Computer Physics Communications 260 (2021) 107742

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Cyclically parallelized treecode for fast computations of electrostatic interactions on molecular surfaces[☆]

Jiahui Chen [a], Weihua Geng [b,*], Daniel R. Reynolds [b]

[a] Department of Mathematics, Michigan State University, East Lansing, MI 48823, USA
[b] Department of Mathematics, Southern Methodist University, Dallas, TX 75275, USA

ARTICLE INFO

Article history:
Received 3 July 2018
Received in revised form 29 September 2020
Accepted 16 November 2020
Available online 23 November 2020

Keywords:
Treecode
Electrostatics
Parallel computing
MPI
Load balancing

ABSTRACT

We study the parallelization of a flexible order Cartesian treecode algorithm for evaluating electrostatic potentials of charged particle systems in which N particles are located on the molecular surfaces of biomolecules such as proteins. When the well-separated condition is satisfied, the treecode algorithm uses a far-field Taylor expansion to compute $O(N \log N)$ particle–cluster interactions to replace the $O(N^2)$ particle–particle interactions. The algorithm is implemented using the Message Passing Interface (MPI) standard by creating identical tree structures in the memory of each task for concurrent computing. We design a cyclic order scheme to uniformly distribute spatially-closed target particles to all available tasks, which significantly improves parallel load balancing. We also investigate the parallel efficiency subject to treecode parameters such as Taylor expansion order p, maximum particles per leaf N_0, and maximum acceptance criterion θ. This cyclically parallelized treecode can solve interactions among up to tens of millions of particles. However, if the problem size exceeds the memory limit of each task, a scalable domain decomposition (DD) parallelized treecode using an orthogonal recursive bisection (ORB) tree can be used instead In addition to efficiently computing the N-body problem of charged particles, our approach can potentially accelerate GMRES iterations for solving the boundary integral Poisson–Boltzmann equation.

1. Introduction

Pairwise interactions among N particles/objects are ubiquitous. These interactions arise in various forms in applications as varied as astrophysics [1], fluid dynamics [2], statistical machine learning [3], electrodynamics [4,5], low-frequency scattering [6], and linear elasticity [7]. Since brute-force computation of these interactions has $O(N^2)$ complexity, which is prohibitively expensive when N is large, numerous fast algorithms have been developed to reduce the computational cost. These algorithms can be categorized into mesh-based methods [8], and tree-based methods [1,4,5,7,9,10]. Tree-based methods have shown tremendous promise in both efficiency and accuracy, and can be further categorized into roughly the particle–cell method [1,9] and the cell–cell method [4,10]. In tree-based methods, particles are partitioned into a hierarchy of clusters having a tree structure, allowing the pairwise particle–particle interactions to be calculated more efficiently. For example, in the treecode method [5,9] particle–particle interactions are replaced by particle–cluster interactions; these can then be evaluated using a far-field multipole expansion when certain criteria are satisfied. Similarly, the fast multipole method (FMM) [4,7,11–15] is a more elaborate procedure that evaluates cluster–cluster interactions using both far-field and near-field expansions.

In principle, for a given order of expansion, the treecode algorithm requires $O(N \log N)$ execution time and the FMM requires $O(N)$ execution time. However, the performance of an algorithm is also determined by the size of pre-factor, memory usage, coding complexity, and parallelizability. In practice, several factors can affect the observed performance, including the number of levels in the tree, the homogeneity or sparseness of the particle distribution, and the cache size of the computer. Optimizing these methods and extending them to new applications are active areas of research. Recently, increased attention has been given to the parallelization of these fast algorithms in response to the rapid development of multicore computers. These efforts have included parallel algorithms for treecode [16–20] and FMM [11,13,14,21–24], as well as their implementations on GPUs [25,26].

Recently, a flexible order Cartesian treecode algorithm [5,27] was developed for efficiently computing N-body interactions based on Barnes–Hut's tree structure [9]. This treecode algorithm has the following important features, which make the challenging task of studying dynamics on large-scale problems possible.
(1) Treecode uses particle–cluster interactions to replace the particle–particle interactions for far-field interactions, thereby

[☆] The review of this paper was arranged by Prof. David W. Walker.
[*] Corresponding author.
E-mail address: wgeng@smu.edu (W. Geng).

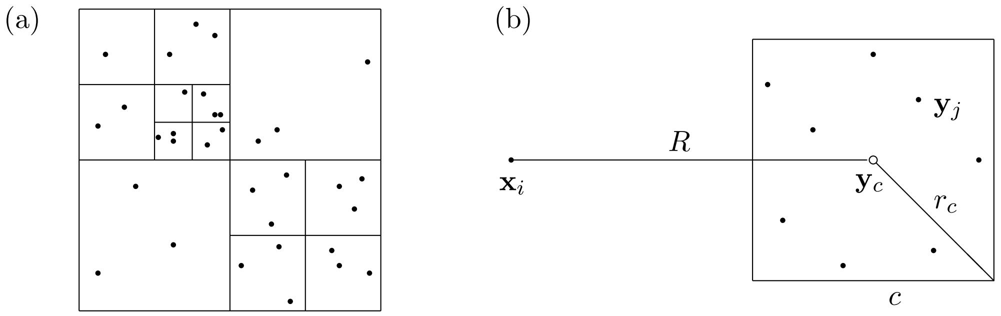


Fig. 1. Details of treecode; (a) tree structure of particle clusters; (b) particle–cluster interaction between particle $\mathbf{x}_i$ and cluster $c = \{\mathbf{y}_j\}$; $\mathbf{y}_c$ is the cluster center, R is the particle–cluster distance, and r_c is the cluster radius.

significantly reducing the computational cost from $O(N^2)$ to $O(N \log N)$.
(2) In treecode, the far-field expansion uses a Cartesian Taylor expansion, with Taylor coefficients computed using effective recurrence relations. The accuracy of the treecode approximation may therefore be flexibly controlled by the order of this Taylor expansion.
(3) Treecode has been applied widely to a variety of N-body problems, including Vortex Sheet [28], Ewald Summation [29], Radial Basis function [30], Plasma simulation [31], and Screened Coulomb potential [5,32]. It has also been extended to solving PDEs using a boundary integral formulation [33].
(4) The key advantages of the treecode algorithm compared with the popular FMM [4] are its ease of implementation, memory savings (an $O(N)$ cost with a small pre-factor), and efficient parallelization.

In this paper, we focus on strategies to improve the parallel performance of this flexible order Cartesian treecode algorithm [5,27]. Our main work is on developing a cyclically parallelized treecode, which is easy to implement with high parallel efficiency. By building the entire tree in the memory of each task, this cyclically parallelized treecode can rapidly compute interactions among tens of millions of particles. However, if the problem size exceeds the memory limit of each task, a scalable domain decomposition (DD) parallelized treecode using an orthogonal recursive bisection (ORB) tree can be used instead. Although the approach applies to general N-body problems, we particularly focus on the electrostatic interaction among charges distributed on the molecular surface [34,35] of proteins, which resembles the induced charges on each element when the molecular surface is discretized by triangles. Fast computation of such interactions forms a critical step toward efficient solution of the boundary integral Poisson–Boltzmann equation, as well as to efficient computation of the electrostatic potential at any spacial location [33]. In addition to computing the induced surface charge interactions on proteins, the current study has many other applications. For example, the parallelization strategy used here can be conveniently extended to other kernels and structures. Furthermore, computation of Coulomb interactions for the more accurate point multipole model (instead of the widely-used point partial charge model) demand highly efficient algorithms for computing N-body interactions [36,37].

The rest of this paper is organized as follows. In Section 2, we provide our treecode algorithm and MPI-based parallelization schemes, paying particular attention to the cyclic ordering scheme for optimal load balancing. In Section 3, we provide numerical results examining parallel performance on one selected protein with different treecode parameters for both sequential order and cyclic order, then on a series of proteins with various sizes and geometries. This paper ends with a section of concluding remarks.

2. Methods

In this section, we first briefly go over the flexible order Cartesian treecode algorithm (for further details see [5]), then provide our scheme for MPI-based parallelization, followed by the cyclic ordering scheme for improved load balance.

2.1. Treecode for electrostatic interactions

For a system of N particles located at $\mathbf{x}_i$ with partial charges $q_i, i = 1, \ldots, N$, we denote the induced potential at $\mathbf{x}_i$ by

$$V_i = \sum_{j=1, j\neq i}^{N} q_i\, G(\mathbf{x}_i, \mathbf{x}_j), \tag{1}$$

where $G(\mathbf{x}, \mathbf{y})$ is the Coulomb or the screened Coulomb potential, defined respectively by

$$G_0(\mathbf{x}, \mathbf{y}) = \frac{1}{4\pi\,|\mathbf{x} - \mathbf{y}|} \tag{2}$$

and

$$G_\kappa(\mathbf{x}, \mathbf{y}) = \frac{e^{-\kappa|\mathbf{x}-\mathbf{y}|}}{4\pi\,|\mathbf{x} - \mathbf{y}|}. \tag{3}$$

Note we attempted to use CGI units here but supply the additional 4π coefficient in the denominator to represent electrostatic potential generated from partial charges with units of fundamental charges, as from most force field generators such as CHARMM [38] and AMBER [39].

