2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Polarization Energy on a Cluster of Multicores*

Jesmin Jahan Tithi and Rezaul A. Chowdhury
Department of Computer Science
Stony Brook University, Stony Brook, New York, 11790,USA
E-mail: {jtithi, rezaul}@cs.stonybrook.edu

***Abstract*—Computing the polarization energy between a ligand (i.e., a small molecule such as a drug molecule) and a receptor (e.g., a virus molecule) is of utmost importance in drug design. We have designed and implemented distributed-memory and distributed-shared-memory parallel algorithms for approximating GB-polarization energy (e.g., polar part of free energy of hydration) of protein molecules. This is an octree-based hierarchical algorithm, built on Greengard-Rokhlin type near-far decomposition of data points (i.e., atoms and points sampled from the molecular surface) for calculating the polarization energy of protein molecules using the surface based r^6-approximation of Generalized Born radii of atoms. We have shown that our implementations outperform state-of-the-art GB-polarization energy implementations, such as *Amber 12*, GBr^6, *Gromacs 4.5.3*, *NAMD 2.9* and *Tinker 6.0*. Using approximations, cache-efficient data structures and efficient load-balancing schemes, we achieve a speedup factor of ~ 400 w.r.t Amber with less than 1% error w.r.t. the naïve exact algorithm using as few as 144 cores (i.e., 12 compute nodes with 12 cores each) for molecules with as many as half a million atoms.**

***Keywords*-Polarization Energy, Generalized Born, Cluster of Multicores, Hybrid Parallelism.**

I. Introduction

Whenever a molecule comes under the influence of an electric field, its charge distribution is relaxed in response to that field. The energy associated with this relaxation is known as the polarization energy (E_{pol}). It is typically negative in quantity, as a relaxation leads to decrease in energy [27]. Electronic polarization plays a crucial role in drug design, discovery & design of new proteins, antivirus and antibiotics, protein-protein docking, molecular dynamics simulations for determining the molecular conformation with minimal total free energy, and so on.

The Poisson-Boltzmann [1], [15], [19], [25] model can be used to approximate E_{pol}. However, due to high computational costs Poisson-Boltzmann method is rarely used for large molecules such as proteins. Instead E_{pol} is approximated using the Generalized Born (GB) model [16], [17], [28] – a popular approximation model which considers solvent as a statistical continuum. However, computing E_{pol} naïvely even based on the GB model takes time quadratic in the number of atoms in the molecule, and thus it remains computationally expensive for large molecules. Hence, another level of approximation over the original GB-approximation is required in order to reduce its complexity below quadratic, and preferably to linear.

*This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575

An additional level of performance boost can be gained in GB-approximation by introducing parallelism in the computation [22]. In fact, multicore computers have already become the mainstream computing devices, and the number of cores in these devices is increasing rapidly. Not only that most of our desktops and laptops are already multicore computers, nowadays most modern supercomputers are also built as clusters of multicore machines. Before multicores became widely available, distributed-memory parallel algorithms were typically used in high performance parallel computing, and these algorithms were designed to use explicit distribution and communication of data among compute nodes. Even though multicore computers allow implicit communication among the cores through the memory hierarchy and the shared memory space, when run on clusters of multicores, distributed-memory algorithms typically require separate memory space for each core of the same compute node, and explicit communication among the cores. One natural way of avoiding the use of data replication and explicit communication among the cores of a compute node is to use hybrid algorithms – algorithms that use shared-memory parallelism inside each multicore node and distributed-memory parallelism across the nodes of the cluster. The goal is to reduce space usage (due to data replication) and communication time (due to explicit communication among threads) whenever possible.

The main contribution of this paper is a hybrid distributed-shared-memory parallel algorithm for approximating GB polarization energy on a cluster of multicores. We use a fast approximation scheme based on a hierarchical spatial decomposition of the molecule[1] [6], [7], and apply a Greengard-Rokhlin type near-far approximation scheme [13] on the decomposition. We also present detailed performance results of our approach. We show that it runs faster than other state-of-the-art implementations of GB polarization energy namely, *Amber 12* [9], *GBr*^{6} [35], *Gromacs 4.5.3* [18], *NAMD 2.9* [31], [34] and *Tinker 6.0* [29], and can handle molecules larger than most of them can process. We have also compared our hybrid algorithm with our own purely distributed-memory implementation of the same algorithm. We found that though for small molecules the hybrid algorithm runs slower, it outperforms the distributed-memory version as the size of the molecule increases.

