

JAMD: Julia-Accelerated Molecular Dynamics

Nisha Chandramoorthy and Gerald (Jerry) Wang
Department of Mechanical Engineering

We present Julia-Accelerated Molecular Dynamics (JAMD), a molecular dynamics (MD) code in the Julia language. We show that JAMD compares very favorably with MATLAB-based MD codes, and is robust enough to enable rapid visualization of simple molecular systems. We demonstrate that there are several avenues to leverage parallelism in Julia, thereby accelerating simulations of interest in nanoscale engineering. We also discuss two algorithms that use prefix sums to perform N -body computations, relevant to MD simulations – one is based on recasting the Fast Multipole Method as a combination of parallel prefix and embarrassingly parallel operations and the other involves computation of potentials as a summation of low-rank functional approximations. We describe both methods in detail with special emphasis on exposing the prefix summations. These methods are generic and can be used anywhere; in particular, they find applications in our MD library when long-range Coulombic forces are involved in the simulation of ionic species or polar molecules. Finally, we suggest further areas of MD development that should be amenable to Julia implementation.

INTRODUCTION AND MOTIVATION

Statistical physics provides a powerful toolbox for analyzing nanoscale systems, with applications ranging from nanofluidic device engineering to chemical process development to biomolecule manipulation and drug design. However, when the number of degrees of freedom in the system becomes large (and the system Hamiltonian grows correspondingly unsightly), analytical approaches in statistical physics are often intractable. For example, the statistical properties of a dense fluid could in principle be obtained through analytical (or numerical) solution of the Born-Bogoliubov-Green-Kirkwood-Yvon (BBGKY) hierarchy of equations, but for real-world systems this is comically impractical due to the high dimensionality and non-linearity of the BBGKY equations [1]. This motivates the need for a computational approach that can generate a large number of system microstates for the purpose of sampling and thermodynamic analysis. For this purpose, molecular dynamics (MD) is a particularly popular and successful simulation algorithm. MD deterministically generates the *time evolution* of a system using information about atomistic kinematics and interatomic potentials. This is accomplished by numerically integrating Newton’s equations of motion for each constituent atom. The atomic trajectories produced by an MD simulation may then be analyzed to extract material or transport properties, or visualized to provide a general sense of the nanoscopic mechanics of the system.

As a very computationally intensive simulation method, we believe that Julia may be well suited for the development of an open-source MD code. To this end, we have developed Julia-Accelerated Molecular Dynamics (JAMD). In this work, we will provide an overview of the MD algorithm as implemented in JAMD (including the Fast Multipole Method algorithm, which can be used for rapid calculation of electrostatic forces) and also provide comparisons between JAMD and comparable MATLAB-

based MD codes. We will also discuss two approaches to MD parallelization – one based on an embarrassingly parallel approach and another based on parallel prefix – that may yield significant speedup for the simulation. Our work suggests that Julia is a strong candidate language for continued development of MD codes.

OVERVIEW OF THE MD ALGORITHM AND IMPLEMENTATION IN JAMD

We begin by supplying a brief summary of the MD algorithm, including technical background on the specific MD features implemented in JAMD.

Time-Integration of Newton’s Laws

The goal of MD simulation is to use knowledge about atomic positions and velocities, along with information about interatomic interactions, to predict positions and velocities in the future. In particular, consider an atom of mass m_i located at position \vec{r}_i , where the potential field is $U = U(\vec{r}_i)$. Then this atom obeys the equation of motion:

$$m_i \frac{d^2 \vec{r}_i}{dt^2} = - \frac{\partial U}{\partial \vec{r}_i} = \vec{f}_i \quad (1)$$

Here, \vec{f}_i is the force exerted on this atom.

Given a system of N interacting atoms, Eqn. (1) represents N coupled non-linear ODEs, which unsurprisingly cannot be solved analytically for non-trivial systems i.e. $N > 2$. Thus we must use a numerical integration technique to update the $6N$ values of atomic positions and velocities.

Velocity Verlet

Using knowledge of \vec{r}_i at time t (and all previous times), we can calculate the value of position at a time δt later, $\vec{r}_i(t + \delta t)$, as:

$$\vec{r}_i(t + \delta t) = \vec{r}_i(t) + \vec{v}_i(t)\delta t + \frac{1}{2}\vec{a}_i(t)\delta t^2 \quad (2)$$

where $\vec{a}_i = \vec{f}_i/m_i$. Analogously, we can update the velocity \vec{u}_i using:

$$\vec{u}_i(t + \delta t) = \vec{u}_i(t) + \frac{\vec{a}_i(t) + \vec{a}_i(t + \delta t)}{2}\delta t \quad (3)$$

This approach is known as the *velocity Verlet algorithm* [2, 3]. This algorithm is the standard basis for numerical time integration in popular research-grade molecular dynamics codes, such as LAMMPS [4]. We point out several important aspects of the velocity Verlet algorithm, which are generally considered to be advantages of the integration scheme:

1. This method is *self starting* and *explicit*. In other words, given initial positions and velocities (and interatomic potentials), we are immediately able to implement Eqns. (2) and (3) without specifying additional initial conditions. Moreover, at every timestep, we are able to calculate the LHS using known values for RHS quantities.
2. Velocity Verlet is a *symplectic integrator*. This means that as long as the forces in the system are conservative, the system Hamiltonian will not deviate substantially from its initial value in the long-time limit; in fact, the Hamiltonian will oscillate around its initial value.
3. The global errors in position and velocity are both $\mathcal{O}(\delta t^2)$.