The cost of evaluating V_i for $i = 1, \ldots, N$ by direct summation is $\mathcal{O}(N^2)$, which is prohibitively expensive when N is large. This cost can be substantially reduced through the treecode algorithm, without significant loss of accuracy.

2.1.1. Particle–cluster interaction

We assume that the particles have been partitioned into a hierarchy of clusters as illustrated in Fig. 1(a). In the partition process, each cluster (a rectangle in 2-D or a rectangular parallelepiped in 3-D) is divided into four (or eight for 3-D) subclusters until the pre-determined treecode parameter N_0, the maximum number of particles per leaf (a cluster without subclusters), is satisfied. Here we illustrate in 2-D using $N_0 = 3$; the more practical 3-D case is similar. Treecode evaluates the potential in Eq. (1) as a sum of particle–cluster interactions,

$$V_i = \sum_{c} V_{i,c}, \tag{4}$$

where

$$V_{i,c} = \sum_{\mathbf{y}_j \in c} q_j\, G(\mathbf{x}_i, \mathbf{y}_j) \tag{5}$$

is the interaction between a target particle $\mathbf{x}_i$ and a cluster of sources $c = \{\mathbf{y}_j\}$. A particle–cluster interaction is shown schematically in Fig. 1(b): the cluster center, $\mathbf{y}_c$, is the geometric center of the rectangle; R is the particle–cluster distance; and the cluster radius, r_c, is the distance from $\mathbf{y}_c$ to one of the vertices of the rectangle.

The treecode algorithm has two options for computing a particle–cluster interaction $V_{i,c}$. It can use direct summation as in the definition Eq. (5), or Taylor approximation as in Eq. (9). In practice, the Taylor approximation is used if the following criterion is satisfied,

$$\frac{r_c}{R} \le \theta, \tag{6}$$

where θ is a user-specified Maximum Acceptance Criterion (MAC) parameter for controlling the error [9]. If the criterion is not satisfied, the code examines the children or sub-clusters of cluster c, or it performs direct summation if c is a leaf of the tree.

While this discussion has focused on the problem of evaluating the electrostatic potential V_i, similar considerations apply to computations of the electric field $E_i = -\nabla V_i$, where treecode can also be applied.

2.1.2. Cartesian Taylor expansion

If the particle $\mathbf{x}_i$ and cluster c are well-separated, i.e. the MAC (6) has been satisfied, then the terms in Eq. (5) can be expanded in a Taylor series with respect to $\mathbf{y}$ about $\mathbf{y}_c$,

$$G(\mathbf{x}_i, \mathbf{y}_j) = \sum_{\|\mathbf{k}\|=0}^{\infty} \frac{1}{\mathbf{k}!} D_{\mathbf{y}}^{\mathbf{k}} G(\mathbf{x}_i, \mathbf{y}_c)(\mathbf{y}_j - \mathbf{y}_c)^{\mathbf{k}}, \tag{7}$$

where Cartesian multi-index notation has been used with $\mathbf{k} = (k_1, k_2, k_3), k_i \in \mathbb{N}$, $\|\mathbf{k}\| = k_1 + k_2 + k_3$, $\mathbf{k}! = k_1!k_2!k_3!$, $\mathbf{y} = (y_1, y_2, y_3), y_i \in \mathbb{R}$, $\mathbf{y}^{\mathbf{k}} = y_1^{k_1} y_2^{k_2} y_3^{k_3}$, and $D_{\mathbf{y}}^{\mathbf{k}} = D_{y_1}^{k_1} D_{y_2}^{k_2} D_{y_3}^{k_3}$. The Taylor expansion Eq. (7) converges for $r_c < R$, and it plays the same role in treecode as the far-field multipole expansion in FMM. Substituting Eq. (7) into Eq. (5) yields

$$\begin{aligned} V_{i,c} &= \sum_{\mathbf{y}_j \in c} q_j \sum_{\|\mathbf{k}\|=0}^{\infty} \frac{1}{\mathbf{k}!} D_{\mathbf{y}}^{\mathbf{k}} G(\mathbf{x}_i, \mathbf{y}_c)(\mathbf{y}_j - \mathbf{y}_c)^{\mathbf{k}} \\ &= \sum_{\|\mathbf{k}\|=0}^{\infty} \frac{1}{\mathbf{k}!} D_{\mathbf{y}}^{\mathbf{k}} G(\mathbf{x}_i, \mathbf{y}_c) \sum_{\mathbf{y}_j \in c} q_j (\mathbf{y}_j - \mathbf{y}_c)^{\mathbf{k}} \qquad (8) \\ &\approx \sum_{\|\mathbf{k}\|=0}^{p} a^{\mathbf{k}}(\mathbf{x}_i, \mathbf{y}_c)\, m_c^{\mathbf{k}}, \qquad (9) \end{aligned}$$

where the Taylor series has been truncated at order p,

$$a^{\mathbf{k}}(\mathbf{x}_i, \mathbf{y}_c) = \frac{1}{\mathbf{k}!} D_{\mathbf{y}}^{\mathbf{k}} G(\mathbf{x}_i, \mathbf{y}_c) \tag{10}$$

is the $\mathbf{k}$th Taylor coefficient of the potential, and

$$m_c^{\mathbf{k}} = \sum_{\mathbf{y}_j \in c} q_j (\mathbf{y}_j - \mathbf{y}_c)^{\mathbf{k}} \tag{11}$$

is the $\mathbf{k}$th moment of cluster c. Note that the Taylor coefficients $a^{\mathbf{k}}(\mathbf{x}_i, \mathbf{y}_c)$ are independent of the sources $\mathbf{y}_j$ in cluster c, and the cluster moments $m_c^{\mathbf{k}}$ are independent of the target $\mathbf{x}_i$. These features may be exploited to reduce execution time.

2.1.3. Recurrence relation

Explicit formulas for the Taylor coefficients of the Coulomb and screened Coulomb potentials in Eq. (10) would be cumbersome to evaluate. However, we may leverage recurrence relations to efficiently compute these coefficients to high order [5,28]. To this end, we define an auxiliary function and its Taylor coefficients,

$$\psi(\mathbf{x}, \mathbf{y}) = e^{-\kappa|\mathbf{x}-\mathbf{y}|}, \quad b^{\mathbf{k}}(\mathbf{x}, \mathbf{y}) = \frac{1}{\mathbf{k}!} D_{\mathbf{y}}^{\mathbf{k}} \psi(\mathbf{x}, \mathbf{y}). \tag{12}$$

With these functions, the recurrence relations are given by [5]

$$\begin{aligned} \|\mathbf{k}\| |\mathbf{x} - \mathbf{y}|^2 a^{\mathbf{k}} - (2\|\mathbf{k}\| - 1) \sum_{i=1}^{3} (x_i - y_i) a^{\mathbf{k}-\mathbf{e}_i} + (\|\mathbf{k}\| - 1) \\ \times \sum_{i=1}^{3} a^{\mathbf{k}-2\mathbf{e}_i} \\ = \kappa \left(\sum_{i=1}^{3} (x_i - y_i) b^{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3} b^{\mathbf{k}-2\mathbf{e}_i} \right), \qquad (13) \\ \|\mathbf{k}\| b^{\mathbf{k}} = \kappa \left(\sum_{i=1}^{3} (x_i - y_i) a^{\mathbf{k}-\mathbf{e}_i} - \sum_{i=1}^{3} a^{\mathbf{k}-2\mathbf{e}_i} \right), \qquad (14) \end{aligned}$$

for $\|\mathbf{k}\| \ge 2$, where $\mathbf{e}_i$ are the Cartesian basis vectors. Note that although the equations for $a^{\mathbf{k}}$ and $b^{\mathbf{k}}$ are coupled, these can be solved by explicit marching; the values of $a^{\mathbf{k}}, b^{\mathbf{k}}$ for $\|\mathbf{k}\| = 0, 1$ are computed from the definitions, and then the recurrence relations are applied to compute the coefficients for $\|\mathbf{k}\| \ge 2$. The computational cost of these recurrence relations for the screened Coulombic interaction/kernel is $O(p^3)$. We further note that Tausch [7] has developed similar recurrence relations for arbitrary Green's functions of Cartesian based FMM having $O(p^4)$ complexity.

2.2. MPI-based parallelization

In designing our MPI-based parallelization strategy, we point out that treecode requires low $O(N)$ memory usage, and our focus is on computing interactions between induced charges on triangular elements characterizing molecular *surfaces*. We therefore store an identical copy of the entire tree on each MPI task (even for very large systems), permitting the application of a simple replicated data algorithm. Assuming that each MPI task has 24 GB of available memory, our parallel algorithm can handle interactions between about 20 million charged particles, which is more than needed in this biological scenario. However, we note that for some three-dimensional applications, e.g. in astrophysics, which have much larger numbers of particles, this approach of tree replication will rapidly limit scalability. To this end, we can alternatively apply a scalable domain decomposition (DD) parallelized treecode using an orthogonal recursive bisection (ORB) tree [40]. Numerical results using both treecode parallelization strategies are provided for comparison.

In treecode, we loop over target particles, and each particle can be treated as an independent interaction with the tree, whose copies are available on every task. Hence our implementation divides the particle array into n_p segments (for the sequential scheme, see below) or groups (for the cyclic scheme, see below) of size N/n_p, where n_p is the number of tasks, and the segments/groups are processed concurrently. The pseudocode is shown in Table 1. Communications are handled using the `MPI_Allreduce` routine with the `MPI_SUM` reduction operation [41].

