximate inverse (SAI(τ)). A technique that is generally less effective than ILU preconditioning, but lends itself more easily to parallelization, is sparse approximate inverse (SAI) preconditioning. Given $\mathbf{H}$, SAI aims to find an approximate inverse matrix $\mathbf{M}$ such that the right preconditioned system $\mathbf{HMy} = \mathbf{b}$, with $\mathbf{x} = \mathbf{My}$ (or the left preconditioned system $\mathbf{MHx} = \mathbf{Mb}$) can be solved with fewer solver iterations than the original sparse linear system and, ideally, with a lower overall execution time. Generally, there are three popular categories of SAI techniques: Frobenius norm minimization, factorized sparse approximate inverse, and incomplete biconjugation [6]. In this work, we focus on the first category. Hence, the objective is to minimize $||\mathbf{I} - \mathbf{HM}||$ for right preconditioning (or to minimize $||\mathbf{I} - \mathbf{MH}||$ for left preconditioning).

To control the cost of constructing and applying the SAI preconditioner, often $\mathbf{M}$ is chosen to be no denser than the coefficient matrix $\mathbf{H}$ itself [12]. Given the relatively

large number of nonzeros per row in our problems, it is crucial that the number of nonzeros in the approximate inverses is significantly lower than that of the charge distribution matrix. Our experiments on a number of sample problems have shown that the positions of the numerically large nonzeros in the charge matrix constitute good candidates for the sparsity pattern of $\mathbf{M}$. In accordance with this observation, we select only the τ percent largest nonzeros in absolute value for inclusion when computing the approximate inverse matrix.

During construction of the SAI preconditioner, a benefit of using the Frobenius form is that we can easily exploit parallelism because the SAI factorization is formulated as

$$||\mathbf{I} - \mathbf{HM}||_F^2 = \sum_{j=1}^{n} ||\mathbf{e}_j - \mathbf{Hm}_j||_2^2 . \tag{3.2}$$

Here, $\mathbf{e}_j$ is the jth column of the identity matrix, and $\mathbf{m}_j$ is the jth column of $\mathbf{M}$. Leveraging the fact that $\mathbf{H}$ is sparse, we build least squares problems with smaller dense matrices $\hat{\mathbf{H}}_j$ and vectors $\hat{\mathbf{m}}_j$, where $\hat{\mathbf{H}}_j$ is obtained by eliminating the rows and columns of $\mathbf{H}$ that do not correspond to the set of nonzeros found in the row of $\mathbf{M}$ currently being computed (see section 2.1 of [5] for details of this construction). Thus, the resulting least squares problems $||\hat{\mathbf{e}}_j - \hat{\mathbf{H}}_j \hat{\mathbf{m}}_j||_2^2$ can be solved independently through QR decomposition of $\hat{\mathbf{H}}_j$.

We note that the distance based sparsification of the coefficient matrix can be easily applied for SAI preconditioning as well. In fact, distance based sparsification can be highly effective in this context by reducing the cost of QR factorizations without much impact on the effectiveness of the SAI preconditioner. Additionally, we utilize optimized LAPACK libraries (such as Intel MKL or Cray LibSci) to perform the QR decomposition tasks, each of which is executed sequentially by different threads using dynamic assignment for load balancing purposes.

3.2. Dynamic determination of preconditioner recomputation. An important insight we gained when designing our preconditioned solver is that the molecular systems (hence the corresponding linear systems) change slowly over the course of a simulation. To leverage this insight, preconditioners are only occasionally computed and reused for several solves to amortize the preconditioner construction costs. The natural question which arises from this is exactly when to recompute the preconditioner. In Algorithm 3.1, we give a heuristic approach that adopts a gain-loss comparison perspective for recomputing the preconditioner versus reusing it.

In this approach, we make the assumption that a preconditioner is most effective in terms of improving solver convergence at the simulation step in which it was computed. The preconditioned solve time required right after a preconditioner is recomputed is recorded as the ideal solve time for this preconditioner (denoted T_{ideal}). From this point onward, we accumulate the excess solve time compared to this ideal solve time (T_{loss}); this is considered the loss incurred by reusing the preconditioner. When the aggregated loss time exceeds the time it would take to recompute the preconditioner (T_{PreComp}) times α, we proceed to recompute it; otherwise, we continue reusing it.

This dynamic scheme essentially justifies the reconstruction of a preconditioner by detecting when the loss of effectiveness of the preconditioner would exceed α times the cost of recomputing it, and tries to balance the gains and losses in preconditioner computation time versus other solver time (preconditioner application, solver operations including sparse matrix-dense vector multiplications and vector operations). We study the consequences of this heuristic in further detail in subsection 4.2.

Algorithm 3.1. Dynamic determination of preconditioner recomputation.

1: Select $\alpha \geq 0$
2: **for** each simulation step **do**
3: **if** $T_{\text{loss}} \geq \alpha \cdot T_{\text{PreComp}}$ **or** first simulation step **then**
4: (Re)compute the preconditioner
5: Define T_{PreComp} as the preconditioner computation time from line 3
6: Perform solve(s)
7: Define T_{ideal} as the solve time from line 5
8: Update $T_{\text{loss}} = 0$
9: Check if the preconditioner quality is satisfactory and take action accordingly
10: **else**
11: Perform solve(s)
12: Define T_{actual} as the solve time from line 10
13: Update T_{loss} += $T_{\text{actual}} - T_{\text{ideal}}$
14: **end if**
15: **end for**

To gain insight into how to select the value for parameter α in Algorithm 3.1, consider the following proof for the α, which minimizes total solver execution time under a fixed preconditioner cost and linear degradation model for solve time with preconditioner reuse.

THEOREM 3.1 (optimal parameterization for fixed preconditioner recomputation). *Suppose that for a sequence of n linear systems, a preconditioned solver is utilized where the preconditioner is recomputed and reused according to t Algorithm* 3.1. *Assuming that the solver performance degradation due to preconditioner reuse across solves is linear (solve time increases linearly with reuse) and the cost is identical for each time the preconditioner is recomputed, the value which minimizes total solve execution time is $\alpha = 1 \pm \sqrt{\frac{d}{T_{PreComp}}}$, where d is the increased solve time per step due to reusing the preconditioner.*

Proof. Let r be the number of solves for which the preconditioner is used. First, observe that from line 3 of Algorithm 3.1, with our linear degradation assumption, we have

$$\begin{aligned} \alpha \cdot T_{\text{PreComp}} &= T_{\text{loss}} \\ &= 0 + d + 2d + \cdots + (r-1) \cdot d \\ &= d \cdot \frac{r^2 - r}{2}. \end{aligned}$$