The Velocity Verlet integrator is contained within the function *take_a_step*, within the module *MD_calc*.

Constraint Dynamics

The basic method described so far should, in principle, allow us to predict the time-evolution of a molecular system. However, the dynamics of real-world systems are often constrained in ways that are not directly or obviously related to Newton’s laws. This motivates the implementation of *constraint dynamics*. In particular, we are often interested in simulating systems that have a fixed temperature, which necessitates a *thermostat*.

If we wish to conduct a simulation in the *canonical ensemble* (in other words, we wish to hold constant the

particle number N , the system volume V , and the temperature T), it may be difficult to maintain a constant temperature since symplectic integration only preserves total system *energy*. The Berendsen thermostat [5] relies on rescaling velocities to achieve the desired temperature. In the simplest sense, this thermostat can be implemented by adjusting every atom’s velocity \vec{u} to a modified velocity \vec{u}' given by:

$$\vec{u}' = \vec{u}\sqrt{\frac{T_{\text{des}}}{T_{\text{inst}}}} \quad (4)$$

where T_{des} is the desired temperature and T_{inst} is the current system temperature. To prevent sharp jumps in temperature, it is common to introduce a relaxation parameter α (less than 1) to “smooth out” the thermostatting process:

$$\vec{u}' = \vec{u}\sqrt{\left(1 + \alpha\left(\frac{T_{\text{des}}}{T_{\text{inst}}} - 1\right)\right)} \quad (5)$$

The Berendsen thermostat is contained within the function *take_a_step*, within the module *MD_calc*.

Interatomic Potentials

We now discuss two interatomic potential models for fluids of interest in this work: The Lennard-Jones (LJ) potential (a good model for simple molecules interacting primarily through van der Waals effects), and the Coulomb potential (which governs electrostatic interactions such as those between ions in an ionic liquid).

The LJ potential [6] between two atoms separated by a distance r is given by:

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6\right] \quad (6)$$

The parameters ϵ and σ serve as characteristic energy and lengthscales (respectively) for the potential; in particular, ϵ is the magnitude of the minimum value of the LJ potential and $\sigma\sqrt[6]{2}$ is the interatomic spacing at which this minimum potential is achieved. This potential captures two key features of (non-electrostatic) intermolecular interactions:

1. Strong repulsion ($\mathcal{O}(r^{-12})$) for small separation distances ($r \rightarrow 0$), which is due to quantum-mechanical restrictions on heavily overlapping electron clouds.
2. Weak attraction ($\mathcal{O}(r^{-6})$) for large separation distances ($r \rightarrow \infty$), which is due to electrostatic interactions between induced dipoles. (This attraction is commonly referred to as the van der Waals effect.)

Because the LJ potential falls off very rapidly with r , it is possible to expedite MD calculations involving the LJ potential while maintaining high accuracy by neglecting the contributions of all atoms beyond a fixed cutoff radius R (typically $R \approx 3\sigma$).

The well known Coulomb potential at a distance r from a particle of charge Q is given by:

$$V(r) = \frac{Q}{4\pi\epsilon_0 r} \quad (7)$$

Here, ϵ_0 is the permittivity of free space ($\epsilon_0 = 8.854 \cdot 10^{-12} F \cdot m^{-1}$). Note that since the Coulomb potential falls off relatively slowly (especially compared to the LJ potential), the electrostatic force is considered to a *long-range* force and cannot be treated with a cutoff radius comparable to the LJ potential.

The choice of potential can be specified within the function `force_calculation`, within the module `MD_calc`.

Sampling of Thermodynamic Quantities from Classical Trajectories

For any system observable A , we can define the expectation value of A as the ensemble average:

$$\langle A \rangle = \int_{\Gamma} A(\{\vec{r}, \vec{p}\}) f(\{\vec{r}, \vec{p}\}) d\Gamma \quad (8)$$

where $\{\vec{r}, \vec{p}\}$ is the set of positions and momenta for all particles in the system and f is the probability density in phase-space Γ .

In equilibrium statistical mechanics, the probability density f can be expressed in terms of the system Hamiltonian $\mathcal{H}(\{\vec{r}, \vec{p}\})$ as:

$$f(\{\vec{r}, \vec{p}\}) = \frac{e^{-\beta\mathcal{H}(\{\vec{r}, \vec{p}\})}}{Q} \quad (9)$$

where β is the inverse temperature $(k_B T)^{-1}$. Here, Q refers to the partition function, which is defined as:

$$Q = \int_{\Gamma} e^{-\beta\mathcal{H}(\{\vec{r}, \vec{p}\})} d\Gamma \quad (10)$$

Discretely speaking, for an ensemble of M identically prepared systems running in parallel (each of which yields a measurement of the observable A), the expectation value of A is given by

$$\langle A \rangle = \frac{1}{M} \sum_{i=1}^M A_i \quad (11)$$

because an MD simulation, by construction, produces samples of $f(\{\vec{r}, \vec{p}\})$.

This formulation of expectation value is not particularly efficient for an MD simulation, which evolves a *single system* over time. To get a larger sample size without

running many MD simulations in parallel, we can make use of the *ergodic hypothesis*; in other words, we assume that successive snapshots of a thermodynamic quantity over time (for a single system) converge *in distribution* to that same quantity measured over a large number of parallel systems. Given the ergodic hypothesis, M represents the number of snapshots over time (as opposed to the number of identically prepared systems running in parallel).