2.3. Optimal load balancing

The initial and intuitive method to assign target particles to tasks is to use *sequential ordering*, in which the 1st task handles the first N/n_p particles in a consecutive segment, the 2nd task handles the next N/n_p particles, etc. The illustration of this job

Table 1
Pseudocode for MPI-based parallel treecode for electrostatic potential using replicated data.

```
1    On the main task:
2        Read protein geometry data (atom locations)
3        Generate triangulation, and assign particles at triangle centroids with unit charges
4        Copy particle locations to all other tasks
5    On each task:
6        Build local copy of tree and compute moments
7        Compute assigned segment/group of source terms by direct sum
8        Compute assigned segment/group of particle–cluster interaction by treecode
9        Copy result to the main task
10   On the main task:
11       Add segments/groups of all interactions and output result
```

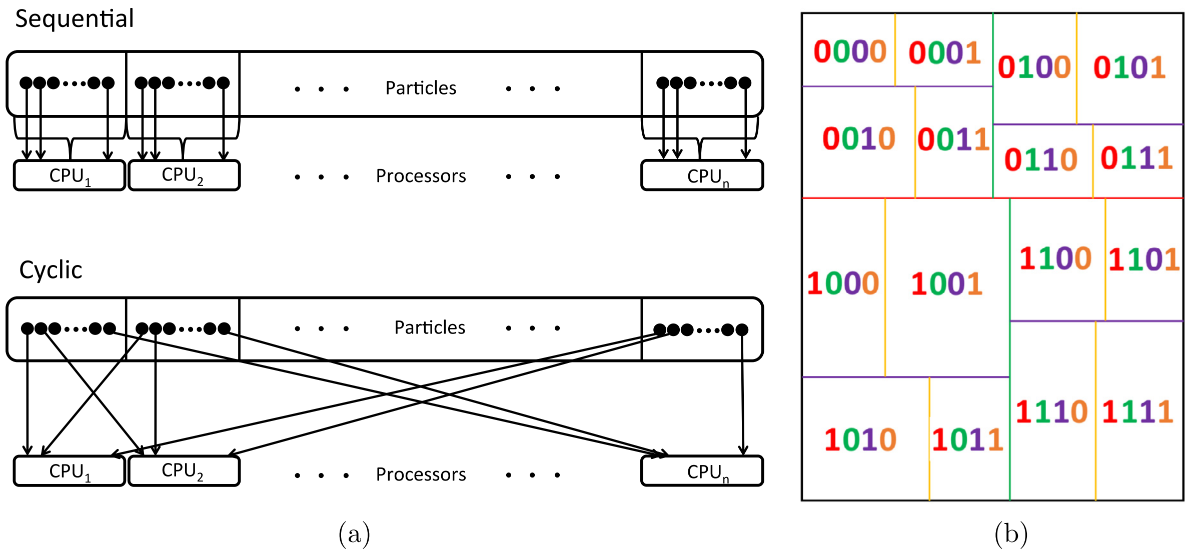


Fig. 2. (a): methods for assigning target particles to tasks: sequential order (top) vs. cyclic order (bottom); (b): an illustration of an ORB tree using tasks 0–15 in four subdivisions. The binary code in color shows the partner of each task at different level. For example: task 0 $\sim (0000)_2$ has task 8 $\sim (1000)_2$, task 4 $\sim (0100)_2$, task 2 $\sim (0010)_2$, and task 1 $\sim (0001)_2$ as its 0–1 partner at level 1 (red), level 2 (green), level 3 (purple), and level 4 (orange) respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

assignment is shown in the top of Fig. 2(a). However, when examining the resulting CPU time on each task, we noticed starkly different times on each task, indicating a severe load imbalance. This may be understood by the fact that for particles at different locations, the types of interactions with the other particles through the tree can vary. For example, a particle with only a few close neighbors uses more particle–cluster interactions than particle–particle interactions, thus requiring less CPU time than a particle with many close neighbors. We also notice that for particles that are nearby one another, their interactions with other particles, either by particle–particle interaction or particle–cluster interaction, are quite similar, so some consecutive segments ended up computing many more particle–particle interactions than others that were instead dominated by particle–cluster interactions. Based on these observations, we designed a *cyclic ordering* scheme, as illustrated on the bottom of Fig. 2(a) to improve load balancing. In this scheme, particles nearby one another are uniformly distributed to different tasks. For example, for a group of particles close to each other, the first particle is handled by the first task, the second particle is handled by the second task, etc. The cycle repeats starting from the $(n_p + 1)$th particle. The numerical results that follow demonstrate the significantly improved load balance from this simple scheme. We note that we also tried other approaches, such as using random numbers to assign particles to tasks, but these did not result in as significant improvements as the cyclic approach.

2.4. Domain decomposition parallelized treecode

The cyclically parallelized treecode algorithm has two significant advantages: easy implementation and high parallel efficiency. However, due to the fact that the entire tree is built on each task, the scale of the problem this algorithm can handle is limited by the memory capacity associated to each task. As a remedy, for very large problems beyond this memory limit, we implement a Domain Decomposition (DD) parallelized treecode under the framework of the orthogonal recursive bisection (ORB) tree from Salmon's thesis [40], whose open source C++ implementation using the 0th moment (center of mass) is contributed by Barkman and Lin [42]. Here we briefly describe the DD-parallelized treecode using the ORB tree structure.

Starting from one rectangular domain containing all particles, the ORB treecode algorithm recursively divides particles into two equal amounts of groups by splitting the domain using an orthogonal hyperplane (perpendicular to the longest dimension of the domain) until the finest level in which the number of tasks equals the number of subdomains at that level as illustrated in Fig. 2(b). In this manner, each task, as loaded with the same number of particles, is associated to a subdomain and has a partner task (illustrated as the 0–1 difference using the same color in their binary code) at each level of the ORB tree division. Once the ORB tree is constructed, each task builds a local B-H tree [5,9] based on their loaded particles, and communicates with its partner task at each level to exchange additional tree structure information such as clusters and their moments. Here cluster information is sent only when the maximum acceptance criteria (MAC) between particles of the receiving task and clusters on the sending task is satisfied. After this procedure each task stores only a small part of the entire tree such that the far fields are seen only at a coarse level while near fields are seen down to the leaves, as controlled by the MAC. Note that such a "local essential tree" is a subset of the full tree and is the necessary tree structure information for computing interactions between the task's loaded particles and the entire tree. This is the major difference from the cyclically parallelized treecode in which the entire tree is built in the memory of each task. The details of constructing the ORB tree can be found in [40] and our new and additional contribution is to implement the *arbitrary order* Taylor expansion as opposed to the original 0th order (center of mass) expansion. In updating the moments for lower levels of (larger) clusters using moments from higher levels of (smaller) clusters, a moments to moments (MtM) transformation as described in [7] is applied.

3. Results

The numerical results in this section serve four purposes. First, we show that the cyclic ordering scheme improves the parallel efficiency of the treecode by improving the load balance. Second, we show how the treecode parameters such as N_0, the number of maximum particles per leaf, p, the order of Taylor expansion, and θ, the maximum acceptance criterion, affect the parallel efficiency. By comparing the parallel efficiency of the cyclic and sequential ordering schemes at different combinations of treecode parameters, we show that the cyclic scheme reduces the effect of these parameters while providing a uniformly improved parallel efficiency. With these data, treecode users can choose an optimal combination of treecode parameters subject to the trade-off between time and error. Third, we provide numerical results in a cube with uniformly distributed charges and on the molecular surfaces of a series of proteins of various sizes, to demonstrate the general usage and consistent performance of the cyclic ordering scheme for our MPI-based parallelization. Finally, the comparison between cyclically parallelized treecode and DD-parallelized treecode in computing electrostatic interactions for charges distributed on the molecular surface of a protein is provided, showing the advantages in parallel efficiency for the former and in memory scalability for the latter.

Except for the one example on a cube, these numerical results compute the electrostatic potential induced from partial charges (point charges) distributed on molecular surfaces. Throughout this section, we report the relative L_2 error of the electrostatic potentials,

$$e_\phi = \left(\frac{\sum_{i=1}^{N} |\phi^{\text{num}}(x_i) - \phi^{\text{dir}}(x_i)|^2}{\sum_{i=1}^{N} |\phi^{\text{dir}}(x_i)|^2} \right)^{1/2} \tag{15}$$

where N is the number of charged elements of the triangulated surface, ϕ^{dir} is the potential computed using direct summation (which serves as a reference value), and ϕ^{num} is the potential computed using treecode.

Unless specified otherwise, simulations are run on the *ManeFrame* cluster, sponsored by the Southern Methodist University (SMU) Center for Scientific Computing. This cluster has 1084 nodes, each with 24G of RAM and 8-core Intel Xeon CPU X5560 @ 2.80 GHz processors. Each simulation uses up to 128 cores, with one MPI task assigned per core. The code is written in C and compiled using the mvapich2/2.0-gcc-4.9.1 library with the `-O2` optimization flag. This cluster uses a high speed DDR infiniband network at 20 Gbps for its interconnect.