Equivalently, we have $r = \sqrt{\frac{2\alpha \cdot T_{\text{PreComp}}}{d} + \frac{1}{4}} + \frac{1}{2}$. Next, considering the total solve

time, we observe the following (where, for simplicity, we assume r divides n):

$$\begin{aligned} T_{\text{total}} &= \sum_{i=1}^{n} T_{\text{actual}}^{(i)} + \frac{n}{r} \cdot T_{\text{PreComp}} \\ &= n \cdot T_{\text{ideal}} + \sum_{i=1}^{n} T_{\text{loss}}^{(i)} + \frac{n}{r} \cdot T_{\text{PreComp}} \\ &= n \cdot T_{\text{ideal}} + \frac{nd}{2} \left[\sqrt{\frac{2\alpha \cdot T_{\text{PreComp}}}{d} + \frac{1}{4}} - \frac{1}{2} \right] \\ &\quad + n \cdot T_{\text{PreComp}} \left[\sqrt{\frac{2\alpha \cdot T_{\text{PreComp}}}{d} + \frac{1}{4}} + \frac{1}{2} \right]^{-1}. \end{aligned}$$

Finally, we see that when $\frac{\partial T_{\text{total}}}{\partial \alpha} = 0$, the minimum of T_{total} is achieved when $\alpha = 1 \pm \sqrt{\frac{d}{T_{\text{PreComp}}}}$. ∎

We note that in practice, $\sqrt{\frac{d}{T_{\text{PreComp}}}} \approx 0$, so we select $\alpha = 1$ for our experiments in section 4.

3.3. Solver implementation. As mentioned above, through carefully controlling the frequency of preconditioner construction, costs associated with computing the approximate inverses in SAI or the incomplete factors in ILU/IC schemes become negligible in practice. However, despite utilizing good initial guesses and the described preconditioning techniques, our solvers typically take tens to hundreds of iterations to converge depending on the charge model and the convergence tolerance used. Hence, implementation of the preconditioned solver is important from a performance point of view.

Since EE and ACKS2 models produce indefinite coefficient matrices, our implementation uses a restarted GMRES solver [32], which has, in fact, been observed to yield better convergence for QEq compared to conjugate gradient and minimum residual solvers, despite the coefficient matrices being symmetric semipositive definite in QEq. In a left preconditioned restarted GMRES(k) implementation, with k being the maximum number of iterations before restarting, the main parts are the sparse matrix-dense vector multiplications (SpMVs), the application of a preconditioner (PreApp), and the dense vector operations (VecOps). In PuReMD, the sparse coefficient matrix **H** is stored in CSR (compressed sparse row) format, as such solver SpMVs are implemented by using loop parallelization over matrix rows. In the PreApp phase of the **ICDD/ILUDD** and **FG-ILUDD** approaches, parallelization is applied across rows within each level of the level scheduling with graph coloring technique. For the **SAI** preconditioner, PreApp essentially amounts to an SpMV operation, again using CSR format with simple parallelization of the loop over matrix rows. Finally, VecOps are also implemented by using simple loop parallelization.

Overall, with a simple Jacobi preconditioner, SpMV costs are expected to dominate the overall computation, while VecOp or PreApp times are negligible. However, for **SAI**, **ICDD/ILUDD,** or **FG-ILUDD** preconditioners, PreApp time is likely to be significant.

4. Numerical experiments.

4.1. Computing environment and benchmark systems. Results from numerical experiments presented in the following subsections have been obtained on

Laconia, a cluster with over 400 compute nodes at Michigan State University's High Performance Computing Center.[1] Each of the base compute nodes on Laconia has 28 cores, consisting of two 14-core Intel Xeon E5-2680v4 Broadwell 2.4 GHz processors, and has 128 GB DDR3 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). Between the 14 cores on a single "Broadwell" processor, a 35 MB L_3 cache is shared. At the time of the experiments, the Laconia nodes ran CentOS version 7.6.1810 distribution of GNU/Linux for x86_64 architectures, kernel version 3.10.0-957.5.1, and glibc version 2.17-260.el7_6.3.

The preconditioned solvers detailed in section 3 were implemented in the shared memory version of the PuReMD ReaxFF software [2, 1]. The software was built using the GNU Compiler Collection version 8.2.0 with the optimizations enabled from the `-O3` flag. For numeric libraries, Intel MKL version 2019.1.144 was utilized. For the experiments, we restricted our simulations to a single socket on Laconia to avoid performance issues due to nonuniform memory access (NUMA) effects. In these cases, the maximum number of OpenMP threads was set to 14 with no hyperthreading enabled.

For benchmarking purposes, four characteristically varied molecular systems from various application scenarios of reactive and polarizable force fields were selected. These systems, which were comprised of bulk water (H_2O), amorphous silica (SiO_2), pentaerythritol tetranitrate (PETN, $C_5H_8N_4O_{12}$), and phospholipid bilayer (not solvated in water), represent liquids, amorphous materials, perfect crystals, and soft matters, respectively. Quantitatively, each system contained between 48,000 and 78,000 atoms; thus, these systems represented moderately sized molecular structures. Table 1 summarizes the systems used for performance evaluation.

TABLE 1

Molecular systems used in performance evaluation, with the third column indicating the number of atoms in the system (N)*, and the fourth column denoting the dimensions of the rectilinear simulation boxes in* $\mathbb{R}^3$ *in angstroms* (Å).

Name	Chem. rep.	N	Sim. box dims. (Å)	Category
Water (W)	H_2O	78480	$120.9 \times 80.6 \times 80.6$	Liquid
Silica (S)	SiO_2	72000	$109.4 \times 100.4 \times 104.2$	Amorphous
PETN (P)	$C_5H_8N_4O_{12}$	48256	$114.5 \times 114.5 \times 155.5$	Crystal
Bilayer (B)	Phospholipid	56800	$82.7 \times 81.5 \times 80.0$	Soft matter

Unless indicated otherwise, all simulations were conducted with the following parameters: periodic boundary conditions enabled, 0.25 fs timestep lengths, NVE ensembles, $r_{\text{nonb}} = 10$ Å, and 2000 total simulation timesteps. For the charge distribution solvers, the initial guesses were extrapolated from solutions in earlier timesteps, with cubic and quadratic spline extrapolation, respectively, used for the two linear systems arising in the QEq model solvers in (2.3), while cubic spline extrapolation was used for the EE and ACKS2 models. All reported data have been averaged over the 2000 simulation timesteps.

Before moving on to the numerical results, in Table 2 we give a qualitative summary of the trade-offs involved with the preconditioning approaches we explore. As indicated in this table and demonstrated quantitatively in the subsections that follow, it is crucial to identify preconditioners that are (i) effective at reducing the number

[1] https://wiki.hpcc.msu.edu/pages/viewpage.action?pageId=20120131

TABLE 2
Qualitative behavior of the preconditioning techniques.