In particular, there are two thermodynamic quantities of interest in this work. The density ρ in a volume V is calculated according to:

$$\rho = \frac{1}{V} \sum_{i \in V} m_i \quad (12)$$

In the absence of a fluid-center-of-mass flow velocity, the temperature T in a volume V is sampled using the Virial Theorem in three dimensions:

$$T = \frac{1}{3Nk_B} \sum_{i \in V} m_i |\vec{u}_i|^2 \quad (13)$$

REDUCED UNITS

To keep our numerical results as “clean” as possible, our Julia code reports quantities in *reduced units*. For a system of N LJ atoms with parameters σ and ϵ , we define:

1. The reduced particle density $N^* = N\sigma^3$.
2. The reduced temperature $T^* = \frac{k_B T}{\epsilon}$.
3. The reduced energy $E^* = \frac{E}{\epsilon}$.
4. The reduced time $t^* = \sqrt{\frac{\epsilon}{m\sigma^2}} t$.

Conversion between reduced and real units can be modified within the main JAMD program.

MD Algorithm Flow

The basic steps of our MD simulation, as implemented in the main JAMD program, are as follows:

1. Initialize the system by specifying all atomic positions and velocities, as well as all relevant interatomic potentials.

2. To advance to the next timestep, loop over each particle $i \in \{1, \dots, N\}$ and:
 - a) Calculate the force on particle i using the interatomic potentials. Note that this step is particularly computationally expensive, as it could impose computational cost up to $\mathcal{O}(N^2)$; in our work, we focus on embarrassingly parallel and parallel-prefix approaches to tackling this high $\mathcal{O}(N^2)$ cost.
 - b) Update the position and velocity of particle i using velocity Verlet. Rescaling of the velocities based on the Berendsen thermostat can be done after velocities are updated.
3. Save atomic positions and velocities. If the system is sufficiently equilibrated, one may perform thermodynamic sampling using these trajectories.
4. Return to Step 2 and repeat for as many timesteps as desired.
5. After simulation, trajectories can be post-processed for the purposes of statistical analysis or visualization.

JAMD VISUALIZATION

Using PyPlot.jl [7], we are able to visualize our simulated system using the trajectories produced by JAMD. A representative snapshot of such a visualization is shown in Figure 1. Visualization options can be specified within the main JAMD program.

COMPARISON BETWEEN JAMD AND MATLAB MD

We performed MD simulations of eight representative nanofluidic systems, detailed in Table I, using both JAMD as well as a highly optimized MATLAB MD code. Each fluid is either a liquid or vapor phase of argon, which is a noble gas and is thus well approximated by the LJ potential.

We observe a very clear speedup of JAMD relative to MATLAB. These speedups are of at least one order of magnitude for all benchmark systems. We believe that Julia’s strong performance is primarily due to our code optimizations, which involve minimal memory allocation.

EMBARRASSING PARALLELIZATION AND SHARED ARRAYS

Because force calculations are the most costly step in the MD algorithm, and since the force between any two

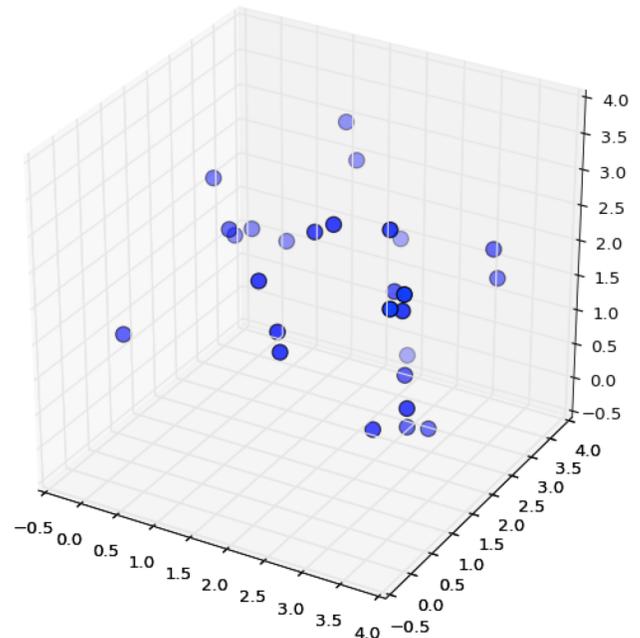


FIG. 1. Snapshot showing 27 atoms of a dilute phase of argon.

Name	ρ [σ^{-3}]	# Atoms	MATLAB/JAMD Ratio
Tiny Vapor	0.25	8	41.34
Small Vapor	0.25	27	11.15
Medium Vapor	0.25	125	14.27
Large Vapor	0.25	512	13.35
Tiny Liquid	0.95	8	38.27
Small Liquid	0.95	27	12.67
Medium Liquid	0.95	125	15.89
Large Liquid	0.95	512	14.28

TABLE I. Timing results for eight benchmark nanofluidic systems. All simulations were carried out for a (dimensionless) time of 5 units. The ratios were calculated by dividing the time taken for the MATLAB MD code to complete the simulation by the time taken for JAMD to complete the simulation. All ratios were calculated as the average over three trial runs of the simulation code.

atoms is independent of any other atom, one simple yet effective approach to parallelization is to split the process of calculating interatomic forces over multiple processors. By taking advantage of the `SharedArray` type in Julia, we can divide the work of calculating entries in the interatomic force matrix. This is implemented for the LJ potential within the function `force_calculation_parallel`, within the module `MD_calc`. Our speedup results are reported in Table II. We note that communication latency can be a very costly component of this parallelization scheme, and so in practice this approach does not yield speedups for small systems. For large systems (containing at least $\mathcal{O}(10^3)$ atoms), this method should yield

# Processors	Relative Time
1	1
2	0.71
3	0.52
4	0.43

TABLE II. Speedup results for an embarrassingly parallel calculation of interatomic forces. All simulations were carried out for a (dimensionless) time of 5 units on the “Large Liquid” system described in Table I. All ratios were calculated relative to the run-time of JAMD on a single processor, using the average over three trial runs of the simulation code.

some amount of parallel speedup.