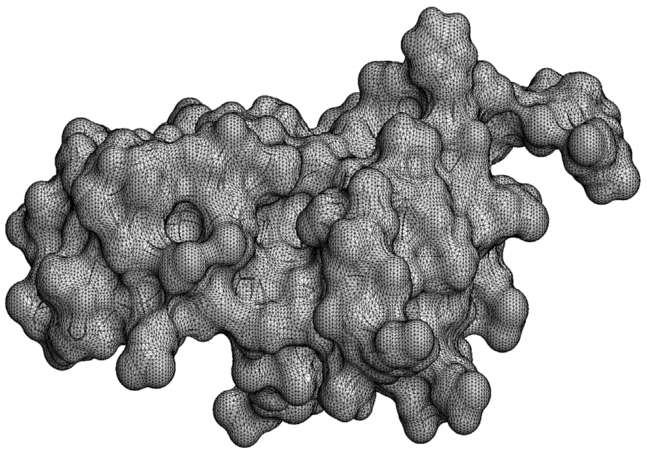


Fig. 3. triangulated molecular surface of the protein 1a63 with MSMS density 10 (vertices/Å^2), which produces 132,196 triangles with point charges at centroids. Note: we use a density of 20 in our simulations, however we show a density of 10 here for better illustration of the triangular surface mesh.

3.1. Improving parallel efficiency through optimal load balance

We first focus on one particular protein to extensively study the parallel performance of our algorithms. We pick the protein with PDB ID 1a63 (Protein Data Bank: www.pdb.org) with 2069 atoms/130 residues. Beyond the structure, the biological significance of this protein is not our main concern. In our simulations, the molecular surface is generated and triangulated by the mesh generator MSMS [43], with atom locations obtained from the PDB file. The software MSMS has a user-specified density parameter d that controls the number of vertices per Å^2 in the triangulation. For this case, the MSMS density d is chosen to be 20, which produces 265,000 triangles. We choose the treecode order $p = 3$, maximum number of particles $N_0 = 500$, and MAC parameter $\theta = 0.8$, for the screened Coulombic potential with ionic screening parameter $\kappa = 1$. The particles are located at the centroid of each triangle with unit partial charges. An illustration of the triangulated molecular surface of protein 1a63 with reduced density $d = 10$ is given in Fig. 3 to illustrate the triangulated surface.

Table 2 reports the CPU time and parallel efficiency (P.E.) for both the cyclically and sequentially parallelized treecode methods using increasing numbers of tasks. For comparison purposes, the same values are also reported for the parallelized direct summation method $(\mathcal{O}(N^2))$. From the "Direct Sum" columns, the CPU time is essentially halved when the number of tasks is doubled, indicating a 95+% parallel efficiency when using 128 tasks. This is due to the fact that electrostatic interactions computed by direct sum for all particles are homogeneous, resulting in almost perfect load balance. However, when the treecode is used for electrostatic interactions, this homogeneity is no longer maintained, as different particles interact with the tree differently.

For the "Treecode" columns in Table 2, the "N-body Interactions + Utilities" columns contain both the serial computations (build the tree and compute the moments) and parallel computations (compute the electrostatic interaction). The "N-body Interactions" columns contain only the parallel computations. We report the CPU time and parallel efficiency for both the sequentially parallelized treecode (seq.) and the cyclically parallelized treecode (cyc.). Due to the fact that small portions of the code are not parallelizable (the "serial" part), when 128 tasks are used the parallel efficiency of sequentially parallelized treecode for "N-body Interactions + Utilities" is reduced to 48%, while the cyclically parallelized treecode improves it to 53%. However,

Table 2
CPU time and parallel efficiency (P.E.) for parallelized direct sum, sequentially parallelized treecode (seq.) and cyclically parallelized treecode (cyc.) for computing electrostatic interactions on the molecular surface of protein 1a63 with 265,000 triangles. The treecode parameters are $\theta = 0.8$, $N_0 = 500$, and $p = 3$, resulting in relative L_2 error $e_\phi = 9.65 \times 10^{-3}$. The number of tasks n_p ranges over $1, 2, 4, \ldots, 128$. We use "amd." to denote the parallel efficiency predicted by Amdahl's law.

n_p	Direct sum		Treecode								
			N-body interaction + Utilities					N-body interaction			
	CPU (s)	P.E. (%)	CPU (s)		P.E. (%)			CPU (s)		P.E. (%)	
			seq.	cyc.	seq.	cyc.	amd.	seq.	cyc.	seq.	cyc.
1	3408.51	100.00	18.06	18.04	100.00	100.00	100.00	17.97	17.96	100.00	100.00
2	1708.79	99.73	9.31	9.12	96.99	98.93	99.20	9.22	9.03	97.44	99.41
4	887.77	95.98	5.11	4.83	88.30	93.44	97.63	5.02	4.73	89.51	94.83
8	446.31	95.46	2.64	2.46	85.40	91.52	94.63	2.55	2.37	88.13	94.70
16	222.70	95.66	1.44	1.29	78.63	87.08	89.17	1.34	1.20	83.74	93.44
32	110.51	96.39	0.80	0.70	70.70	80.90	79.93	0.70	0.60	80.18	93.01
64	55.66	95.69	0.47	0.41	59.93	68.23	66.21	0.38	0.32	74.37	87.72
128	27.80	95.80	0.29	0.27	48.41	53.01	49.29	0.20	0.17	71.10	81.51

Table 3
Profile of 8 costliest subroutines in treecode for computing the electrostatic potential. Treecode parameters: $\theta = 0.8$, $N_0 = 500$, $p = 3$, and $N = 265{,}000$.

Index	%	Time (s)	Subroutine	Description
1*	67.75	7.66	compp_direct	Compute direct summation
2*	16.28	1.84	compp_tree	Compute particle–cluster interactions
3*	13.98	1.58	comp_tcoeff	Compute Taylor coefficients
$\underline{4}$	0.97	0.11	readin	Input protein structure, triangulation
5	0.62	0.07	comp_ms	Compute moments
$\underline{6}$	0.18	0.02	main	Main subroutine
7	0.18	0.02	partition	Partition particles into upper/lower groups
$\underline{8}$	0.09	0.01	triangle_area	Calculate triangle area of each element

if we consider only the parallelizable portion of the algorithm, consisting of the N-body electrostatic interactions after the tree has been constructed, the parallel efficiency with 128 tasks is 71.10% for the sequential ordering scheme, and 81.51% for the cyclic ordering scheme, a very encouraging result for the treecode parallelization.

The reduction in overall parallel efficiency (N-body Interaction + Utilities) can be well-explained by the information contained in Table 3. Here, we report the profile (CPU time elapsed on each routine, excluding the portion taken by its subroutines) of the eight most time-consuming subroutines. These results were obtained using the GNU profiling tool `gprof`, which returns the top time-consuming subroutines, with one MPI task. We divide these subroutines into three groups: Group 1 (underlined indices 4,6,8) are those that generate particle location and charges, which are not included in our CPU time calculation for Table 2; Group 2 (bold indices 5 and 7) are the subroutines that build the tree and compute the moments, which are implemented in serial; Group 3 (indices with a star: 1–3) are subroutines for the N-body interaction, which are implemented in parallel. The time of Group 2 is negligible compared with Group 3 for small numbers of tasks. However, as more tasks are used, the percentage of time used by Group 2 becomes more and more significant, which explains the overall parallel efficiency reduction in Table 2.

The reduction in parallel efficiency when "Utilities" are included in the CPU time can be predicted using Amdahl's law. To see this, we set $T_1 = 18.04$ from the first entry of the 5th column of Table 2 as the serial time. We then use the data from Table 3 to compute the parallelizable fraction $f = t_1/(t_1 + t_2) = 11.08/11.17 \approx 0.9919$, where $t_1 = 7.66 + 1.84 + 1.58 = 11.08$ is the CPU time of Group 1 and $t_2 = 0.07 + 0.02 = 0.09$ is the CPU time of Group 2. Amdahl's law then predicts the parallel efficiency as $T_1/T(n_p)/n_p$, where $T(n_p) = f(T_1/n_p) + (1 - f)T_1$. These predictions are shown in the "amd." column in Table 2, which are relatively consistent with the computed parallel efficiency of the cyclic order scheme, "cyc.".

To examine how the choice of sequential versus cyclic ordering scheme affects load balancing, we plot the CPU time on each task for computing the electrostatic interactions in Fig. 4. From Fig. 4(a)(c)(e), we can see that when using 32, 64, and 128 tasks for the protein 1a63, the cyclic scheme has much more balanced load (red circles) than the sequential scheme (blue squares). To verify this result for more general cases, e.g. for particles distributed uniformly in space, in Fig. 4(b)(d)(f) we perform the same experiment for 265,000 particles, uniformly distributed on a cube with 10 Å length per side. The result shows a similar pattern as for protein 1a63, which justifies the general use of the cyclic order scheme. We quantified this load balance by calculating the standard deviations among all the CPU times used within each experiment (shown in Table 4), which shows significant reduction in standard deviation of the CPU time for the cyclic ordering scheme in comparison with the sequential ordering scheme.

Table 4
The standard deviation of the CPU times reported in Fig. 4.