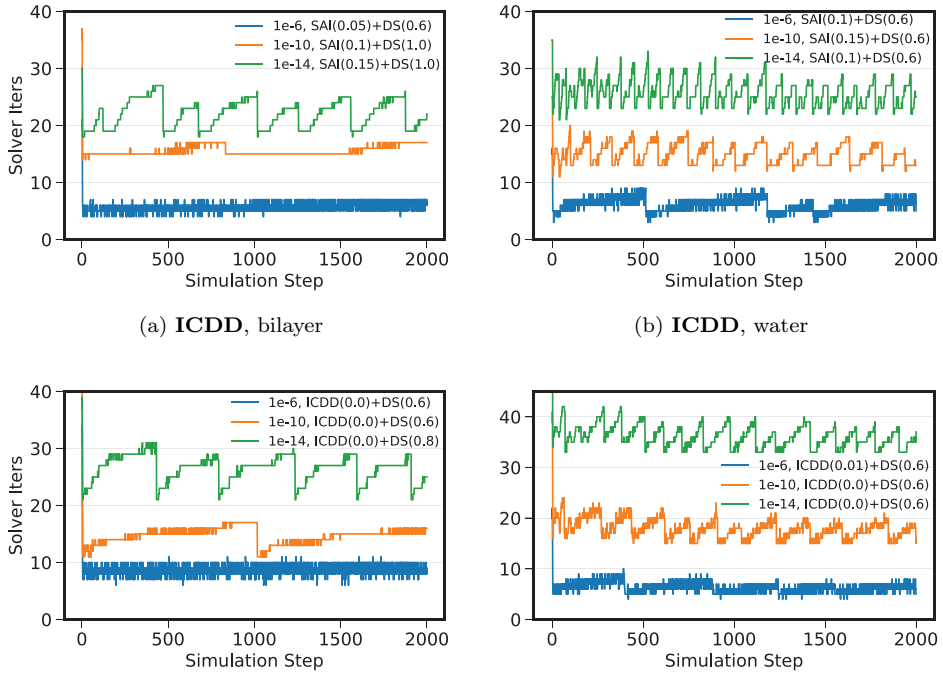
Preconditioner	Cost	Longevity	Effectiveness	Parallelism
Jacobi	Very low	High	Low	High
ICDD(t)	Moderate	Moderate	High	Low
ILUDD(t)	High	Moderate	High	Low
FG-ICDD(t, s)	Very high	Moderate	High	High
FG-ILUDD(t, s)	Very high	Moderate	High	High
SAI(τ)	Moderate	High	Moderate/high	High

of solver iterations, (ii) computationally inexpensive, and (iii) easily parallelizable, especially in the PreApp phase. In the following, we examine each of these trade-off points to identify preconditioners with optimal overall performance.

4.2. Preconditioner longevity. As discussed previously and indicated in Table 2, construction of ILU and SAI based preconditioners constitutes a significant cost in terms of computational time. To reduce this cost, we leverage the fact that atomic positions, and hence the coefficient matrices, evolve slowly over the course of a simulation; as such, a preconditioner, once computed, is reused over several timesteps. An important consideration is then the number of subsequent timesteps for which a preconditioner can be effective. In Figure 1, we show the longevity of the **ICDD** and **SAI** preconditioners (the former behaves very similarly to its **FG-ICDD** and ILU based counterparts) for the bilayer and bulk water systems under our dynamic refactorization scheme. For systems in which atoms have limited mobility, such as the bilayer system, we observe that both preconditioners remain effective for hundreds to thousands of steps. For systems such as water, in which atoms can move more freely, we observe that the heuristic employed in Algorithm 3.1 leads to significantly more frequent preconditioner recomputations to decrease the overall solve time. Additionally, we observe that as solver tolerances decrease, the frequency of preconditioner recomputation tends to increase as the potential losses that result—in terms of solve time—from using a poor quality preconditioner tend to increase (while the preconditioner construction time remains constant). To precisely quantify the frequency, we report in Table 3 the total number of preconditioner recomputations during the course of the 2000 step simulations.

Overall, these results provide a strong basis for the fact that preconditioner construction costs can be amortized by reusing ILU and SAI based preconditioners over hundreds of steps for high tolerances (e.g., 10^{-6}) and over tens of steps for lower tolerances (e.g., 10^{-10} or 10^{-14}). As will be shown in subsequent results (see Figure 2), such reuse rates are sufficient to reduce the preconditioner construction costs to only a small percentage of the overall solver time. As such, the term "preconditioner cost" will solely refer to the preconditioner application cost (but not its construction cost) in the remainder of this section.

4.3. Tuning of preconditioner parameters for cost and effectiveness. With our proposed sparsification techniques, there is a large number of possible configurations to choose from for each preconditioner (beyond the Jacobi method). Since there exist important trade-offs regarding the effectiveness (convergence rate) and cost of different configurations, we conducted a parameter search study to determine which parameters to utilize for each scheme. In Table 4, we present a small snapshot from this study, in which parameters that control preconditioner quality and sparsity for various schemes were varied for the QEq model. To illustrate the aforemen-



(a) **ICDD**, bilayer (b) **ICDD**, water

(c) **SAI**, bilayer (d) **SAI**, water

FIG. 1. *Longevity of the **ICDD** and **SAI** preconditioners for the bilayer (left column) and bulk water (right column) systems with the QEq model at various solver tolerances.*

TABLE 3

Number of preconditioner recomputations over the 2000 *simulation steps for the **ICDD**, **ILUDD**, **FG-ICDD**, **FG-ILUDD**, and **SAI** preconditioned solvers.*

Solver tolerance		10^{-6}				10^{-10}				10^{-14}			
Systems		B	P	S	W	B	P	S	W	B	P	S	W
CM	Preconditioner												
QEq	**ICDD**	1	7	1	4	2	28	2	16	3	45	2	27
	FG-ICDD	1	2	1	2	2	9	2	4	3	13	2	6
	SAI	1	3	1	5	2	12	2	13	6	20	2	17
EE	**ILUDD**	2	4	1	2	2	9	2	7	2	13	2	11
	FG-ILUDD	1	1	1	1	1	4	1	3	1	6	1	4
	SAI	5	16	1	3	3	22	2	8	16	35	5	12
ACKS2	**ILUDD**	-	-	-	23	-	-	-	35	-	-	-	55
	FG-ILUDD	-	-	-	5	-	-	-	8	-	-	-	13
	SAI	-	-	-	9	-	-	-	12	-	-	-	22

tioned trade-offs, focusing on **ICDD(0.0)+DS(1.0)** and **ICDD(0.01)+DS(0.8)**, we note that the former yields fewer solver iterations (5.4 versus 5.7 for a tolerance of 10^{-6}, widening to 29.6 versus 32.0 for 10^{-14}), but the latter produces the lower mean solve times (0.01 versus 0.03 for 10^{-6}, widening to 0.03 versus 0.08 for 10^{-14}). Note that during application of the IC preconditioner, each nonzero requires a single

multiply/add operation, and therefore fewer nonzeros translates into fewer floating point operations. The better solve time for **ICDD(0.01)+DS(0.8)** can then be attributed to a lower preconditioner application cost as it has an order of magnitude less nonzeros than the **ICDD(0.0)+DS(1.0)** variant. Also, as we show in more detail in the next subsection, fewer nonzeros allows higher parallelizability during preconditioner application. Similar trade-offs for other techniques can be observed in Table 4. Overall, we notice that sparsification strategies hit the "sweet spot" in between the two extremes of achieving high effectiveness and keeping costs as low as possible.