PARALLEL PREFIX AND THE FAST MULTIPOLE METHOD

The Fast Multipole Method is an algorithm that computes a dense matrix-vector product in $\mathcal{O}(N)$ time. The entries of the matrix correspond to gravitational/electrostatic interactions in N -body problems, usually. In general, the entries represent the evaluations of the Green’s function of the Laplace/Helmholtz operator or more generally, evaluations of interaction potentials that fall rapidly such as $1/r$ or even exponentially (in Fast Gauss Transforms). The algorithm calculates an approximate local expansion (Taylor expansion) that can be evaluated at target points to obtain the interactions due to all the source points. This report addresses the question of how the algorithm can be viewed as a cumulative summation of certain function representations.

The Algorithms

In this section, we discuss the algorithms in question, namely:

1. FMM
2. Neighbor-exclusive Cumulative Sum in Serial and in Parallel.

The traditional FMM algorithm in 2D

The computational box is assumed to be split into 4 (2 and 8 in case of 1D and 3D respectively) child boxes recursively, l times, to obtain a quadtree structure. The number of levels l is decided by the desire to not have more than a certain number of particles per box at the finest level.

Now, the FMM computes a Taylor series approximation centered about each box at the finest level, which

can be evaluated at particles in each box (assuming target points are source points). This Taylor series approximation pertains to the interactions felt by the particles in that box, due to all the particles outside the near neighbors of the box (near neighbors are the boxes that share an edge with the box, at the same level – maximum of 9, including the box itself in case of 2D). The interactions with the near neighbors are computed directly. This, in brief, is how the FMM arrives at an approximate local expansion, through a series of *projection* and *interpolation* steps, following [8]:

1. At the finest level, multipole expansions (M) are calculated around each box center. These are coefficients of an approximate function that has singularities near the origin (box center) but is valid with high accuracy in the far field ($z > 2R$, where z is the evaluation point in the complex plane and R is the box width at the finest level). The approximate function refers to ϕz :

$$\phi(z) = a_0 \log(z) + \sum_{i=1}^s a_i / z^i \quad (14)$$

Now, M refers to the vector $\{a_0, \dots, a_s\}$, computed as follows:

$$a_0 = \sum_{i=1}^m q_i \quad (15)$$

$$a_k = \sum_{i=1}^m \frac{-q_i z_i}{k} \quad (16)$$

where q_i are the charges in the box and s is chosen as approximately $\log_2 \epsilon$, for a relative accuracy (with respect to a_0) of ϵ .

2. M at every level, which is centered around the box center is translated to be centered around the parent box’s center – this is done recursively starting at the finest level. We refer to this step as the M-M translation step with the operator that shifts the origin of the multipole expansion from child to parent box, being referred to as, $\tau_{MM}(A = \{a_0, \dots, a_s\} \rightarrow B = \{b_0, \dots, b_s\})$ where $b_0 = a_0$ and b_l is given by:

$$b_l = -\frac{a_0 z_0^l}{l} + \sum_{k=1}^s a_k z_0^{l-k} \binom{l-1}{k-1} \quad (17)$$

These can be derived from binomial series. The important point to note here is that τ_{MM} is a linear form in a_0 and a_k . This fact will be exploited in a future section.

In the above, z_0 refers to the center of the child box, with the center of the parent box as the origin. We

must remember that the vector M of a box gives the expansion coefficients for evaluation outside the box, of the potential due to particles inside the box.

- In this step, the translation operator $\tau_{ML}(A = \{a_0, \dots, a_s\} \rightarrow B = \{b_0, \dots, b_s\})$ is applied at every level, starting at level 0, to shift the multipole expansion coefficients A of the j -th box to local expansion coefficients of the i -th box, B . Here, j refers to the indices of the interaction list of i (children of the near neighbors of i 's parent that aren't neighbors of i , with a maximum of 27 per box) and this operation is repeated for all i , at a given level. Now, we can see that the interactions that have yet to be counted in, at this stage and at every level, can be obtained shifting the local expansion coefficients of the parent of a box to the child, at every level. This allows us to get the local expansion coefficients. We refer to this translation using the operator $\tau_{LL}(A = \{a_0, \dots, a_s\} \rightarrow B = \{b_0, \dots, b_s\})$. τ_{ML} and τ_{LL} can be obtained from Lemmas 2.2 and 2.3 in [8] and various other FMM literature. The point we must note is that these operators are linear in the input vector coefficients and this fact can be used to parallelize the translation operations.