This paper extends our prior work for shared-memory architectures [6], [7] to the distributed-shared-memory setting. The resulting algorithm has the following properties:

[1]consisting of atoms and points sampled from the surface of the molecule

- **Hybrid parallelism.** We use shared-memory parallelism inside each compute node and distributed-memory parallelism across compute nodes.
- **Cache- and space-efficient data structures.** We use *octrees* [20] for finding nonbonded atoms, which, unlike traditional *nonbonded lists* [30], always use space linear in the number of atoms in the molecule independent of any distance cutoff used, and are also known to be cache-friendly.
- **Space-independent speed-accuracy tradeoff.** The algorithm uses user-defined approximation parameters, and by tuning these parameters one can get a more accurate approximation of E_{pol} at the cost of increasing the running time and vice versa. Unlike traditional distance cutoff based methods, the space usage is independent of the values of the approximation parameters.
- **Load balancing.** Inside each compute node, we use dynamic load balancing based on efficient (fast and cache-efficient) randomized work-stealing [3], and across nodes, we use static load balancing in order to reduce the communication overhead.

The rest of the paper is organized as follows. In Section II we provide necessary background on polarization energy as well as on the data structures and algorithms we use. In Section III we describe related work on the estimation of polarization energy. Section IV presents our algorithms along with their theoretical complexity analysis. In Section V we present simulation results and a detailed comparison with other existing approaches namely, *Amber 12*, *GBr*^{6}, *Gromacs 4.5.3*, *NAMD 2.9* and *Tinker 6.0*. Finally, Section VI concludes the paper with some future research directions.

II. Background

In this section we first explain the mathematical expressions for estimating E_{pol}. Then we provide some background on the cache-efficient octree data structure, and the near-far approximation scheme from [6], [7] which we extend to the distributed-shared-memory setting.

Polarization Energy: The polarization energy of a molecule depends on the difference of potential of that molecule in solvent and gas-phase, and its charge density:

$$E_{pol} = \frac{1}{2}\int \emptyset_{reaction}(r).\rho(r), \qquad (1)$$

where $\emptyset_{reaction}$=$\emptyset_{solvant} - \emptyset_{gass-phase}$, and $\emptyset(r)$ and $\rho(r)$ are the electrostatic potential and charge density of the molecule, respectively.

In the GB-model, the polarization energy of a molecule is given by the following equation:

$$E_{pol} = \frac{1}{2}\left(1 - \frac{1}{\epsilon_{solv}}\right)\sum_{i,j}\frac{q_i \cdot q_j}{{f_{ij}}^{GB}}, \qquad (2)$$

where ${f_{ij}}^{GB} = \left[{r_{ij}}^2 + R_i R_j \exp^{\frac{-{r_{ij}}^2}{4R_iR_j}}\right]^{\frac{1}{2}}$, and ϵ_{solv} = solvent di-electric, r_{ij} = distance between atoms i and j, R_k and q_k $(k\ \epsilon\{i,j\})$ denote the Born radius and charge value of atom k, respectively.

The effective Born radius reflects how deep a charge is buried inside the molecule. The Born radius of an atom i, R_i shows the extent of interaction of the atom with a solvent when it is dissolved in that solvent. If the atom is close to the molecular surface, R_i is small. An atom with large R_i has a weaker interaction with the solvent.

To approximate Born radii and polarization energy, we have used Gaussian quadrature points sampled from the molecular surface. Gaussian quadrature attempts to obtain the best numerical estimate of an integral (e.g., molecular surface function) by picking optimal abscissas x_i to evaluate the function. Gaussian quadrature is considered to be optimal as it fits all polynomials exactly up to a certain degree [36]. The triangulation of Gaussian quadrature function of the molecular surface yields an estimation of molecular surface normal at triangulation vertices, and at Gauss quadrature numerical integrations points in each triangle's interior. A constant number of quadrature points per triangle are needed for high accuracy of the Born radii calculation.

The evaluation of Born radii is essentially based on the Coulomb field approximation [14], which assumes that the electric displacement is in the Columbic form. Using this approximation Born radii can be calculated as follows:

$$\tfrac{1}{R_i} = \tfrac{1}{4\pi}\int \tfrac{1}{|r-x_i|^4} d^3r,$$

where x_i represents the center of atom i.