	32 cores		64 cores		128 cores	
	Sequential	Cyclic	Sequential	Cyclic	Sequential	Cyclic
1a63	0.0561	0.0023	0.0318	0.0021	0.0181	0.0033
Cube	0.0565	0.0012	0.0332	0.0048	0.0195	0.0006

3.2. The effect of treecode parameters on parallel efficiency

The CPU time and memory use of the serial treecode algorithm subject to treecode parameters has been extensively examined in [5]. Here we study how the CPU time and parallel efficiency change subject to treecode parameters within our parallel implementation. We first show in Section 3.2.1 that the cyclic ordering scheme improves the parallel efficiency at various choices of the parameters, and it also makes the parallel efficiency less sensitive to the choice of parameters in a case, as compared with the sequential ordering scheme. This is done by plotting the parallel efficiency as each parameter is varied. Second, in Section 3.2.2 we show that the general stability of cyclic ordering scheme can

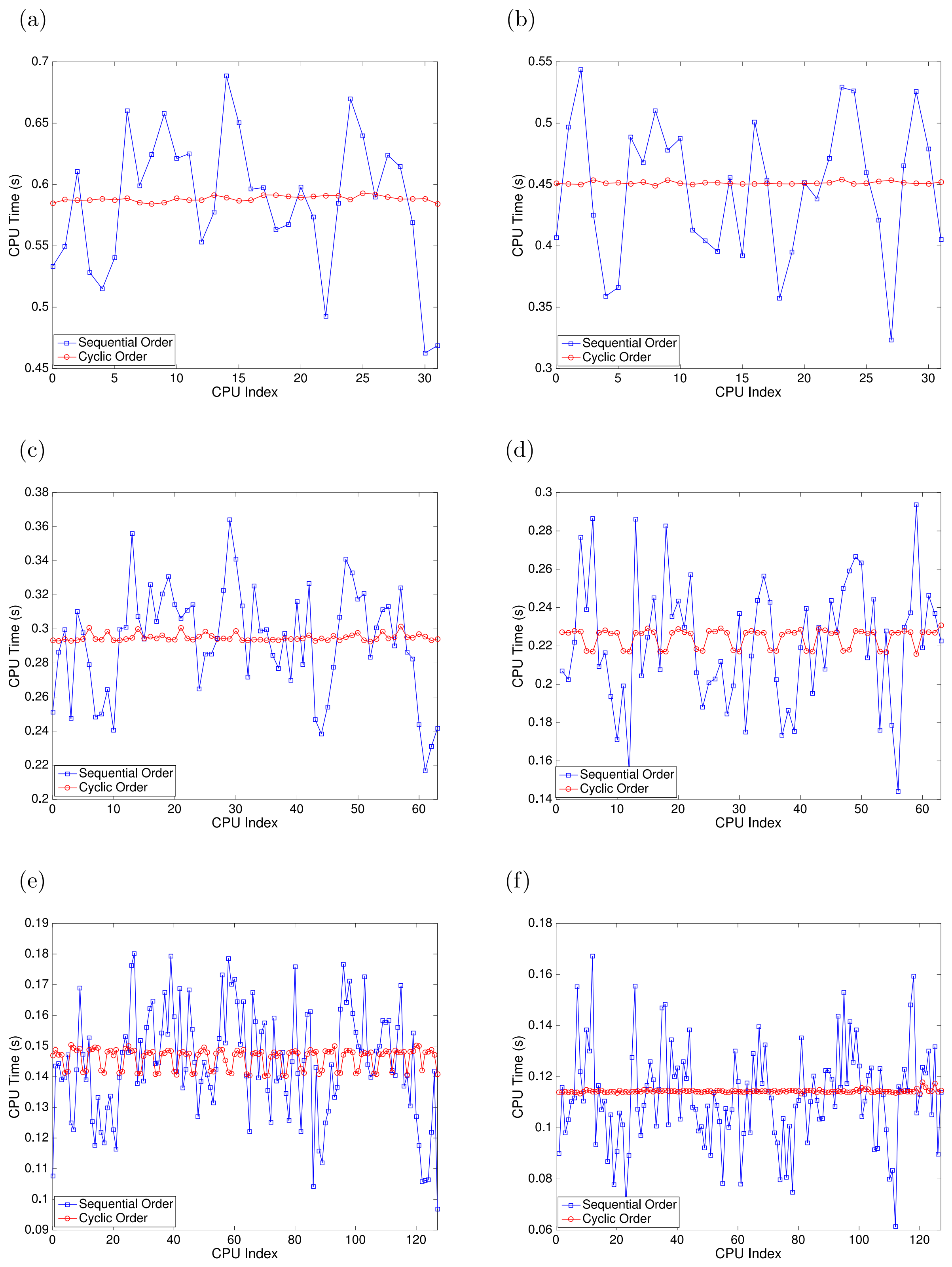


Fig. 4. CPU time consumed on each task with sequential ordering (red circles) and cyclic ordering (blue squares) for 32 tasks (a)(b), 64 tasks (c)(d), and 128 tasks (e)(f), using protein 1a63 with 265,000 triangles (a,c,e) and on a cube with 265,000 particles (b,d,f). Treecode parameters: $\theta = 0.8$, $N_0 = 500$, $p = 3$.

be further verified on a time-error scatter plot, which can also help us to choose the most efficient treecode parameters for a desired error tolerance. Third, in Section 3.2.3 we use a (parallel efficiency)-error scatter plot to reveal the fact that when the number of particles per leaf (N_0) is small, the parallel efficiency becomes more sensitive to the choices of other treecode parameters, thus leaving space for future research on improving parallel efficiency.

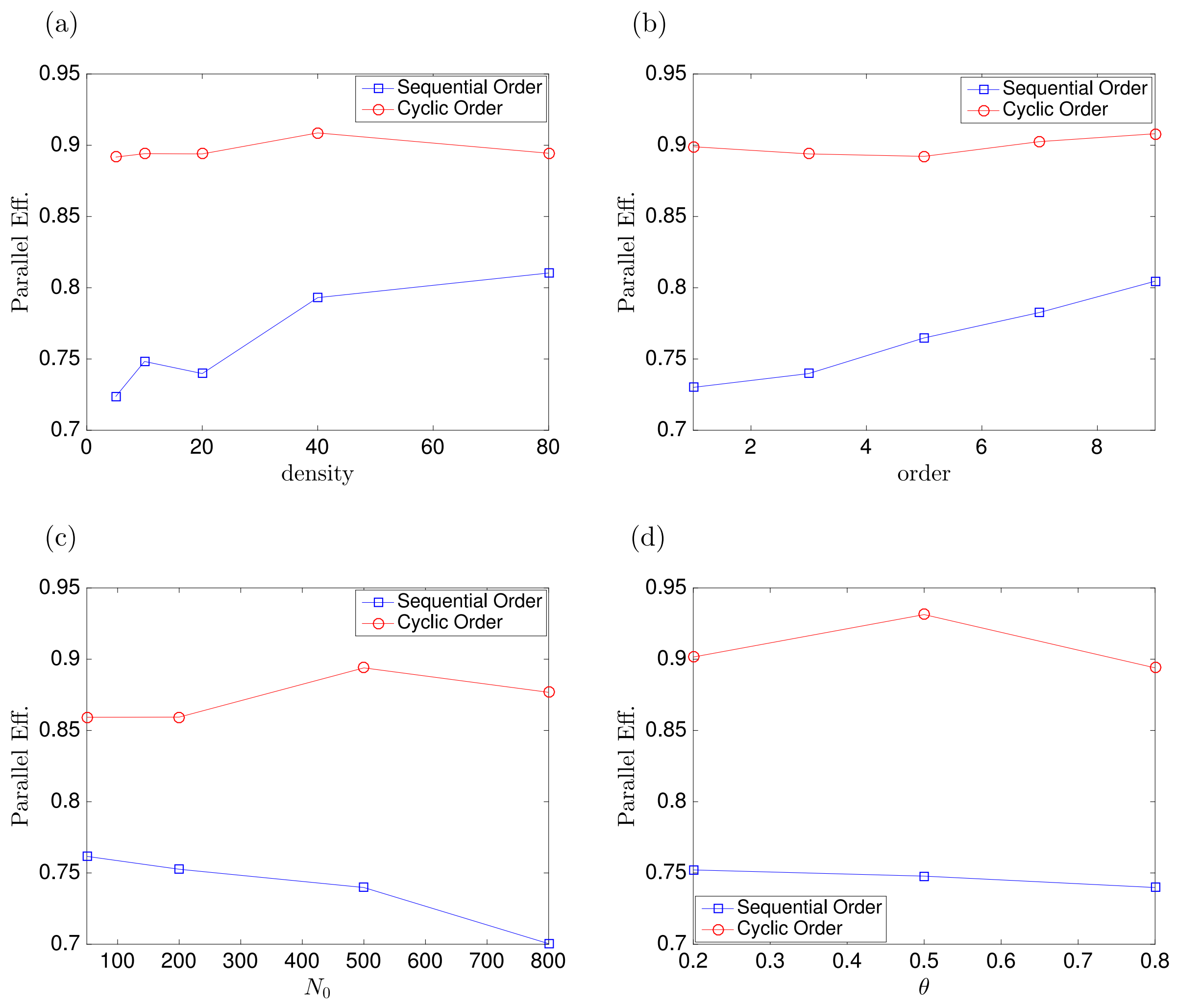


Fig. 5. Effects of treecode parameters on parallel efficiency (1 vs. 64 tasks). Reference treecode parameters: $\theta = 0.8$, $N_0 = 500$, $p = 3$, $d = 20$. (a) $d = 5, 10, 20, 40, 80$ (resulting $N = 70{,}018,\ 132{,}196,\ 265{,}000,\ 536{,}886,\ 1{,}100{,}549$); (b) order $p = 1, 3, 5, 7, 9$; (c) $N_0 = 50, 200, 500, 800$; (d) $\theta = 0.2, 0.5, 0.8$.