Cumulative Sums

Given a vector x , the inclusive cumulative/prefix sum is the vector y given by: $y[i] = \sum_{j=1}^i x[j]$. The inclusive suffix sum refers to $y[i] = \sum_{j=i}^N x[j]$. The exclusive version of these sum types don't count the element itself i.e. exclusive prefix would be $y[i] = \sum_{j=1}^{i-1} x[j]$. Now, the exclusive prefix-suffix refers to the sum of all the elements except the current element. (From here on, we refer to the vectors x and y to indicate the input and output vectors, respectively, of cumulative sums of any kind).

The algorithms that perform the exclusive prefix-suffix directly are trivial and take $\mathcal{O}(2N)$ operations. Note that they must be performed as a prefix operation "added" together with a suffix and not as the current element "subtracted" from the total sum – this is because "subtraction" may not be a backward stable operation. For instance, consider the case of "add" referring to the translation τ_{ML} . In this case, we will be "subtracting" multipole expansion evaluations within the box and that means evaluation at singularities.

Essentially, the "sum" here makes the prefix algorithm a general purpose polymorphism – it could refer to any binary operation. A parallel version of the prefix basically works by performing hierarchical sums in $\mathcal{O}(\log N)$

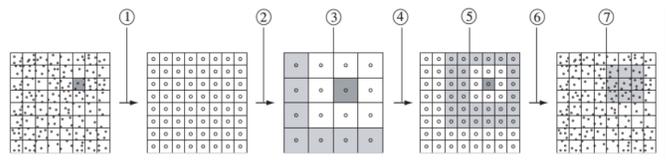


FIG. 2. Pictorial representation of hierarchical FMM in 2D, from [9].

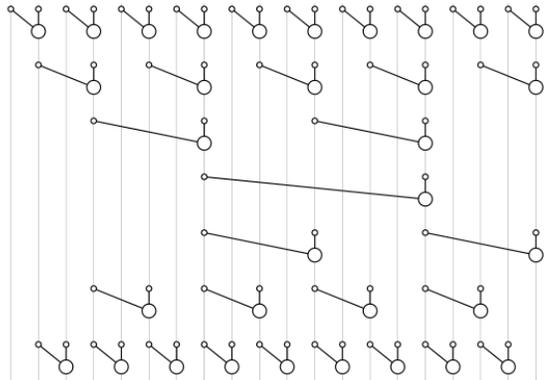


FIG. 3. Inverse spanning tree, from course notes.

time, in a spanning tree followed by an inverse spanning tree (the order of operations is shown in Figure 3).

FMM in 2D as Parallel Prefix + Embarrassingly Parallel Operations

In this section, we see how we can retain the hierarchical structure of the non-adaptive FMM algorithm and implement it as a combination of parallel prefixes and embarrassingly parallel operations. Let us recast the above mentioned FMM algorithm as follows:

- The computation of M 's at the finest level is an embarrassingly parallel operation. This should be an $\mathcal{O}(N/p)$ operation on p processors.
- The Upward Pass: For each box i at the finest level, we construct a "path" that represents an array of boxes spanning all the levels such that $path[i+1]$ is the parent of $path[i]$. Populating the multipole expansion coefficients of $path$ is a cumulative sum, which can be implemented as a parallel prefix! That is, the sum operation refers to the τ_{MM} operator defined above and the $x[i]$ refers to the M vector of a box at the $l-i+1$ th level. Here is a code snippet, in Julia, that constructs $path$ and calls the prefix function:

```
for i= 1:nbox_fineest_level
  pathind[1,i] = i
```

```

path[1,i] = tree[end,i]
for j = 1:levels-1
  t = pathind[j,i]
  pathind[j+1,i] =
  get_parent(t,levels-j+1)
  path[j+1,i] = tree[end-j,pathind[j+1,i]]
end

FMM.prefix!(path[:,i], MM_prefix)
for j = 1:levels-1
  [tree[end-j,pathind[j+1,i]].M[k] +=
  path[j+1,i].M[k]
  for k=1:s]
end

end

```

3. The Downward Pass: The downward pass can be thought of as applying the parallel prefix on the *path* vectors, this time constructed in reverse order. We need to ensure that the interaction list calculations (which are embarrassingly parallel) enter the *x* array, which now consists of local expansion coefficients. Note that we need not wait for all the IL computations to finish but only that they are in the prefix algorithm, as they are added: this is possible because τ_{ML} and τ_{LL} (which is the “sum” part of parallel prefix) are linear in their input.

As a note on the implementation of the above in Julia, we could use a `SharedArray` to store the two types of functional coefficients: Multipole and Local. The worker processes need to be synced at the end of the upward and downward passes.

In summary, we look at the FMM computations in the upward and downward passes as a set of embarrassingly parallel operations. Each of these operations is further a parallel prefix, with the relevant translation operator being the “sum” and the vector *x* being either the multipole or local expansion arrays of boxes across different levels. As a final note, we must acknowledge that this approach makes the abstractions in FMM clearer and gives a speedup over regular parallel FMM code when *N* is large enough for $\log N$ to be significant.