We can obtain a discrete approximation of Born radii by applying Gaussian quadrature as shown in Equation 3 (known as r^4-approximation) [11]:

$$\frac{1}{R_i} \approx \frac{1}{4\pi}\sum_{k=1}^{N} w_k \frac{(r_k - x_i).\overrightarrow{n_k}}{|r_k - x_i|^4}, \qquad (3)$$

where r_ks denote N Gaussian quadrature points on the molecular surface, $\overrightarrow{n_k}$ is the unit outward surface normal at r_k, and w_k is a weight assigned to the quadrature point in order to achieve higher order of accuracy for small N. However, the following approximation of Born radius (known as r^6-approximation) shows better accuracy for spherical solutes, e.g., proteins [14]:

$$\frac{1}{{R_i}^3} = \frac{3}{4\pi}\int \frac{1}{|r_k - x_i|^6} \approx \frac{1}{4\pi}\sum_{k=1}^{N} w_k \frac{(r_k - x_i).\overrightarrow{n_k}}{|r_k - x_i|^6}. \qquad (4)$$

Octrees vs. Nblists: An Octree is a tree data structure that recursively and adaptively sub-divides the $3D$ space into 8 octants, and is often used as a container for rectilinear scalar field data. Octrees are very cache friendly because of their recursive nature. We use octrees to store the atoms in a molecule and the surface quadrature points. Once an octree

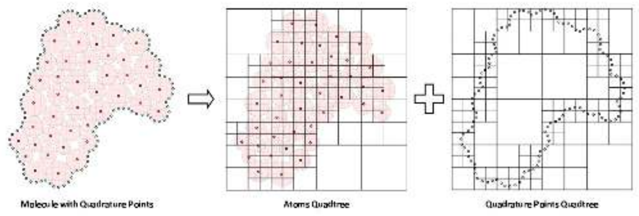


Figure 1: In the Born radius approximation algorithm two octrees are constructed: one for the atoms in the molecule, and the other for the quadrature points. Born radii of all atoms are approximated by recursively traversing both octrees simultaneously. For simplicity, the octrees are drawn as quadtrees.

is built, it can be used for any approximation parameter (similar to distance cutoff used in other molecular dynamics (MD) packages). Some existing MD packages, e.g., $Amber$, $NAMD$ and $Gromacs$ use `nblists` (nonbonded list) to represent interacting atom pairs. The size of the `nblist` of any given atom grows linearly with the number of atoms in the system, and cubically with the distance cutoff that truncates the non-bonded interactions. On the other hand, an octree uses space linear in the number of data points it holds, and its size does not change with the approximation parameter. Updating the `nblist` after the initial construction is costly and also not scalable with the distance cutoff. Often MD implementations that use `nblists` run out of memory for molecules with millions of atoms. For large cutoffs, octree is more space-efficient, update-efficient and cache-efficient compared to nblists [8].

Approximating Born Radii and GB Energy: This section gives a quick review of the approximation algorithms for Born radii and polarization energy calculation described in [6]. We use the same basic ideas of near-far approximation in our distributed and distributed-shared-memory algorithms, although we change the algorithms as well as the approximation schemes for efficient work-division. Let A be the set of atoms in a molecule, and Q be the set of quadrature points (denoted *q-points*) sampled from the molecular surface. First, two octrees T_A and T_Q for A and Q, respectively, are built, and then Born radii are approximated by traversing them simultaneously starting at their root nodes.

Approximate integrals (using Equation 4) are collected at appropriate internal nodes of T_A and atoms of A. Suppose at some point during this traversal we are at node $A \in T_A$ and node $Q \in T_Q$. Let r_A (resp. r_Q) be the radius of A (resp. Q). If A and Q are far enough, i.e., the distance between their centers, r_{AQ} is larger than $(r_A + r_Q)\left(\frac{(1+\epsilon)^{1/6}+1}{(1+\epsilon)^{1/6}-1}\right)$ for some user-defined approximation parameter $\epsilon > 0$, then the contribution of all q-points in Q to the Born radius integral of each atom in A can be approximated by treating A (resp. Q) as a single pseudo-atom (resp. pseudo q-point) centered at the geometric center of the atoms (resp. q-points) under it. These approximated contributions are collected in A. If A and Q are not far enough but at least one of them is a non-leaf, we recurse using the children of the non-leaf/non-leaves. If both are leaves, then we compute the contributions exactly using the atoms under A and the q-points under Q, and collect them in the respective atoms. Next, we traverse T_A top-down, and collect and add partial integrals from all ancestors of an atom to it. Finally, we compute the Born radii values from these accumulated values [6]. E_{pol} is approximated using similar techniques. The pseudo-code for the Born radii and E_{pol} calculation can be found in [6]. Note that the accuracy and speedup of these algorithms can be tuned by changing the approximation parameters, ϵ; increasing ϵ gives better speedup while sacrificing accuracy in results more and vice-versa.