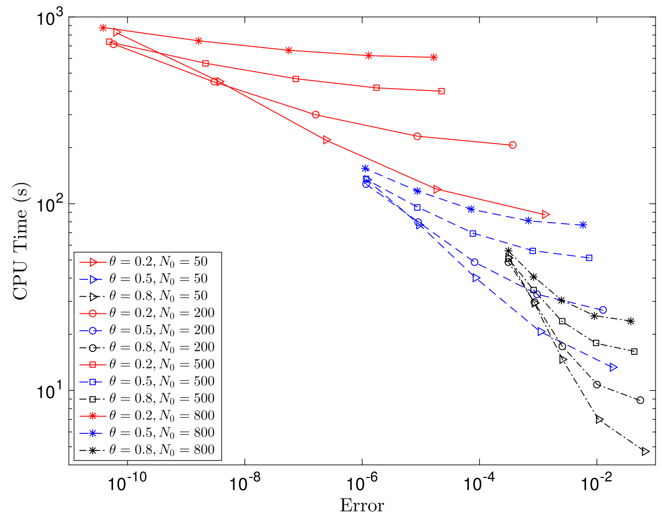
3.2.1. *Parallel efficiency across various treecode parameters*

We consider four treecode parameters: number of particles per leaf N_0, MSMS density d (proportional to the number of particles N), Taylor expansion order p, and MAC threshold θ. We change these variables one at a time and plot the parallel efficiency (N-body interaction only) using 64 tasks in Fig. 5. Each data point on these plots represents a different combination of the parameters θ, N_0, p, and d. We use $\theta = 0.8$, $N_0 = 500$, $p = 3$, and $d = 20$ ($N = 265{,}000$) as fixed parameters when one of the parameters is changing. Two significant patterns may be observed from these graphs. First, the cyclic ordering scheme shows better parallel efficiency than the sequential ordering scheme for all parameter combinations. Second, the cyclic ordering scheme has more stable results than the sequential ordering scheme, as all the red circles are near 90% when different choices of the parameters are made, whereas the blue squares fluctuate more significantly (particularly as d and p are varied). Additional simulations on several other proteins confirms the first pattern, but not the second pattern in general.

3.2.2. *Scatter plot results: time vs. error*

To further investigate the parallel treecode using the cyclic ordering scheme, we provide scatter plots of CPU time versus error as treecode parameters are varied. Each data point represents a different combination of the θ, N_0, and p. Once again, we use $\theta = 0.8$, $N_0 = 500$, $p = 3$ as fixed parameters when one of the parameters is changing. The density d is uniformly kept at 20 (thus $N = 265{,}000$).

Fig. 6(top) is the scatter plot when using one task; similar results can be found in our previous work [5,33]. Using this scatter plot, we can identify optimal combinations of the parameters θ, N_0, and p for a given accuracy tolerance. For example, to obtain a relative error of 10^{-4}, one should choose $N_0 = 50$, $\theta = 0.5$, and $p = 5$ for the fastest speed. Additionally, we see that for fixed θ (same color), the order p essentially determines the accuracy (especially for larger θ). Therefore, if these two parameters are fixed for a desired accuracy, smaller N_0 will generally provide the best results. This can be explained by the fact that smaller N_0 results in deeper trees, and thus more particle–cluster interactions are used, resulting in reduced CPU time. Fig. 6(bottom) is the corresponding scatter plot when using 128 tasks. We note that this shows similar results as the single task plot, which supports our general conclusion that the parallel efficiency when using the cyclic ordering scheme is rather stable. Both scatter plots show that larger θ brings faster but less accurate results. In addition, they indicate that smaller N_0 (triangle) uses less CPU time for a desired error at fixed p and θ, thus we should use smaller N_0 for faster calculations. However, if we instead investigate a scatter plot of parallel efficiency against error, as in the next section, an interesting phenomenon is revealed.



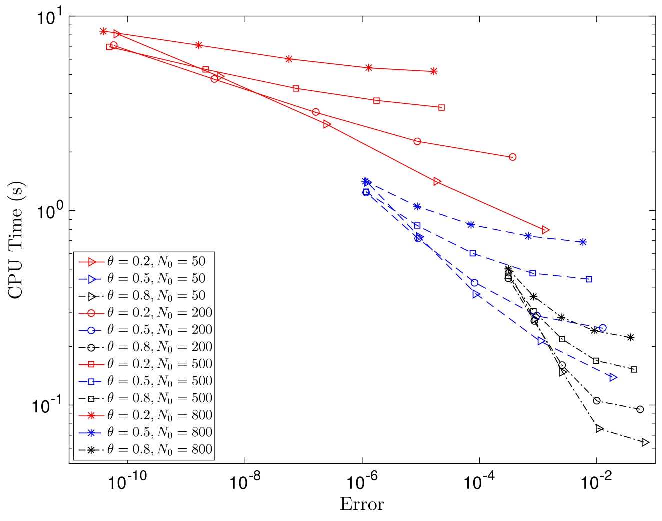


Fig. 6. Scatter plots of time vs. error on protein 1a63 for 1 task (top) and 128 tasks (bottom). Treecode parameters: $p = 1, 3, 5, 7, 9$ from right to left on one connected line; $d = 20$ ($N = 265{,}000$); $N_0 = 50, 200, 500, 800$; $\theta = 0.2, 0.5, 0.8$.

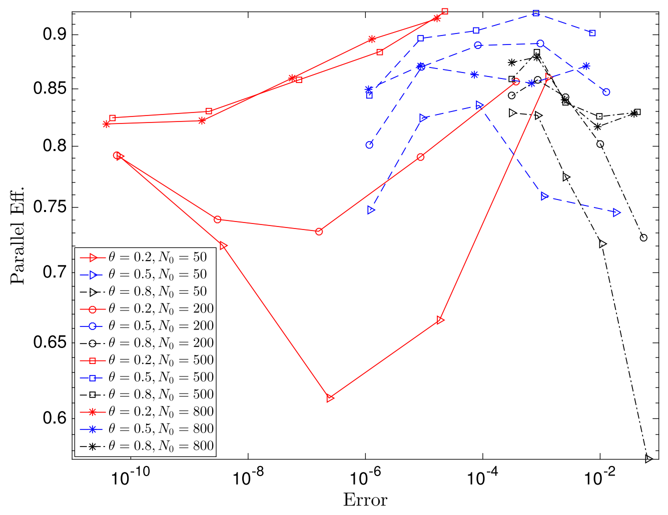


Fig. 7. Scatter plot of parallel efficiency vs. error. Treecode parameters: $p = 1, 3, 5, 7, 9$ from right to left on one connected line; $N = 265{,}000$; $N_0 = 50, 200, 500, 800$; $\theta = 0.2, 0.5, 0.8$.

3.2.3. *Scatter plot results: parallel efficiency vs. error*

In Fig. 7 we show a scatter plot for the same testing case as in Fig. 6, but where the vertical axis is the parallel efficiency instead of the CPU time. While the results in this plot may at first seem slightly erratic, they elucidate some interesting phenomena. First, if we temporarily hide the lines with triangles ($N_0 = 50$), all parallel efficiency values are within the 75%–90% range, and thus the parallel efficiency values are generally stable. Second, focusing only on the lines connecting triangles ($N_0 = 50$) we see that the parallel efficiency values fluctuate rapidly, indicating that the load balance becomes sensitive to p and θ when N_0 is small. This observation opens space for future research to improve parallel efficiency for the case of small numbers of particles per leaf, N_0.

3.3. *Accuracy and efficiency across a wide collection of proteins*

We finally compute the electrostatic interactions on a series of 24 proteins with different sizes and geometries. The numerical results are reported in Table 5. Here, the first column is an identification index for convenience in the discussion that follows. The second column is each protein's four-digit protein data bank (PDB) ID. Column 3 is the number of elements on the triangulated molecular surfaces of each protein. This determines the surface areas of each protein, as shown in column 6. Column 4 is the number of atoms in, and column 5 is the total charge carried by, the proteins, respectively. We uniformly choose MSMS density $d = 20$ so that proteins with larger molecular surface areas (column 6) will normally generate larger numbers of elements. A few exceptions occur because MSMS modifies the given density to fit its triangulation needs, resulting in a slightly mismatched order for the data in columns 3 and 6; however, the general pattern remains that a larger number of atoms results in larger molecular surface areas and numbers of elements. Columns 7 and 8 show the time for computing N-body electrostatic interactions on one CPU (T_1) and 128 CPUs ($T(128)$) in seconds, respectively. Column 9 reports the parallel efficiency calculated resulting from columns 7 and 8, and indicate an overall parallel efficiency of 70%–85%. More importantly, larger systems generally result in higher parallel efficiency. Column 10 shows the memory used in each calculation, which is small and linear $O(N)$ with respect to N from column 3. This memory saving feature is one of the key advantages of the treecode algorithm. The last column shows the L_2 potential errors, which are consistently about 10^{-2}. This table demonstrates the general applicability of the cyclic ordering scheme for parallel treecode on different geometries and structures.