FMM as Neighborhood Exclusive Cumsum

In this section, we will examine how the computation of function approximations at the finest level fits the abstraction of serial neighborhood exclusive prefix-suffix – this idea is outlined in [10]. Let’s begin by recalling that at the end of the FMM algorithm, we end up with a local function approximation that we evaluate inside each box, at the finest level, to obtain the interactions due to particles in all other boxes at the finest level, excluding

the near neighbors of the box. Now, in a neighborhood exclusive cumsum:

$$y = \begin{bmatrix} . & . & 1 & 1 & 1 & 1 & 1 & 1 \\ . & . & . & 1 & 1 & 1 & 1 & 1 \\ 1 & . & . & . & 1 & 1 & 1 & 1 \\ 1 & 1 & . & . & . & 1 & 1 & 1 \\ 1 & 1 & 1 & . & . & . & 1 & 1 \\ 1 & 1 & 1 & 1 & . & . & . & 1 \\ 1 & 1 & 1 & 1 & 1 & . & . & . \\ 1 & 1 & 1 & 1 & 1 & 1 & . & . \end{bmatrix} x$$

This gives the idea that if the *x* vector refers to a functional approximation of the influence of the boxes at the finest level, then the local approximation at the finest level can be thought of as the *y* vector. This leaves us to find out what exactly *x* can be.

x being Multipole Expansion coefficients

Now, let us consider *x* to be the *M* vectors of the boxes at the finest level. Then, the “sum” operation required, in order for $y[i]$ to hold local expansion coefficients of the *i*th box, must be the translation operator τ_{ML} . At this point, two remarks are in order:

1. In every “add” operation defined as above, we must keep in mind that the center of the local expansion we would like is the center of the box *i*. Therefore, we need a total of `nbox` such cumsums, to get the correct local expansions in all the `nbox` boxes.
2. The neighborhood excluded must be all boxes within $2R$ and not just the near neighbors. This is because the M-L translation converges inside a box of width *R* only when the source box is farther than $2R$ away i.e. the singularities of the multipole expansion are well-separated from the target points. Now, therefore, the number of interactions that must be computed directly has increased to include the previously defined interaction list as well.

Given the cost of this approach, we may be able to do better if we chose a different *x*. This calls for an alternate representation of the approximation to the potential function, an idea proposed in [10].

Projection using SVD

We have thus far utilized two types of functional approximations to represent the potential – multipole and local expansions. In this section, we shall explore the possibility of using other functional approximations of the potential. In order to overcome the limitations of

our previous approach, we need this $x[i]$ to be centered around the box center of i and be valid for evaluation inside box i . We can start over by considering the original N -body problem:

$$\underbrace{\begin{bmatrix} \phi(y_1) \\ \phi(y_2) \\ \cdot \\ \cdot \\ \cdot \\ \phi(y_n) \end{bmatrix}}_{\Phi} = \underbrace{\begin{bmatrix} G(x_1, y_1) & \dots & G(x_n, y_1) \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ G(x_1, y_n) & \dots & G(x_n, y_n) \end{bmatrix}}_{\mathcal{G}} \underbrace{\begin{bmatrix} q_1 \\ q_2 \\ \cdot \\ \cdot \\ \cdot \\ q_n \end{bmatrix}}_{\mathcal{Q}} \quad (18)$$

where, G is the Green's function representation of the potential (for instance, it could be the Green's function of the Laplace operator in 2D, $G(x, y) = \log \|x - y\|$). Now, say that all the target points $\{y_i\}$ are well-separated from the source points $\{x_i\}$. Then, the set of charges $\{q_i\}$ can be *projected* onto a center $q = \sum_{i=1}^N q_i$ at x (say, geometric center of $\{x_i\}$) and the interactions at the target points will approximately be the interactions with the center. That is:

$$\begin{bmatrix} \phi(y_1) \\ \phi(y_2) \\ \cdot \\ \cdot \\ \cdot \\ \phi(y_n) \end{bmatrix} \approx \begin{bmatrix} q \times G(x, y_1) \\ \dots \\ \dots \\ \dots \\ \dots \\ q \times G(x, y_n) \end{bmatrix} = \underbrace{\begin{bmatrix} G(x, y_1) \\ \dots \\ \dots \\ \dots \\ \dots \\ G(x, y_n) \end{bmatrix}}_{\mathcal{G}_1} \times \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} \times \begin{bmatrix} q_1 \\ q_2 \\ \cdot \\ \cdot \\ \cdot \\ q_n \end{bmatrix}$$

In essence, by the above *projection* of charges onto a single charge, we have replaced the matrix \mathcal{G} with its rank-1 approximation, \mathcal{G}_1 . Similarly, we could consider a rank- p ($p \ll n$) projection of \mathcal{G} to get a better functional approximation of the potential. That is, we could approximate ϕ by considering a rank- p projection of \mathcal{G} , \mathcal{G}_p such that:

$$\begin{bmatrix} \phi(y_1) \\ \phi(y_2) \\ \cdot \\ \cdot \\ \cdot \\ \phi(y_n) \end{bmatrix} \approx \begin{bmatrix} G(\tilde{x}_1, y_1) & \dots & G(\tilde{x}_p, y_1) \\ \dots & \dots & \dots \\ G(\tilde{x}_1, y_n) & \dots & G(\tilde{x}_p, y_n) \end{bmatrix} \times \begin{bmatrix} \tilde{q}_1 \\ \tilde{q}_2 \\ \dots \\ \tilde{q}_p \end{bmatrix}$$

We know that the best rank- p approximation of a matrix can be obtained from the SVD of the matrix as: $\sum_{i=1}^p \sigma_i u_i v_i^T$, where the SVD of G is given by $G = U \Sigma V^T$ and u_i and v_i are the columns of U and V respectively. But, performing an SVD is as expensive as or more expensive than a direct matrix-vector multiplication of the original problem. Following [10], we use instead p Lanczos iterations ($\mathcal{O}(Np^2)$) to pick p virtual charges and positions, thus giving us a good approximation to evaluate at any target point not within R of the box. This method gives us an accuracy equivalent to the $s = 2p$ term multipole expansion. When $p = 1$, we have the COM projection method just discussed above. We detail this method in the next section.