TABLE 4

*Preconditioner parameter comparison for cost (number of nonzeros), mean solver iterations (SI), and mean solver time (ST) in seconds with a smaller version of the bulk water system (*6540 *atoms) for* 100 *simulation steps using the QEq charge method. Preconditioners were computed once at step* 0 *and reused over the course of the simulation. The top row indicates the specified solver tolerance. Dynamic refactorization was disabled for this experiment.*

Solver tolerance		10^{-6}		10^{-10}		10^{-14}	
Preconditioner	Cost	SI	ST	SI	ST	SI	ST
ICDD(0.0)+DS(0.6)	2.98E5	5.7	0.01	17.5	0.02	32.2	0.03
ICDD(0.0)+DS(0.8)	7.05E5	5.4	0.02	16.5	0.03	30.3	0.05
ICDD(0.0)+DS(1.0)	1.37E6	5.3	0.03	16.1	0.05	29.6	0.08
ICDD(0.01)+DS(0.6)	1.46E6	5.7	0.01	18.1	0.02	32.6	0.03
ICDD(0.01)+DS(0.8)	1.57E5	5.6	0.01	17.6	0.02	32.0	0.03
ICDD(0.01)+DS(1.0)	1.59E5	5.6	0.01	17.4	0.02	31.7	0.04
FG-ICDD(0.0,2)+DS(0.6)	2.98E5	5.6	0.01	17.8	0.03	32.7	0.05
FG-ICDD(0.0,2)+DS(0.8)	7.05E5	5.5	0.02	17.1	0.05	31.4	0.09
FG-ICDD(0.0,2)+DS(1.0)	1.37E6	5.5	0.04	17.1	0.10	31.3	0.17
FG-ICDD(0.0,3)+DS(0.6)	2.98E5	5.6	0.01	17.4	0.03	32.0	0.05
FG-ICDD(0.0,3)+DS(0.8)	7.05E5	5.4	0.02	16.4	0.05	30.2	0.09
FG-ICDD(0.0,3)+DS(1.0)	1.37E6	5.3	0.05	16.3	0.11	29.9	0.18
FG-ICDD(0.1,2)+DS(0.6)	2.28E4	11.1	0.01	38.6	0.02	74.1	0.07
FG-ICDD(0.1,2)+DS(0.8)	2.29E4	11.0	0.02	38.6	0.02	74.0	0.07
FG-ICDD(0.1,2)+DS(1.0)	2.32E4	11.0	0.02	38.6	0.02	74.0	0.08
FG-ICDD(0.1,3)+DS(0.6)	2.28E4	10.9	0.01	37.9	0.02	72.8	0.07
FG-ICDD(0.1,3)+DS(0.8)	2.29E4	10.8	0.02	37.9	0.02	72.6	0.07
FG-ICDD(0.1,3)+DS(1.0)	2.32E4	10.8	0.03	37.9	0.02	72.5	0.08
SAI(0.01)+DS(0.6)	6.54E3	18.9	0.01	62.2	0.04	134.0	0.12
SAI(0.01)+DS(0.8)	7.58E3	18.9	0.01	63.0	0.04	135.3	0.12
SAI(0.01)+DS(1.0)	2.09E4	10.6	0.01	35.0	0.02	72.3	0.06
SAI(0.5)+DS(0.6)	2.33E4	10.1	0.01	33.0	0.02	67.5	0.05
SAI(0.5)+DS(0.8)	6.40E4	6.7	0.01	21.0	0.02	40.4	0.03
SAI(0.5)+DS(1.0)	1.31E5	14.5	0.01	30.3	0.03	58.4	0.05
SAI(0.1)+DS(0.6)	5.36E4	6.8	0.01	21.5	0.02	41.1	0.03
SAI(0.1)+DS(0.8)	1.35E5	14.4	0.01	29.9	0.02	58.7	0.05
SAI(0.1)+DS(1.0)	2.68E5	8.4	0.01	23.0	0.01	43.2	0.04

Using insights from the above study, high quality parameters were selected for each preconditioning scheme for a given molecular system and convergence tolerance by scanning the potential values of preconditioner parameters. These parameters are the dual drop threshold $t \in \{0.0, 0.1, 0.01\}$ in **ICDD**(t), **ILUDD**(t), **FG-ICDD**(s,t), and **FG-ILUDD**(s,t); the additional parameter s for the latter two methods is the number of sweeps (ranging from 2 to 4). For SAI, the percentage of the top nonzeros to be kept (in terms of magnitude) varied from 3% to 15% in 1.5% increments. For all solvers, distance based pruning was also investigated for values of $d \in \{0.6, 0.8, 1.0\}$. In the results presented below, the combination that yields the