III. Related Work

Octree-based hierarchical treecode algorithms have already been used for energetics computations. These algorithms are typically based on Barnes-Hut clustering [21] or the Fast Multipole Method (FMM) [4], and have been implemented for both serial and distributed-memory parallel machines to compute Coulomb, London, Lennard-Jones, H-bonds potentials [10], [33], polarized Coulomb interactions [24], Yukawa potential [37], etc.

A. Popular Parallel E_{pol} Implementations

The well-known *Amber 12* [9] package has an MPI-based distributed-memory implementation for GB-energy calculation. Amber also has a shared-memory parallel implementation of GB-energy which uses vectorization [32]. *Gromacs* [18] has OpenMP based shared-memory and MPI based distributed-memory implementations of E_{pol}. On the other hand, *NAMD* [31], [34] uses Charm++ [23] and MPI for its shared and distributed-memory implementations, respectively. *Tinker-6.0* [29] is also a well-known MD package which supports OpenMP based shared-memory parallelism. On the other hand, GBr^6 has a serial approximation algorithm that uses volume-based r^6-approximation of Born radii as opposed to our surface-based r^6-approximation. Note that all existing MD packages currently use either shared-memory or distributed-memory parallelism for computing GB-energy. Most of these MD packages support multiple GB-models such as HCT [17], STILL [16], OBC [28] etc.

IV. Our Contributions

Our main contributions in this paper are as follows:

- We have designed an efficient and scalable hybrid distributed-shared memory parallel algorithm for approximating Born radii and polarization energy. A number of different load balancing/work distribution schemes have been explored.
- We have analyzed the time complexity and scalability of the algorithm.

- We have implemented our algorithm with distributed-shared- & distributed-memory parallelism, and compared our implementations with five other state-of-the-art implementations of E_{pol}, namely, *Amber 12*, *GBr*^{6}, *Gromacs 4.5.3*, *NAMD 2.9* and *Tinker 6.0*, and showed that our implementations outperform all of them.

The major difference of our approach from algorithms presented in [6] is that we only traverse one octree instead of two, and hence the approximation scheme is also different. Figures 2 and 3 show our modified algorithms.

A. Load Balancing

There are basically two ways of load balancing in our distributed-shared- and distributed-memory algorithms:

- distribute only the work/computation (each process will have all the data),
- distribute both the data and work evenly among the processes (each process gets only a part of the data).

In the current paper, we have reported only the implementations in which we divide the work (each process has a complete set of data).

Load balancing on octree data structures has been discussed in [5]. We have used both static and dynamic load balancing schemes in our algorithms. We use static load balancing among the processes because static load balancing is more efficient and less costly than dynamic load balancing in this case. Our load balancing scheme works as follows:

- EXPLICIT STATIC LOAD BALANCING: Work is divided evenly among processes. The i^{th} process computes the Born radii and E_{pol} for the i^{th} segment of atoms and leaf nodes, respectively, from the atoms octree.
- IMPLICIT DYNAMIC LOAD BALANCING:We also ensure dynamic load balancing among the threads inside a compute node using the `cilk++` work-stealing scheduler [3].

Different Work Distribution Approaches: In the distributed/distributed-shared-memory algorithms, one can distribute the work of calculating Born radii and polarization energy among the processes or the cores of the compute nodes of a cluster, either by dividing the leaf nodes (NODE-BASED-WORK-DIVISION[2]) or by dividing the atoms (ATOM-BASED-WORK-DIVISION[3]). We have used `MPI` [12] and `cilk++` [26] to implement our distributed and distributed-shared-memory algorithms. We chose `cilk++` because our algorithms are mainly based on nested parallelism, and such recursive parallel algorithms can be implemented very easily in `cilk++`. Also, `cilk++`'s randomized work-stealing scheduler allows efficient parallel execution of these recursive divide-and-conquer algorithms. In the rest of the paper we will refer to our Hybrid distributed-shared implementation as $OCT_{MPI+CILK}$ and distributed implementation as OCT_{MPI} .

[2]Each compute node computes only for the leaves assigned to it.