The Lanczos Projection Method

This section is about how we use a well-known eigenvalue algorithm, the Lanczos iteration method, to find a low-rank functional approximation for our required potential function. The algorithm itself is outlined in [10] and various other literature. Our focus here will be to explain how and why it works in turning our N -body problem into a direct summation of certain functional approximations.

We know that p iterations of Lanczos algorithm on a Hermitian matrix A results in a $p \times p$ tridiagonal matrix T_{pp} . In constructing this matrix, the Lanczos method computes an orthogonal basis for the Krylov subspace $\{b, Ab, A^2b, A^3b \dots A^{p-1}b\}$. Here, b is any arbitrary normalized vector. It is also known that the eigen values of T_{pp} provide a close approximation to the p largest eigen values of A [11].

Therefore, the questions that need to be answered are:

1. How do we choose A and b for our potential approximation problem?
2. How do the eigenvalues and eigenvectors of T_{pp} help us in writing our potential as a direct sum?

Gauss Quadrature And Lanczos

To answer the first question posed above, let us look at the continuous (integral) analog of our N -body problem in Eqn. (18):

$$A\sigma(x) = \int_{\Omega} \sigma(y)G(x, y)dy$$

where $A : \mathbb{L}^2(\Omega) \rightarrow \mathbb{L}^2(\Omega)$ is an integral operator that acts on the function $\sigma(y)$ that represents the continuous charge distribution (whose discrete analog was \mathcal{Q}) to produce the required potential function at target point x .

To evaluate the above integral, one could use the p -point Gauss quadrature rule, which is exact for polynomials of degree up to $2p - 1$ (Theorem 37.3, [11]). We have:

$$A\sigma(x) \approx \sum_{i=1}^p w_i \sigma(y_i) G(x, y_i) \quad (19)$$

Note that we recover our *exact* discrete problem for the set of weights $\{w_i \sigma(y_i)\} \equiv \{q_i\}$, the set of nodes $\{y_i\} \equiv \{x_i\}$ (where x_i are the original locations of the source charges), and $p = n$.

We now aim to compute the quadrature nodes and weights for $p \ll n$, through the Lanczos algorithm. Therefore, we need to answer our first question (1) on choosing the Hermitian matrix A and an appropriate vector b . In order to do so, let us recall from Eqn. (19) above that:

$$\phi(x) = \sum_{i=1}^n q_i G(x, x_i) \quad (20)$$

Focussing our attention on 1D or 3D problems, $G(x, x_i) = 1/||x - x_i||$. Define D^{-1} to be a diagonal matrix such that $D_{ii}^{-1} = \text{sgn}(q_i)/x_i$. Then, $D^{-1}q = \{\phi(0, x_i)\} = \Psi(\text{say})$. Therefore, $\Psi \in \{q, Dq, D^2q, \dots, D^{n-1}q\}$. Hence, in our Lanczos Algorithm, A needs to be D such that $D_{ii} = \text{sgn}(q_i)x_i$, where x_i are the original locations. b needs to be a normalized vector of the absolute values of the original charges q_i . But, when D is as defined, according to Theorem 37.4 of [11], we have, y_i : set of nodes of Gauss quadrature are the eigenvalues of the tridiagonal matrix T_{pp} and the weights, which are equal to the new set of charges ($\tilde{q}_i = w_i q_i$) can be obtained from the eigenvectors of T_{pp} as:

$$\tilde{q}_i = 2 \cdot |v_{i1}|^2$$

where, v_{i1} is the first element of the i -th eigenvector of T_{pp} . (Notice that \tilde{q}_i are always positive and the signs have been preserved in D instead.)

To summarize, $\phi(x)$ has been expressed as a sum of p orthogonal Legendre polynomials, through this Lanczos procedure. In the FMM, the local expansion is a polynomial of degree p but here, we have achieved the best possible accuracy with polynomials of degree up to $2p - 1$, with only p charges and positions – this was possible because of the connection of the virtual charges and positions to Gaussian quadrature weights and nodes, respectively.

Projections and Evaluations As Cumsums

We are now only left with the second question: how can the above-mentioned Lanczos procedure be used to treat our problem as a cumulative sum?

The idea is that given any box center i as the origin, we use the Lanczos algorithm to obtain the p virtual charges



FIG. 4. Simulation box in 1D: The open circles (red) are the charges and the closed circles (black) mark off box widths. Far field interactions are those that act from beyond one box width – these are added to the near neighbor interactions computed directly.

and their positions that effectively describe the interactions of the charges in the far field (far field is defined as $|x - x_i| > R$, where R is one box width). The near-neighbor interactions are computed directly for every box, in the final step, like in the FMM.

However, the reason this algorithm can be looked at as a cumulative sum is that the Lanczos procedure is not performed for all the particles in the far field of every box but rather in increments. That is, we compute the far-field interactions in two loops of size equal to the number of boxes: the prefix loop and the suffix loop.