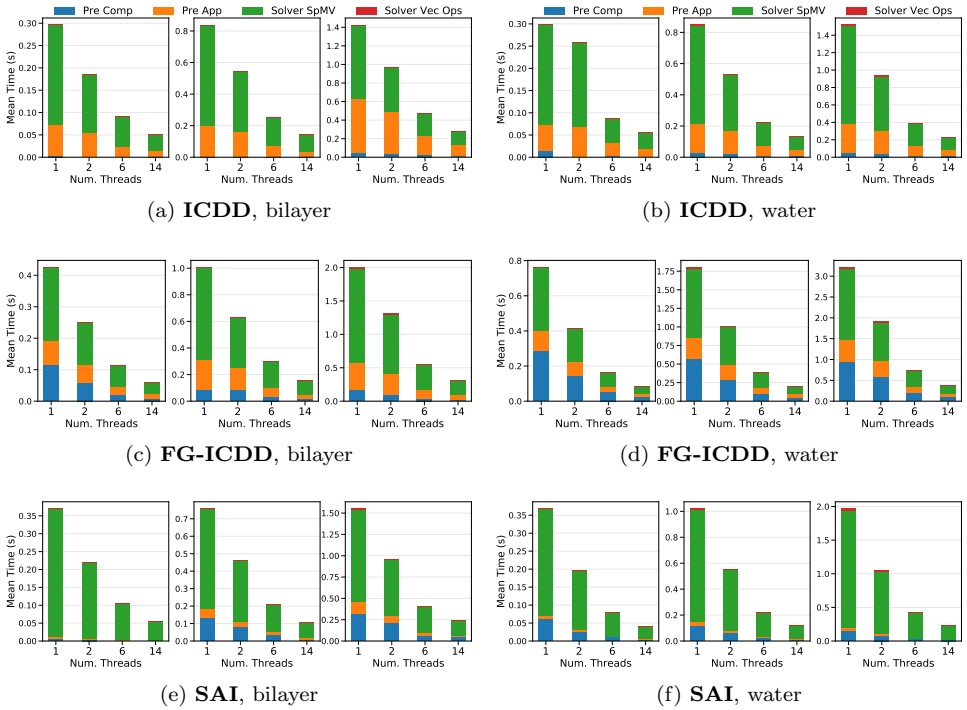
best execution time for each preconditioner is used. We report these parameters in Table 8 of Appendix A. As can be seen in that table, both dual dropping and distance based pruning are heavily utilized across the board by different preconditioning schemes, providing evidence of their merit. We note that such a parameter search can be made an integral part of the PuReMD software itself, so that an automated tuning is performed for the particular simulation and architecture for the first few hundred to thousand steps of a simulation. Considering the fact that a production simulation typically takes millions to billions of timesteps, overhead incurred would be negligible.

TABLE 5

*Comparison of the application cost and effectiveness of the **Jacobi**, **ICDD**, **ILUDD**, **FG-ILUDD**, and **SAI** preconditioned solvers for the bulk water system. For cost, the number of nonzeros in the preconditioner is reported for the chosen parameters at a solver tolerance of 10^{-6} (sum of factors for **ILU** based methods). For effectiveness, the mean solver iterations* (SI) *and mean solve time* (ST) *in seconds are given for various solver tolerances (top row). To enable easier comparisons, the solver iterations for the preconditioners have been normalized for improvement over the Jacobi preconditioner (bold and unnormalized).*

Solver tolerance			10^{-6}		10^{-10}		10^{-14}	
CM	Prec.	Cost	SI	ST	SI	ST	SI	ST
QEq	**None**	0	0.8x	0.12	0.8x	0.51	0.9x	0.97
	Jacobi	7.84E4	**16.3**	0.09	**62.7**	0.37	**128.4**	0.80
	ICDD	3.46E6	2.7x	0.06	4.3x	0.14	5.0x	0.25
	FG-ICDD	7.16E6	2.3x	0.08	3.3x	0.21	3.8x	0.38
	SAI	6.42E5	2.6x	0.05	3.5x	0.13	3.6x	0.24
EE	**None**	0	0.8x	0.10	0.8x	0.30	0.9x	0.55
	Jacobi	7.84E4	**12.2**	0.08	**36.2**	0.23	**80.9**	0.55
	ILUDD	1.97E6	2.4x	0.05	3.9x	0.10	4.4x	0.17
	FG-ILUDD	7.32E6	1.8x	0.06	3.0x	0.16	3.4x	0.26
	SAI	6.53E5	2.8x	0.03	3.5x	0.08	3.5x	0.15
ACKS2	**None**	0	0.1x	14.98	0.1x	27.58	0.1x	64.60
	Jacobi	1.56E5	**125.6**	1.62	**216.1**	3.01	**425.7**	5.94
	ILUDD	5.07E6	5.4x	0.43	5.0x	0.81	4.5x	1.74
	FG-ILUDD	8.48E6	2.4x	1.02	2.9x	1.55	2.6x	3.38
	SAI	6.34E5	2.2x	0.72	2.3x	1.25	2.1x	3.06

Next, in Table 5 we demonstrate the trade-off between preconditioner cost and effectiveness for parameters determined as a result of the above search procedure across all charge models. As expected, despite its low cost, **Jacobi** is the least effective in improving the convergence rate, while still yielding important improvements over the unpreconditioned case, especially for the ACKS2 charge model. Incomplete factorization methods **ICDD**, **ILUDD**, **FG-ICDD**, and **FG-ILUDD** are the most effective. The effectiveness gap between **Jacobi** and ILU based schemes widens as the convergence tolerance decreases from 10^{-6} to 10^{-10}, with fewer gains when moving to 10^{-14}. However, this effectiveness comes at the expense of increased preconditioner costs, for which the number of nonzeros is about the same order of magnitude as the charge matrices themselves. Comparatively, the cost of **SAI** is between that of the **Jacobi** and ILU schemes, but the effectiveness of **SAI** is closer to the latter. In addition to the cost and effectiveness examined here, parallel efficiency of applying the preconditioners represents an important consideration for overall performance in actual simulations. We examine this issue next.



(a) **ICDD**, bilayer (b) **ICDD**, water

(c) **FG-ICDD**, bilayer (d) **FG-ICDD**, water

(e) **SAI**, bilayer (f) **SAI**, water

FIG. 2. *Strong scaling plots of three preconditioned solvers with QEq for the bilayer (left column) and bulk water (right column) systems. Figures from left to right within a grouping show results at convergence tolerance levels of* 10^{-6}, 10^{-10}, *and* 10^{-14}.

4.4. Scalability of preconditioned solvers. In order to analyze the scalability of our preconditioned solvers, strong scaling experiments were performed on a single socket of a dual socket 28-core node on Laconia. To better illustrate the tradeoffs of the preconditioning approaches, the mean solver time was broken down by the major computational kernels in the solver: sparse matrix-dense vector multiplications (SpMVs), preconditioner computation (PreComp), preconditioner application (PreApp), and the remaining vector operations (VecOps).

As shown in Figure 2, the percentage of time spent in the preconditioner computation is insignificant for the **ICDD** and **SAI** methods as a result of preconditioner reuse. However, the iterative approach used for asynchronously computing the **FG-ICDD** preconditioner takes a significant percentage of the overall solver execution time, despite reusing the preconditioners and keeping the number of sweeps very low (2–4 sweeps). The high cost of preconditioner computation renders the **FG-ICDD** method less competitive—at least on the multicore architectures on which we have experimented. But the fact that preconditioners generated by **FG-ICDD** exhibit convergence rates similar to those of the regular **IC** techniques shows that **FG-ICDD** can provide advantages on massively parallel architectures, such as GPUs or Xeon Phis.