[3]Each compute node computes only for the atoms assigned to it.

WORK DISTRIBUTION FOR BORN RADII CALCULATION: For Born radii calculation work can be divided by first dividing the atoms or nodes from any of the two octrees (atoms octree or quadrature points octree) evenly among the processes, and then assigning the job of computation on a particular segment of nodes or atoms to a particular process. To compute Born Radii, we distribute the work in two phases. Firstly, we evenly divide the leaf nodes from the quadrature points octree to the MPI processes. We assign the work of computing approximated integrals for the i^{th} segment of leaf nodes to the i^{th} MPI process. In the second phase (in PUSH-INTEGRALS-TO-ATOMS), we divide the atoms evenly among the processes, and the i^{th} MPI process computes the final Born Radii for the i^{th} segment of the atoms. Note that each MPI process only traverses the atoms octree, and for each leaf node of the quadrature points octree that has been assigned to it, it computes the approximated integrals. In another implementation, we divide the atoms in both of these phases, and each process traverses both octrees (T_A and T_Q), but computes only for those nodes and atoms that fall within its range.

WORK DISTRIBUTION FOR E_{pol} CALCULATION: For E_{pol} calculation, we first divide the leaf nodes of the atoms octree into P equal segments, where P is the number of MPI processes. Then we assign the work of computing the interaction of the i^{th} segment of leaf nodes with the entire atoms octree to the i^{th} MPI process. In this case, each process computes the interaction energy due to all leaf nodes assigned to it, either by considering them in parallel (in $OCT_{MPI+CILK}$) or by taking them one at a time (in OCT_{MPI}) while it traverses the other atoms octree. We refer to the work division that divides leaf nodes for Born radii and energy computation as the *node–node work division*.

Other combinations of work divisions (e.g., *atom–node*, *atom–atom*, *qpoint–node*, *node–atom*, etc.) are also possible, but the *node–node* type work division scheme performed better than other alternatives in the experiments we conducted. We have observed that *atom–node* work division takes slightly more time than the purely node based (*node–node*) work division. Moreover, in *node–node* work division, only leaf nodes (of one octree) are considered during interaction computation (with other octree) which leads to less approximation compared to approximating at internal nodes. For this reason, the *node–node* work division performs better than others with respect to the percentage of error in the energy value. The error of atom based work division keeps changing with the number of processes even when the approximation parameters are kept fixed, because different division boundaries can split the same treenode differently in atom-based work division. On the contrary, for node-based work division, the error is constant for constant parameters, because each compute node always gets a full treenode, and

APPROX-INTEGRALS(A, Q) (Here A denotes a node from atoms octree, and Q denotes a leaf node from quadrature points octree. For each atom a under the subtree rooted at the given node A in the atoms octree, this function approximates $\sum_{q \in Q} w_q \frac{(\mathbf{p}_q - \mathbf{p}_a) \cdot \mathbf{n}_q}{|\mathbf{p}_q - \mathbf{p}_a|^6}$. By $\mathbf{p}_a = \langle x_a, y_a, z_a \rangle$ we denote the center of an atom a, while by $\mathbf{p}_q = \langle x_q, y_q, z_q \rangle$, w_q and $\mathbf{n}_q = \langle nx_q, ny_q, nz_q \rangle$ we denote the location of a quadrature point q, weight assigned to q, and the unit outward normal on the molecular surface at q, respectively. By r_A (resp. r_Q) we denote the radius of the smallest ball that encloses all atom centers (resp. integration points) under A (resp. Q). The distance between the geometric centers of A and Q is given by $r_{A,Q}$. We also assume $\widetilde{nx}_Q = \sum_{q \in Q} w_q nx_q$. Similarly for $\widetilde{ny}_Q$ and $\widetilde{nz}_Q$. Each atom a has a field s_a, and each node A in the atoms octree has a field s_A, all of which are initialized to zero. The approximated sum is added to s_A provided A and Q are far enough in space so that the sum can be approximated reasonably well (controlled by an approximation parameter $\epsilon > 0$). Otherwise the sums are computed recursively and added to the s field of appropriate descendants of A. By CHILD(A) we denote the set of non-empty octree nodes obtained by subdividing node A.)

```
1) if $r_{A,Q} - (r_A + r_Q) > 0 \ \wedge \ \frac{r_{A,Q} + (r_A + r_Q)}{r_{A,Q} - (r_A + r_Q)} > (1+\epsilon)^{\frac{1}{6}}$ then $s_A = s_A + \frac{\widetilde{nx}_Q \cdot (x_Q - x_A) + \widetilde{ny}_Q \cdot (y_Q - y_A) + \widetilde{nz}_Q \cdot (z_Q - z_A)}{(r_{A,Q})^6}$   {far enough to approximate}
2) elif LEAF($A$) then   {too close to approximate; compute exact value}
        for each atom $a \in A$ do
            for each quadrature point $q \in Q$ do
                $s_a = s_a + \frac{w_q (nx_q \cdot (x_q - x_a) + ny_q \cdot (y_q - y_a) + nz_q \cdot (z_q - z_a))}{(r_{a,q})^6}$
3) else $\forall A' \in$ CHILD(A) : APPROX-INTEGRALS( $A'$, $Q$ )
```