In the prefix loop, we traverse the boxes from left to right. At the end of the i -th iteration of the prefix loop, we have approximated the cumulative effect of all the particles q_j that are distributed in boxes $\{1 \dots i\}$ at positions x_j into \tilde{q}_k^i ($1 \leq k \leq p$). That is:

$$\phi(x) = \sum_j q_j G(x, x_j) \approx \sum_{k=1}^p \tilde{q}_k^i G(x, \tilde{x}_k^i) \quad |x - x_c^i| > R \quad (21)$$

where R is the box width and x_c^i is the center of the i -th box. This reduction can be performed by doing p Lanczos iterations with x_c^i as the origin. This functional approximation is evaluated at all charge positions in the box $i + 2$.

Now, i is updated to the next box and the old virtual charges obtained in the previous step, which are positioned within the set of boxes $\{1 \dots i\}$, are retained as $l \leq p$ “original” charges. To these, the set of charges in the cell i , N_i , are added and a Lanczos projection is performed on the diagonal matrix containing the positions of the $p + N_i$ charges. This is done to reduce the number of virtual charges and positions to p , with the box center of $i + 1$ as the origin. This new set of charges and positions, $\{\tilde{q}^{i+1}\}$ can now be evaluated at target points in the box $i + 3$.

Note that if the total number of charges is $\leq p$, the virtual charges and positions are the actual ones. Now the same procedure is performed right to left, in the suffix loop. The final piece of the puzzle is the near-neighbor interactions, which are added directly at each target position.

Here is the pseudocode for the prefix loop:

$$q_v^0 = 0, x_v^0 = 0$$

for all $i \in \text{box } \{1 \dots N_{\text{box}}\}$ **do**

Add charges and positions in box i to q_v^{i-1} and x_v^{i-1} .

Perform Lanczos projection, if total size of $q_v > p$, and obtain q_v^i and x_v^i .

Evaluate at target positions in box $i + 2$ using Eqn. (21).

end for

The suffix loop is identical except from right to left. To summarize, the “+” operation in the `prefix` loop consists of a Lanczos projection followed by potential evaluation at the target points in box $i + 2$. Now, if we didn’t use the old virtual charges at every iteration but performed a Lanczos projection of the original charges up to box i at every iteration, then the algorithm will be much more expensive ($\mathcal{O}(NN_{box}p^2)$) but now the N_{box} calls to the `prefix` function can be run in parallel. However, the accuracy of recycling the old virtual charges needs to be analyzed! One obvious fact to note is that in keeping the number of virtual charges p constant, we are seeking better approximations for lower values of i : that is, the near parts of the far field are indeed approximated better, thereby possibly justifying this algorithm.

CONCLUSIONS AND FUTURE DIRECTIONS

We have developed JAMD, an MD code in the Julia language. JAMD demonstrates very strong performance against comparable MATLAB-based codes for a variety of benchmark systems used in nanoscale engineering. Moreover, we have identified and discussed in detail several major approaches for force-calculation parallelization based on embarrassingly parallel operations as well as parallel prefix, which both show good promise for the future parallelization of other interatomic potentials.

We have discussed the correspondence between the Fast Multipole Method and cumulative sums of function approximations. In our first approach, we retained the hierarchical structure of the FMM and exposed the underlying parallel prefix sums in the projection and interpolation. In our second, we try to reproduce the effect of the FMM using serial cumulative summations that exclude neighbor interactions.

We intend to carry out further development of JAMD, including the following additions and areas of further study:

1. Implementation of more-complex and multi-body interatomic potentials, such as the three-body Stillinger-Weber potential [12]. These potentials should also be amenable to embarrassing parallelization.
2. Creation of additional visualization tools using PyPlot.jl.
3. Integration of the Fast Multipole Method into force calculations involving electrostatic forces, as well as support for systems featuring multiple types of interatomic interactions.
4. Robust support for multi-component systems, including a built-in library that supports different atomic and molecular species.
5. Development of function libraries that are commonly used in advanced MD codes. These would include automatic geometry initialization for complex geometries, alternative and additional constraint dynamics methods (e.g. the Nosé-Hoover thermostat [13] or the SHAKE geometry-constraint algorithm for large molecules [14]), as well as application-specific thermodynamic output formats.

Based on the present work, we believe that Julia provides the “best of both worlds” for MD simulations, combining the raw speed of a low-level language with the convenience and abstraction of a high-level language. JAMD is a promising beginning for the further development of Julia-based MD codes.

-
- [1] M. Kardar, *Statistical Physics of Particles* (Cambridge University Press, 2007).
 - [2] L. Verlet, *Physical Review*, **159**, 98 (1967).
 - [3] L. Verlet, *Physical Review*, **165**, 201 (1968).
 - [4] S. Plimpton, *Journal of Computational Physics*, **117**, 1 (1995).
 - [5] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak, *The Journal of Chemical Physics*, **81**, 3684 (1984).
 - [6] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford University Press, 1989).
 - [7] S. G. Johnson, “Pyplot.jl”.
 - [8] L. Greengard and W. D. Gropp, *Computers Math. Applic.*, **20** (1990).
 - [9] T. K. Sheel, *Progress In Electromagnetics Research B*, **27**, 327 (2011).
 - [10] A. Edelman and P.-O. Persson, (2004).
 - [11] L. N. Trefethen and D. Bau III, *Numerical Linear Algebra* (SIAM, 1997).
 - [12] F. H. Stillinger and T. A. Weber, *Physical Review B*, **31**, 5262 (1985).
 - [13] W. G. Hoover, *Physical Review A*, **31**, 1695 (1985).
 - [14] J.-P. Ryckaert, G. Ciccotti, and H. J. Berendsen, *Journal of Computational Physics*, **23**, 327 (1977), ISSN 0021-9991.