3.4. *Comparison between cyclically parallelized and DD-parallelized treecodes*

Here we provide numerical results demonstrating difference in terms of CPU time, parallel efficiency and memory usage between the cyclically parallelized treecode and the DD-parallelized treecode, both using the ORB tree structure for a fair comparison. The simulations are run on clusters using intel processors (Intel(R) Xeon(R) Gold 6148 CPU @ 2.40 GHz) sponsored by the Institute for Cyber-Enabled Research (ICER) at Michigan State University (MSU).

Table 6 shows a comparison between domain decomposition (DD) parallelized ORB treecode and cyclically parallelized ORB treecode (cyc.) in computing electrostatic interactions between charges located on the surface of protein 1a63 as used in Table 2. In the column titled “N-body Interactions + Utilities”, the time for tree construction and the moment computation are included

Table 5
Parallel efficiency for electrostatic interaction calculations on molecular surfaces of 24 proteins: N is the number particles; treecode parameters are $p = 3$, $\theta = 0.8$, $N_0 = 500$, $d = 20$.

ID	PDB	N	N_a	Charge	Area	T_1 (s)	$T(128)$ (s)	P. E. (%)	Memory (K)	e_ϕ
1	1ajj	81 798	519	−5	2171.5	5.121	0.055	73.43	10 264	9.56e−3
2	2erl	87 136	573	−6	2323.5	5.738	0.059	75.68	10 868	9.40e−3
3	1cbn	88 765	648	0	2371.4	5.307	0.060	69.72	11 156	8.40e−3
4	1vii	94 458	596	2	2482.1	6.795	0.067	78.81	11 624	9.42e−3
5	1fca	95 901	729	−7	2552.7	7.738	0.076	79.37	12 012	1.07e−2
6	1bbl	98 311	576	1	2610.6	5.636	0.060	73.72	12 324	9.43e−3
7	1sh1	102 866	702	0	2750.1	6.268	0.067	73.55	12 828	8.71e−3
8	2pde	103 456	667	3	2721.7	6.776	0.069	76.81	12 828	9.14e−3
9	1vjw	105 744	828	−6	2792.4	5.837	0.061	74.34	13 216	8.90e−3
10	1uxc	107 704	809	4	2842.1	6.273	0.070	70.48	13 404	8.92e−3
11	1ptq	108 958	795	3	2904.0	7.164	0.073	77.11	13 472	9.13e−3
12	1bor	109 649	832	−3	2910.4	7.088	0.072	76.87	13 392	9.02e−3
13	1fxd	110 126	824	−15	2928.7	6.535	0.068	74.71	13 624	8.55e−3
14	1r69	115 278	997	4	3061.5	6.904	0.071	75.53	14 088	8.92e−3
15	1mbg	116 093	903	6	3080.5	7.396	0.076	76.12	14 164	9.17e−3
16	1bpi	120 948	898	6	3240.2	9.720	0.102	74.47	14 756	1.13e−2
17	1hpt	123 178	858	−1	3270.1	7.519	0.075	78.08	15 184	9.44e−3
18	451c	158 468	1216	−1	4168.6	9.231	0.092	78.31	18 980	9.72e−3
19	1frd	165 392	1478	−11	4377.2	11.327	0.109	81.09	19 268	8.68e−3
20	1a2s	169 679	1272	−9	4447.0	10.539	0.104	79.00	19 704	9.34e−3
21	1svr	176 906	1435	−2	4654.8	11.991	0.114	82.21	20 176	9.11e−3
22	1neq	179 327	1187	4	4727.4	12.502	0.119	81.79	20 300	9.87e−3
23	1a63	265 000	2065	−1	6989.4	17.960	0.172	81.51	30 348	9.65e−3
24	1a7m	294 285	2809	7	7751.8	27.428	0.250	85.70	32 584	1.12e−2

Table 6
CPU time, parallel efficiency (P.E.), and memory usage for DD-parallelized ORB treecode (DD) and cyclically parallelized ORB treecode (cyc.) for computing electrostatic interactions on the molecular surface of protein 1a63 with 265,000 triangles. The treecode parameters are $\theta = 0.8$, and $p = 3$. The number of tasks n_p ranges over $1, 2, 4, \ldots, 128$. Memory is reported as the average memory usage per core.

n_p	N-body interaction + Utilities				N-body interaction				Memory usage			
	CPU (s)		P.E. (%)		CPU (s)		P.E. (%)		Mem. (MB)		P.E. (%)	
	DD	cyc.	DD	cyc.	DD	cyc.	DD	cyc.	DD	cyc.	DD	cyc.
1	50.67	50.84	100.00	100.00	49.80	49.84	100.00	100.00	97.20	97.2	100.00	100
2	25.39	25.64	99.77	99.16	25.09	25.13	99.26	99.14	51.29	97.2	94.76	50
4	13.66	13.53	92.73	93.94	13.38	13.23	93.09	94.17	26.96	97.2	90.13	25
8	6.97	7.01	90.86	90.64	6.80	6.81	91.58	91.46	14.96	97.2	81.21	12.5
16	4.36	3.96	72.68	80.34	3.97	3.83	78.46	91.43	8.20	97.2	74.04	6.25
32	2.51	2.44	63.13	65.01	2.10	1.95	73.98	79.90	4.73	97.2	64.23	3.13
64	1.46	1.24	54.11	64.18	1.15	1.00	67.66	78.04	3.27	97.2	46.40	1.56
128	1.15	1.09	34.41	36.31	0.62	0.52	63.26	74.05	2.19	97.2	34.63	0.78

in addition to the time for computing electrostatic interactions. In the column titled "N-body Interactions", only the time for computing electrostatic interactions is recorded. From these two columns, we can see that the parallel efficiency for building local essential trees is not as high as N-body interactions, thus drags down the overall parallel efficiency. This is due to the intensive message passing process involved in building the tree and exchanging cluster information, particularly when high order moments are used for arbitrary order Taylor expansion. For example, when 128 cores are used the parallel efficiency of DD-parallelized treecode is 34.41% as compared with 36.31% for the cyclically parallelized treecode. Note in order to test the cyclically parallelized treecode on the ORB tree, a full tree on each task is built using local essential trees on all tasks by message passing. As for the time for "N-body Interaction", the parallel efficiency with 128 cores is 63.26% for DD-parallelized treecode and 74.05% for the cyclically parallelized treecode. This difference in parallel efficiency can be well explained by the load balance shown in Fig. 8. The cyclically parallelized treecode builds the entire tree in the memory of each task thus could optimize the load balance by cyclically assigning particles that are geometrically close to different tasks while the DD-parallelized treecode must assign particles that are geometrically close to the same task. Meanwhile, parallelization by building the entire B-H tree on each task has higher parallel efficiency as seen in Table 2 than parallelization by building local essential trees on each task as seen in Table 6. However, from the memory usage shown Table 6, we can see that the DD-parallelized treecode is scalable, thus it can handle very large sized N-body problems, while the cyclically parallelized treecode inherently limits the problem size to the memory capacity associated with each MPI task.

4. Conclusions

The flexible order Cartesian treecode method uses particle–cluster computations to replace the particle–particle computations for far-field interactions for N-body problems, thereby significantly reducing the computational cost from $O(N^2)$ to $O(N \log N)$. The key advantages of the treecode algorithm compared with the popular FMM [4] are its ease of implementation, memory savings (an $O(N)$ cost with a small pre-factor), and efficient parallelization.

In this paper, we show that through replication of the tree structure on all tasks, MPI-based parallelization of treecode is straightforward, but care must be taken when decomposing the work among tasks. To this end, the novel cyclic ordering scheme significantly improves the load balancing, and thus the parallel efficiency, in comparison with a standard sequential ordering. We show that when using 128 tasks for a protein surface with 265,000 partial charges, the cyclic ordering scheme can compute the pairwise screened Coulombic interaction in 0.17s with

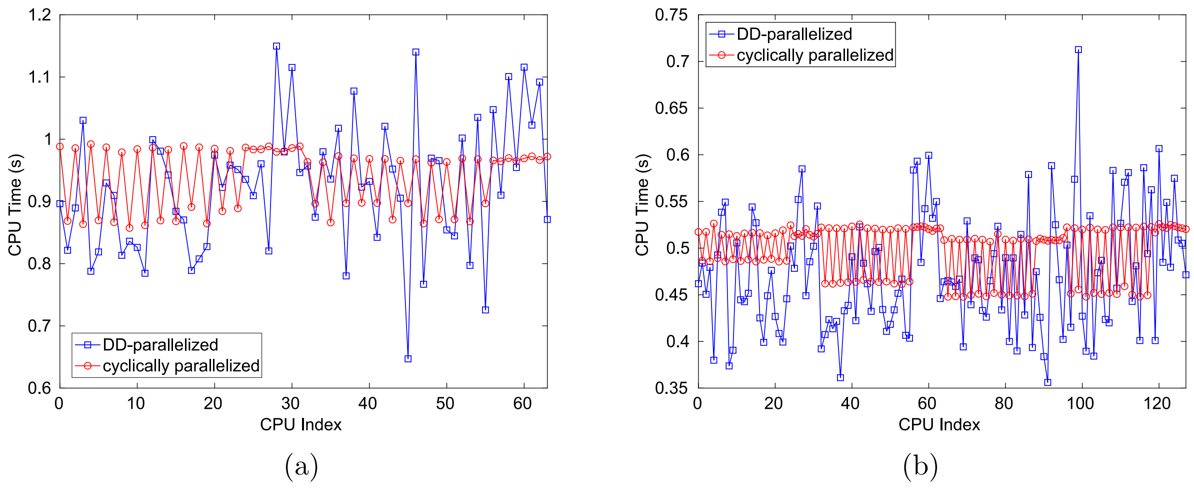


Fig. 8. CPU time consumed on each task with cyclically parallelized treecode (red circles) and DD-parallelized treecode (blue squares) for 64 tasks (a), and 128 tasks (b), using protein 1a63 with 265,000 triangles. Treecode parameters: $\theta = 0.8$, $p = 3$.