PUSH-INTEGRALS-TO-ATOMS(A, s , s_{id}, e_{id}) (A is a node in the atoms octree, and $s = \sum_{A' \in \text{ANCESTORS(A)}} s_{A'}$. This function pushes $s + s_A$ to each descendant of A. If A is a leaf it computes the Born radius of each atom $a \in A$ using $s + s_A + s_a$. Here, s_{id} and e_{id} denote the start_id and end_id of the atoms assigned to a process.)

```
1) if LEAF($A$) then $\forall a \in A$ that falls in $[s_{id}, e_{id}]$: $R_a = \max \left\{ r_a, \ \left( \frac{s_a + s + s_A}{4\pi} \right)^{-\frac{1}{3}} \right\}$   {compute Born radii of A's atoms}
2) else $\forall A' \in$ CHILD(A) : PUSH-INTEGRALS-TO-ATOMS( $A'$, $s + s_A$ )   {push integrals to A's descendants (parallel)}
```

Figure 2: Octree-based algorithm for r^6-approximation of Born radii.

APPROX-E_{pol}(U, V) (For two given nodes U and V in the atoms octree $\mathcal{T}_{\mathcal{A}}$ where, V is a leaf, approximate the part of E_{pol} resulting from the interaction between the set of atoms under U and V. By r_U we denote the radius of the smallest sphere that encloses all atom centers under U. For any atom $u \in U$, its center, radius, charge and Born radius are given by (x_u, y_u, z_u), r_u, q_u and R_u, respectively. For $0 \le k < M_\epsilon = \log_{1+\epsilon}(R_{max}/R_{min})$, $q_U[k] = \sum_{(u \in U) \ \wedge \ (R_u \in [R_{min}(1+\epsilon)^k, R_{min}(1+\epsilon)^{k+1}))} q_u$, where R_{min} and R_{max} are the minimum and the maximum Born radius among all atoms in $\mathcal{A}$. By CHILD(A) we denote the set of non-empty octree nodes obtained by subdividing node A.)

```
1) if LEAF($U$) then return $-\frac{\tau}{2} \sum_{(u \in U) \ \wedge \ (v \in V)} q_u q_v \Big/ \sqrt{r_{uv}^2 + R_u R_v e^{-r_{uv}^2 / 4 R_u R_v}}$   {exact value}
2) elif $r_{U,V} > (r_U + r_V)\left(1 + \frac{2}{\epsilon}\right)$ then return $-\frac{\tau}{2} \sum_{0 \le i,j < M_\epsilon} q_U[i] \cdot q_V[j] \Big/ \sqrt{r_{UV}^2 + R_{min}^2 (1+\epsilon)^{i+j} e^{-r_{UV}^2 / 4 R_{min}^2 (1+\epsilon)^{i+j}}}$   {approximate}
3) else return $\sum_{U' \in \text{CHILD}(U)}$ APPROX-$E_{\text{pol}}$( $U'$, $V'$ )   {recurse on U (parallel)
```

Figure 3: Octree-based algorithm for approximating E_{pol} from Born radii.

hence the approximation does not change with the change of division boundaries. We have also observed the same trend of errors in Gromacs that also uses atom based work division techniques.

Dynamic load balancing among threads: In our distributed-shared-memory algorithm, inside each compute node multiple threads (or cores) are used to accomplish the work assigned to a process. The `cilk++` runtime system provides dynamic load balancing among threads using a randomized work-stealing scheduler [3]. In `cilk++` work-stealing scheduler, each thread maintains a double ended queue (deque) to store its outstanding work/tasks and adds the newly generated work to the bottom of the queue. On the other hand, when a thread runs out of work, it chooses a random victim thread and steals work from top of the victim's queue which helps to reduce inter-thread communication and guarantees progress [2].