In terms of parallel scaling, **ICDD** achieves strong scaling efficiencies ranging from 36 to 46 percent through 14 threads for the bilayer and bulk water systems, as shown in Figure 2. Improvements over **ICDD** are observed for **FG-ICDD** with scaling efficiencies ranging between 45 and 65 percent. For the **SAI** preconditioned

solver, we see efficiencies between 45 and 50 percent for the bilayer system, and between 58 and 59 percent for the bulk water system. Given that the vast majority of operations involved in the preconditioned solvers have low arithmetic intensities and are therefore memory-bound, the observed efficiencies are in the expected range.

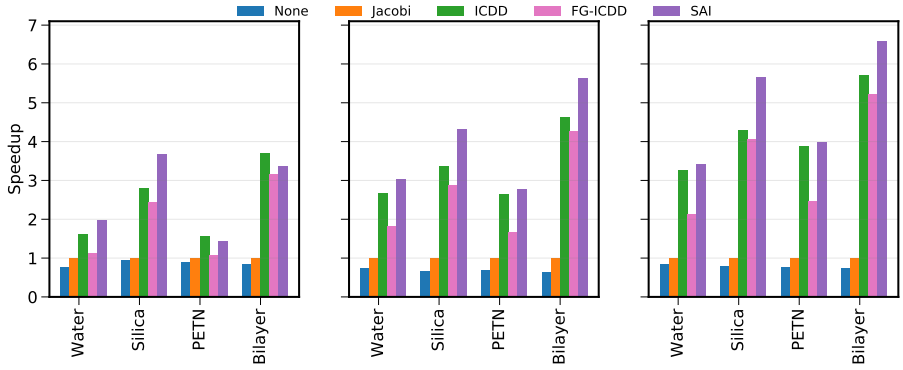
An important observation in Figure 2 is the relatively higher percentage of time spent in the PreApp phase for **ICDD** and **FG-ICDD** when compared to the **SAI** preconditioner. One would normally expect the PreApp phase of the ILU based methods to exhibit poorer scaling than that of the **SAI** method, as the latter boils down to a simple SpMV operation, whereas the former requires level-scheduled forward and backward solves. However, we observe that the PreApp phase for the ILU based methods (which exploits graph coloring based reordering of matrix rows) and for the **SAI** methods scales roughly at the same rate, which is also similar to the scaling efficiency of the SpMV kernel. As such, the reason for the high percentage of time spent by ILU based methods in the PreApp phase is directly related to the cost of applying the preconditioners, which is essentially characterized by the number of nonzeros they contain (as given in Tables 4 and 5).

4.5. Preconditioned solver performance. In this subsection, we analyze the overall benefits of the preconditioning techniques we explored (on a single socket, i.e., 14 cores, of a node on Laconia). Figure 3 shows the speedups over the Jacobi preconditioner for our preconditioned solvers on the four benchmark systems. Note that we take the Jacobi preconditioner as our base case due to its simplicity, but we also show the unpreconditioned solver performance for reference. We observe that for QEq, the preconditioners **ICDD**, **FG-ICDD**, and **SAI** result in speedups of 1.1–6.7 times the mean solve time using the Jacobi preconditioner, with speedups generally improving as the solver tolerance decreases. A similar trend is observed for these preconditioners with the EE model.

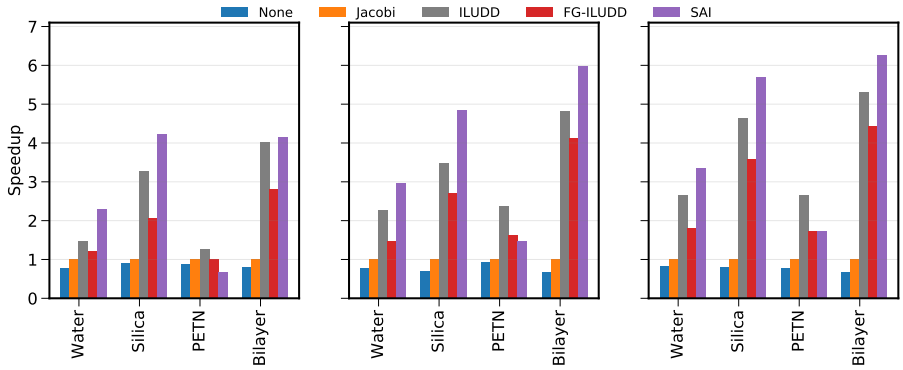
In the case of the ACKS2 model, only the force field parameters for the bulk water system were available to us. For bulk water, **SAI** produces speedups ranging from 1.8 to 2.4, which are similar to the speedups observed in the QEq and EE models. On the other hand, **ILUDD** produces speedups ranging from 3.3 to 3.7 for ACKS2, despite being less parallelizable than **SAI**. In fact, **ILUDD** reduces the Jacobi solver iteration numbers by about a factor of 5, as can be seen in Table 5, compared to the roughly 2.2× improvement observed with **SAI**. This suggests that the quality of the **SAI** preconditioner can potentially be improved further by selecting a better sparsity pattern—we currently do not make any distinction between the $\mathbf{H}_{\mathrm{QEq}}$ and $\mathbf{X}$ subblocks in the ACKS2 matrices while selecting the sparsity pattern.

In contrast to the **ILUDD** and **SAI** techniques, the **FG-ILUDD** preconditioner leads to speedups of 1.6–1.9 mostly due to the high preconditioner construction cost and lack of pivoting to handle zero diagonal elements in the charge matrix (the zero values were substituted with unit values for preconditioner computation, thereby introducing errors). One idea that was explored to improve the performance of **FG-ILUDD** was to significantly increase the convergence of incomplete factorization by raising the number of sweeps performed. However, this approach proved to be far too costly, as any gains from decreasing the solver iterations were more than outweighed by the increase in preconditioner computation time.

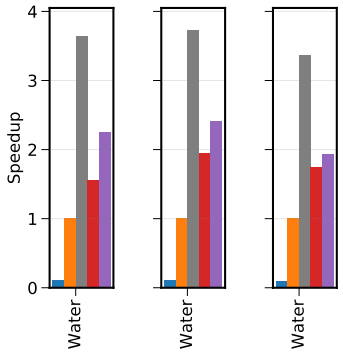
As highlighted in Table 7, the very poor condition numbers of the ACKS2 charge matrices pose a significant challenge, as the total number of solver iterations is still very large despite the limited success of the **ILUDD** and **SAI** preconditioners. This results in long solution times during simulations, leaving better preconditioners than the ones explored here to reduce the overall solve time desirable.