81.5% parallel efficiency, which has more than 10% improvement compared with the sequential ordering scheme. Additionally, the cyclic ordering scheme for the treecode parallelization can be conveniently extended to other kernels and structures.

We also investigate how the parallel efficiency changes based on different choices of the treecode parameters, such as Taylor expansion order p, Maximum Acceptance Criterion θ, and maximum number of particles per leaf N_0. By studying plots of parallel efficiency against parameters as well as the time-error and (parallel efficiency)-error scatter plots, we conclude that the cyclic ordering scheme improves the parallel efficiency at various choices of the parameters, and also makes the parallel efficiency less sensitive to these parameter choices than the sequential ordering scheme for the tested case. We further show that the time-error scatter plot can help to select the most efficient treecode parameters for a desired error tolerance. Furthermore, the (parallel efficiency)-error scatter plot reveals the fact that when using a small number of particles per leaf N_0 (for saving CPU time), the parallel efficiency becomes more sensitive to the selection of treecode parameters. We will research this topic for further improvement in future work.

The current work can be further extended in the following directions. First, we plan to parallelize the processes for building the tree and computing moments, which will further improve the overall parallel efficiency by increasing the parallelizable fraction. Second, we are working toward the parallelization of a Poisson–Boltzmann equation solver based on the boundary element method; this uses treecode to efficiently compute the matrix–vector product Ax in each GMRES iteration. Due to its algorithmic simplicity and small memory requirements, treecode is a good candidate for GPU-based parallelization [26,44]. We note that recursion has been recently supported on GPUs, which will make our treecode implementation on such architectures more convenient. Third, the Coulomb interactions for the more accurate point multipole model (instead of the widely used point partial charge model) demand highly efficient calculations of N-body interactions [36,37], and may additionally benefit from the advances in this work. Finally, for some three-dimensional applications, e.g. in astrophysics, which have much larger numbers of particles, this approach of tree replication will rapidly limit scalability, which calls for parallelization using MPI domain decomposition. To this end, we implement a DD-parallelized treecode algorithms using the ORB trees. the numerical results show its scalability in memory as the problem size and number of tasks are simultaneously increased.

For dissemination to the greater science community, we published the cyclically parallelized treecode and the DD-parallelized treecode as open source software on GitHub (https://github.com/Jiahuic/treecode_parallel) under the General Public License (GNU); this software is maintained by Jiahui Chen, who was an SMU graduate student and is now as a postdoc at MSU.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work was supported by National Science Foundation, USA grant DMS-1418957, DMS-1819193, SMU University Research Council (URC), USA grant, SMU new faculty startup fund, and the SMU Center for Research Computing, USA. The authors thank Professor Robert Krasny for useful discussions.

References

[1] A.W. Appel, SIAM J. Sci. Stat. Comput. (1985).
[2] J.K. Salmon, M.S. Warren, G.S. Winckelmans, Int. J. Supercomput. Appl. 8 (1986) 129–142.
[3] A.G. Gray, A.W. Moore, in: T.K. Leen, T.G. Dietterich, V. Tresp (Eds.), Advances in Neural Information Processing Systems 13, MIT Press, 2001, pp. 521–527.
[4] L. Greengard, V. Rokhlin, J. Comput. Phys. 73 (2) (1987) 325–348.
[5] P. Li, H. Johnston, R. Krasny, J. Comput. Phys. 228 (10) (2009) 3858–3868.
[6] B. Engquist, L. Ying, Commun. Math. Sci. 7 (2009) 327–345.
[7] J. Tausch, Contemp. Math. 329 (2003) 307–314.
[8] T. Darden, D. York, L. Pedersen, J. Chem. Phys. 98 (12) (1993) 10089–10092.
[9] J. Barnes, P. Hut, Nature 324 (1986) 446–449.
[10] P.B. Callahan, S.R. Kosaraju, J. ACM 42 (1995) 67–90.
[11] L. Greengard, W. Gropp, Comput. Math. Appl. 20 (7) (1990) 63–71.
[12] L.F. Greengard, J. Huang, J. Comput. Phys. 180 (2) (2002) 642–658.
[13] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, G. Biros, Commun. ACM 55 (2012) 101–109.
[14] L. Ying, G. Biros, D. Zorin, H. Langston, Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC '03, ACM, New York, NY, USA, 2003, p. 14.
[15] B. Zhang, B. Lu, X. Cheng, J. Huang, N.P. Pitsianis, X. Sun, J.A. McCammon, Commun. Comput. Phys. 13 (2013) 107–128.

[16] J. Dubinski, New Astron. 1 (2) (1996) 133–147.
[17] U. Becciani, V. Antonuccio-Delogu, M. Gambera, J. Comput. Phys. 163 (1) (2000) 118–132.
[18] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, M. Taiji, Comput. Sci. - Res. Dev. 24 (2009) 21–31.
[19] J. Makino, Publ. Astron. Soc. Japan 56 (3) (2004) 521–531.
[20] M. Winkel, R. Speck, H. Habner, L. Arnold, R. Krause, P. Gibbon, Comput. Phys. Comm. 183 (4) (2012) 880–889.
[21] P. Gibbon, R. Speck, A. Karmakar, L. Arnold, W. Frings, B. Berberich, D. Reiter, M. Masek, IEEE Trans. Plasma Sci. 38 (9) (2010) 2367–2376.
[22] D. Malhotra, G. Biros, Commun. Comput. Phys. 18 (2015) 808–830.
[23] B. Zhang, B. Peng, J. Huang, N.P. Pitsianis, X. Sun, B. Lu, Comput. Phys. Comm. 190 (2015) 173–181.
[24] D.J. Hardy, M.A. Wolff, J. Xia, K. Schulten, R.D. Skeel, J. Chem. Phys. 144 (11) (2016).
[25] G. Lukat, R. Banerjee, New Astron. 45 (2016) 14–28.
[26] M. Burtscher, K. Pingali, An Efficient CUDA Implementation of the Tree-Based Barnes-Hut N-Body Algorithm, Elsevier Inc, 2011, pp. 75–92.
[27] Z.-H. Duan, R. Krasny, J. Comput. Chem. 22 (2) (2001) 184–195.
[28] K. Lindsay, R. Krasny, J. Comput. Phys. 172 (2) (2001) 879–907.
[29] Z.-H. Duan, R. Krasny, J. Chem. Phys. 113 (9) (2000) 3492–3495.
[30] R. Krasny, L. Wang, SIAM J. Sci. Comput. 33 (5) (2011) 2341–2355.
[31] A.J. Christlieb, R. Krasny, J.P. Verboncoeur, Comput. Phys. Comm. 164 (13) (2004) 306–310, Proceedings of the 18th International Conferene on the Numerical Simulation of Plasmas.
[32] H.A. Boateng, R. Krasny, J. Comput. Chem. 34 (25) (2013) 2159–2167.
[33] W. Geng, R. Krasny, J. Comput. Phys. 247 (2013) 62–78.
[34] M.L. Connolly, J. Mol. Graph. 3 (1985) 19–24.
[35] B. Lee, F.M. Richards, J. Mol. Biol. 55 (3) (1971) 379–400.
[36] P. Ren, J.W. Ponder, J. Phys. Chem. B 107 (24) (2003) 5933–5947.
[37] H.A. Boateng, J. Chem. Phys. 147 (16) (2017) 164104.
[38] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D. States, S. Swaminathan, M. Karplus, J. Comput. Chem. 4 (1983) 187–217.
[39] D.A. Perlman, D.A. Case, J.W. Caldwell, W.S. Ross, T.E. Cheatham, S. Debolt, D. Ferguson, G. Seibel, P. Kollman, Comput. Phys. Comm. 91 (1995) 1–41.
[40] J.K. Salmon, Parallel Hierarchical N-Body Methods (Ph.D. thesis), California Institute of Technology, 1991.
[41] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, MA, USA, 1994.
[42] https://github.com/barkm/n-body.
[43] M.F. Sanner, A.J. Olson, J.C. Spehner, Biopolymers 38 (1996) 305–320.
[44] W. Geng, F. Jacob, Comput. Phys. Comm. 184 (6) (2013) 1490–1496.