B. Algorithm

Figure 4 shows a sketch of our Hybrid distributed-shared-memory parallel octree based GB-radii and E_{pol} algorithms, where p denotes the number of threads running concurrently in shared-memory and is upper bounded by the number of cores in a single compute node. If the distributed-shared-memory algorithm runs with P processes, each running p threads internally, the corresponding distributed-memory algorithm should run P x p MPI processes to achieve the same level of parallelism (using the same number of cores). It is important to design hybrid (distributed-shared) algorithms and explore their performance for the following reasons.

- Most modern supercomputers are networks of multi-cores, and hence the future computation model is likely to be of distributed-shared-memory type.
- A purely distributed-memory approach typically requires more memory than its distributed-shared-memory counterpart.[4]
- Running two threads on the same compute node (multi-core machine) incurs less communication overhead than running two single threaded processes on two different compute nodes.
- No distributed-shared-memory implementation of GB-energy is available yet.

Suppose, in a shared-memory algorithm k threads share the same data of size s. Now if we launch these k threads as k different processes as in a distributed-memory setting, each process will require a separate copy of the same data occupying ks space in total. As long as this ks data fits in the shared-cache/main memory, the speedups from both distributed and distributed-shared memory approaches should be comparable. However, as k independent processes (distributed) use k times more memory than used by one process with k threads (shared), at some point, the distributed-shared-memory algorithm should outperform the distributed-memory algorithm. This happens when the input becomes so

[4]Distributed memory implementations are typically designed to replicate data instead of sharing.

DISTRIBUTED/DISTRIBUTED-SHARED-MEMORY OCTREE BASED GB-POLARIZATION ENERGY COMPUTATION ALGORITHM (Suppose, we have P processes, each of which is running p threads internally. Therefore, if $p = 1$, it's a purely distributed approach and if $p > 1$, it's a distributed-shared approach. We first divide the work among the processes as evenly as possible. Inside each node, work is further distributed among multiple threads dynamically by the `cilk++` framework.)

1) Each compute node builds atoms-octree, T_A and quadrature-points-octree, T_Q independently.
2) For $1 \le i \le P$ [$in\ parallel$], the i^{th} process calculates the approximated integrals due to the i^{th} segment of leaf nodes from the quadrature points octree by traversing T_A using the APPROX-INTEGRALS algorithm. {*Node based work division*}.
3) Each process gathers the partial approximated integrals due to other segments of leaf nodes computed by other processes using MPI_Allreduce.
4) For $1 \le i \le P$ [$in\ parallel$], the i^{th} each process calls the PUSH-INTEGRALS-TO-ATOMS function and computes final Born radii for the i^{th} segment of atoms.
5) Each process gathers the Born radii of other segments of atoms from other processes.
6) Each process traverses T_A, and for $1 \le i \le P$ [$in\ parallel$], the i^{th} process calculates partial energy by computing the one-to-one interactions of the i^{th} segment of leaf nodes from T_A on other nodes of T_A. {*Node based work division*}.
7) The master process accumulates partial energy values form step 6 and generates the final E_{pol}.

Figure 4: Octree based distributed- and distributed-shared-memory algorithm.

large that the ks data does not fit into the shared-cache/main memory or incurs severe memory overhead (page fault/cache misses) causing a slowdown of the program. Moreover, the typical cost of communication among k threads in shared-memory $<$ (is less than) cost of communication among k processes on a single compute node/socket $<$ cost of communication among k processes on different sockets or computing nodes across the cluster. This also implies that as we increase the number of processes, the overhead of purely distributed algorithm will be more than the distributed-shared-memory algorithm. We have also observed similar trends in our experiments.

C. Analysis of Time Complexity

In this section, we present the time complexity analysis of our distributed/distributed-shared-memory octree-based algorithms. We have used complexity results proved in [6] and [7] for this analysis. Let P be the number of MPI processes, and p be the number of threads running internally inside each process. Let, the molecule has M atoms in it.

Computational Cost, T_{comp}:

Step 1: Each process builds octrees from atoms and quadrature points which takes $O(M \log M)$ time (assuming the number of Gaussian quadrature points, $m = O(M)$) [6]. Once the octrees have been built, we can approximate for any ϵ (recall that ϵ is an approximation parameter) without reconstructing them. Moreover, for drug-design and docking where we need to place the ligand at thousands of different positions w.r.t. the receptor, we can move the same octree to different positions or rotate it as needed by multiplying with proper transformation matrices, and then recompute the energy values. Therefore, we can consider the octree construction cost as a pre-processing cost and ignore it.