(a) QEq



(b) EE



(c) ACKS2

FIG. 3. *Speedups for different solvers for the four benchmark systems by the charge distribution method. Figures from left to right show results at convergence tolerance levels of* 10^{-6}, 10^{-10}, *and* 10^{-14}, *respectively. Speedups achieved are over the Jacobi preconditioned solver.*

5. Conclusions. We explored several shared memory preconditioning methods to improve the performance of iterative linear solvers in the computationally expensive charge distribution models. By applying sparsification techniques, carefully tuning the relevant preconditioner parameters, and adopting an efficient dynamic refactorization

scheme, we observed that incomplete factorization based methods (such as **ICDD**, **ILUDD**, and **FG-ILUDD**) and SAI based methods produce good quality preconditioners with relatively low costs. Performant implementation of these techniques on shared memory architectures results in solvers that significantly outperform the **Jacobi** baseline in time-to-solution across the majority of the charge models and molecular systems studied. As such, these techniques have been incorporated into PuReMD through an easy-to-use interface. Our future work on this topic will focus on developing GPU and distributed memory versions of the solvers explored in this study.

Appendix A. Supplemental data. In this appendix, supporting data are provided including the utilized force field parameters for simulation (Table 6), the condition numbers of the linear systems resulting from the charge models (Table 7), and solver parameters which produced the best mean solve times (Table 8).

TABLE 6
Force field parameter values utilized for simulations with the molecular systems in Table 1*, where the elements comprising the different systems use the identification from the periodic table.*

System	Element	γ_i	χ_i	η_i	σ_i	Λ
Bilayer	C	0.7631	5.9993	6.0000	-	-
and	H	0.8203	3.7248	9.6093	-	-
silica	N	1.0000	6.8287	7.2217	-	-
	O	1.0898	8.5000	8.3122	-	-
	P	1.0000	1.8292	7.2520	-	-
	Si	0.8925	4.6988	6.0000	-	-
PETN	C	0.8712	5.7254	6.9235	-	-
	H	0.8910	3.8446	1.0698	-	-
	N	0.8922	6.7424	6.2435	-	-
	O	0.8712	8.5000	7.1412	-	-
Water	H	0.8203	3.7248	9.6093	3.4114	-
	O	1.0898	8.5000	8.3122	0.9745	-
	-	-	-	-	-	548.6451

TABLE 7
Charge matrix condition numbers for the charge distribution models. Condition numbers derived from the eigenspectrum computed by the `LAPACKE_dgesvd` *function with Intel Math Kernel Library (MKL) version* 2018.1.163.

System	Method	$\dim(\mathbf{H})$	$\kappa(\mathbf{H}) = \frac{\sigma_{\max}}{\sigma_{\min}}$
Bilayer	QEq	56800	2.569E2
	EE	56801	3.166E2
PETN	QEq	48256	1.347E2
	EE	48257	2.117E2
Silica	QEq	72000	1.061E2
	EE	72001	2.315E2
Water	QEq	78480	6.008E1
	EE	78481	1.022E2
	ACKS2	156962	8.231E4

TABLE 8

Tuned preconditioned solver parameters. For selection, a parameter search was conducted in order to minimize mean solver time for the ***ICDD(t)+DS(d)****,* ***ILUDD(t)+DS(d)****,* ***FG-ICDD(t, s)+DS(d)****,* ***FG-ILUDD(t, s)+DS(d)****, and* ***SAI(τ)+DS(d)*** *preconditioned solvers. Parameters in the table are ordered from left to right as appearing in the above preconditioner names.*

Solver tolerance			10^{-6}	10^{-10}	10^{-14}
CM	Preconditioner	System			
QEq	**ICDD**	Bilayer	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.8)
		PETN	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
		Silica	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
		Water	(0.01, 0.6)	(0.0, 0.6)	(0.0, 0.6)
	FG-ICDD	Bilayer	(0.0, 4, 0.6)	(0.0, 3, 0.6)	(0.01, 3, 0.6)
		PETN	(0.0, 3, 0.6)	(0.0, 2, 0.6)	(0.01, 2, 0.6)
		Silica	(0.0, 2, 0.6)	(0.0, 2, 0.6)	(0.0, 3, 0.6)
		Water	(0.0, 3, 0.6)	(0.0, 3, 0.6)	(0.0, 4, 0.6)
	SAI	Bilayer	(0.05, 0.6)	(0.1, 1.0)	(0.15, 1.0)
		PETN	(0.1, 0.6)	(0.15, 1.0)	(0.15, 1.0)
		Silica	(0.075, 0.8)	(0.1, 0.8)	(0.1, 0.8)
		Water	(0.1, 0.6)	(0.15, 0.6)	(0.1, 0.6)
EE	**ILUDD**	Bilayer	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
		PETN	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
		Silca	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
		Water	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
	FG-ILUDD	Bilayer	(0.01, 3, 0.6)	(0.01, 3, 0.6)	(0.01, 4, 0.6)
		PETN	(0.1, 2, 0.6)	(0.0, 2, 0.6)	(0.0, 2, 0.6)
		Silica	(0.0, 3, 0.6)	(0.0, 3, 0.6)	(0.01, 4, 0.6)
		Water	(0.1, 2, 0.6)	(0.0, 2, 0.6)	(0.01, 2, 0.6)
	SAI	Bilayer	(0.15, 1.0)	(0.1, 1.0)	(0.15, 1.0)
		PETN	(0.125, 0.8)	(0.15, 0.6)	(0.125, 1.0)
		Silica	(0.1, 0.8)	(0.1, 0.8)	(0.1, 0.8)
		Water	(0.1, 0.6)	(0.1, 0.6)	(0.1, 0.6)
ACKS2	**ILUDD**	Water	(0.0, 0.6)	(0.0, 0.6)	(0.0, 0.6)
	FG-ILUDD	Water	(0.0, 2, 0.6)	(0.0, 2, 0.6)	(0.0, 2, 0.6)
	SAI	Water	(0.075, 0.6)	(0.15, 0.6)	(0.15, 0.6)

Acknowledgment. This research used computing resources of the High Performance Computing Center at Michigan State University.

REFERENCES

[1] H. M. AKTULGA, J. C. FOGARTY, S. A. PANDIT, AND A. Y. GRAMA, *Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques*, Parallel Comput., 38 (2012), pp. 245–259.

[2] H. M. AKTULGA, S. A. PANDIT, A. C. T. VAN DUIN, AND A. Y. GRAMA, *Reactive molecular dynamics: Numerical methods and algorithmic techniques*, SIAM J. Sci. Comput., 34 (2012), pp. C1–C23, https://doi.org/10.1137/100808599.

[3] H. M. AKTULGA, C. KNIGHT, P. COFFMAN, K. A. O'HEARN, T.-R. SHAN, AND W. JIANG, *Optimizing the performance of reactive molecular dynamics simulations for many-core architectures*, Int. J. High Performance Comput. Appl., 33 (2019), pp. 304–321, https://doi.org/10.1177/1094342017746221.

[4] J. R. ALLWRIGHT, R. BORDAWEKAR, P. D. CODDINGTON, K. DINCER, AND C. L. MARTIN, *A Comparison of Parallel Graph Coloring Algorithms*, Technical report SCCS-666, Northeast Parallel Architecture Center, Syracuse University, 1995.

[5] M. BENZI AND M. TUMA, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM J. Sci. Comput., 19 (1998), pp. 968–994, https://doi.org/10.1137/S1064827595294691.

[6] M. Benzi and M. Tûma, *A comparative study of sparse approximate inverse preconditioners*, Appl. Numer. Math., 30 (1999), pp. 305–340, https://doi.org/10.1016/S0168-9274(98)00118-4.

[7] E. F. F. Botta and A. van der Ploeg, *Renumbering strategies based on multi-level techniques combined with ILU-decompositions*, Zh. Vychisl. Mat. Mat. Fiz., 37 (1997), pp. 1294–1300 (in Russian); translation in Comput. Math. Math. Phys., 37 (1997), pp. 1252–1258.

[8] B. Carpentieri and M. Bollhöfer, *Symmetric inverse-based multilevel ILU preconditioning for solving dense complex non-Hermitian systems in electromagnetics*, Progr. Electromagn. Res., 128 (2012), pp. 55–74, https://doi.org/10.2528/PIER12041006.

[9] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, *Graph coloring algorithms for multi-core and massively multithreaded architectures*, Parallel Comput., 38 (2012), pp. 576–594, https://doi.org/10.1016/j.parco.2012.07.001.

[10] E. Chow and A. Patel, *Fine-grained parallel incomplete LU factorization*, SIAM J. Sci. Comput., 37 (2015), pp. C169–C193, https://doi.org/10.1137/140968896.

[11] W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz, D. M. Ferguson, D. C. Spellmeyer, et al., *A second generation force field for the simulation of proteins, nucleic acids, and organic molecules*, J. Am. Chem. Soc., 117 (1995), pp. 5179–5197.

[12] M. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM J. Sci. Comput., 18 (1997), pp. 838–853, https://doi.org/10.1137/S1064827594276552.

[13] T. A. Halgren and W. Damm, *Polarizable force fields*, Current Opinion in Structural Biology, 11 (2001), pp. 236–242.

[14] P. Hénon and Y. Saad, *A parallel multistage ILU factorization based on a hierarchical graph decomposition*, SIAM J. Sci. Comput., 28 (2006), pp. 2266–2293, https://doi.org/10.1137/040608258.

[15] P. Li and K. M. Merz, Jr., *Metal ion modeling using classical mechanics*, Chem. Rev., 117 (2017), pp. 1564–1686, https://doi.org/10.1021/acs.chemrev.6b00440.

[16] A. D. MacKerell, Jr., D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, et al., *All-atom empirical potential for molecular modeling and dynamics studies of proteins*, J. Phys. Chem. B, 102 (1998), pp. 3586–3616.

[17] W. J. Mortier, S. K. Ghosh, and S. Shankar, *Electronegativity-equalization method for the calculation of atomic charges in molecules*, J. Am. Chem. Soc., 108 (1986), pp. 4315–4320.

[18] W. J. Mortier, K. Van Genechten, and J. Gasteiger, *Electronegativity equalization: Application and parametrization*, J. Am. Chem. Soc., 107 (1985), pp. 829–835, https://doi.org/10.1021/ja00290a017.

[19] A. Nakano, *Parallel multilevel preconditioned conjugate-gradient approach to variable-charge molecular dynamics*, Comput. Phys. Comm., 104 (1997), pp. 59–69.

[20] M. Naumov, *Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU*, Technical report NVR-2012-003, NVIDA, 2012.

[21] M. Naumov, P. Castonguay, and J. Cohen, *Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU*, Technical report NVR-2015-001, NVIDA, 2015.

[22] R. A. Nistor, J. G. Polihronov, M. H. Müser, and N. J. Mosey, *A generalization of the charge equilibration method for nonmetallic materials*, J. Chem. Phys., 125 (2006), 094108.

[23] K. O'Hearn and H. Aktulga, *Towards fast scalable solvers for charge equilibration in molecular dynamics applications*, in Proceeding of the 2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), IEEE, 2017, pp. 9–16, https://doi.org/10.1109/ScalA.2016.006.

[24] X.-M. Pan and X.-Q. Sheng, *Sparse approximate inverse preconditioner for multiscale dynamic electromagnetic problems*, Radio Sci., 49 (2014), pp. 1041–1051, https://doi.org/10.1002/2014RS005387.

[25] S. Patel and C. L. Brooks III, *Fluctuating charge force fields: Recent developments and applications from small molecules to macromolecular biological systems*, Mol. Simul., 32 (2006), pp. 231–249.

[26] S. Plimpton, *Fast parallel algorithms for short-range molecular dynamics*, J. Comput. Phys., 117 (1995), pp. 1–19.

[27] A. K. Rappé, C. J. Casewit, K. S. Colwell, W. A. Goddard III, and W. M. Skiff, *UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations*, J. Am. Chem. Soc., 114 (1992), pp. 10024–10035.

[28] A. K. Rappe and W. A. Goddard III, *Charge equilibration for molecular dynamics simulations*, J. Phys. Chem., 95 (1991), pp. 3358–3363, https://doi.org/10.1021/j100161a070.

[29] S. W. Rick and S. J. Stuart, *Potentials and algorithms for incorporating polarizability in computer simulations*, Rev. Comput. Chem., 18 (2002), pp. 89–146.

[30] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numer. Linear Algebra Appl., 1 (1994), pp. 387–402.
[31] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, 2003 https://doi.org/10.1137/1.9780898718003.
[32] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869, https://doi.org/10.1137/0907058.
[33] T. P. Senftle, S. Hong, M. M. Islam, S. B. Kylasa, Y. Zheng, Y. K. Shin, C. Junkermeier, et al., *The ReaxFF reactive force-field: Development, applications and future directions*, Nature NPJ Computational Materials, 2 (2016), 15011.
[34] A. C. T. van Duin, S. Dasgupta, F. Lorant, and W. A. Goddard, *ReaxFF: A reactive force field for hydrocarbons*, J. Phys. Chem. A, 105 (2001), pp. 9396–9409.
[35] T. Verstraelen, P. W. Ayers, V. Van Speybroeck, and M. Waroquier, *ACKS2: Atom-condensed Kohn-Sham DFT approximated to second order*, J. Chem. Phys., 138 (2013), 074108, https://doi.org/10.1063/1.4791569.