Step 2: Each process calculates the Born radii by traversing the atoms octree starting at the root node. The i^{th} process computes only for the i^{th} segment of leaf nodes from the quadrature points octree using the APPROX-INTEGRALS algorithm. Since each process gets approximately $\lceil M/P \rceil$ atoms, and inside each process each of the p cores/threads again does approximately $\frac{\lceil M/P \rceil}{p}$ part of the work, it costs $O((\frac{1}{\epsilon^3}(\frac{M}{P}\frac{1}{p} + \log \frac{M}{P})) = O((\frac{1}{\epsilon^3}(\frac{M}{P}\frac{1}{p} + \log M))$ time as $M >> P$ (using results from [7]).

Step 4: Each process calls PUSH-INTEGRALS-TO-ATOM, and the i^{th} process calculates Born radii only for the i^{th} segment of atoms. Traversing the entire tree takes $O(M \log M)$ time but each process traverses only that part of the tree that falls in its range. Eventually each thread traverses approximately $O(\frac{1}{P}(\frac{1}{p}))$ fraction of the tree. Therefore, this function will take $O(\frac{1}{P}(\frac{1}{p}(M \log M)))$ time.

Step 6: Each process traverses T_A, and the i^{th} process calculates partial energy by computing the one-to-one interactions of the i^{th} segment of leaf nodes from T_A with other nodes of T_A. Since each process gets $\lceil 1/P \rceil$ fraction of the total number of leaf nodes from the atoms-octree containing approximately $\lceil M/P \rceil$ atoms, each core/thread will get around $\frac{\lceil M/P \rceil}{p}$ of the atoms for computation. Hence, this step will take $O(\frac{1}{P}(\frac{1}{\epsilon^3}(\frac{M}{p} + 1) \log M))$ time (using results from [7]).

Therefore, the total computation time is, $T_{comp} = O\left(\frac{1}{P}\frac{1}{\epsilon^3}(\frac{M}{p} + 1) \log M\right)$.

Communication cost T_{comm}:

Step 3 & 5: Each process gathers the approximated integrals and Born radii of other segments from other processes. It takes $O(t_s \log P + t_w \frac{M}{P}(P-1))$ time, where t_s is the startup time and t_w is the message passing time per word (costs for MPI primitives can be found in Table 4.1 of [12]).

Step 7: The master process accumulates partial energy values from Step 6 using *MPI_Allreduce* and generates the final E_{pol} which takes $O(t_s \log P + t_w(P-1))$ time.

Therefore, the total parallel time, $T_p = T_{comp} + T_{comm}$
$$= O\left(\frac{1}{P}\frac{1}{\epsilon^3}(\frac{M}{p} + 1) \log M + t_s \log P + t_w \frac{M}{P}(P-1)\right)$$
$$= O\left(\frac{1}{Pp}\frac{1}{\epsilon^3} M \log M + t_w M\right).$$

Attribute Name	Property
Processors	3.33 GHz-Hexa-Core 64-bit Intel-Westmere
Cores/node	12
RAM size and speed	24 GB, 1333 MHz
Cluster Interaction Type	InfiniBand, fat-tree topology, 40Gb/s p2p bandwidth
Cache	12 MB $L3$, 64 KB private $L1$, 256 KB private $L2$
Operating System	Linux CentOS 5.5.
Parallelism Platform	Intel Cilk-4.5.4, MPI (MVAPICH2/1.6)
Optimization parameter	-O3

Table I: Simulation Environment

V. Simulation Results

All experiments included in this section were performed on the Lonestar4 computing cluster located at the Texas Advanced Computing Center (TACC). All algorithms were tested on ZDock Benchmark Suite-2.0 containing 84 complexes (168 proteins) both in bound and unbound states. We used proteins from the bound dataset only. The number of atoms per protein varied from around 400 to 16,000. Important properties of the simulation environment are summarized in Table I.

We have compared three different octree based implementations, namely, the shared-memory, distributed-memory, and distributed-shared-memory implementations with GBr^6 [35], and the GB-polarization energy implementations from four existing well-known Molecular Dynamics Packages, namely, *Gromacs 4.5.3* [18], *NAMD 2.9* [31], [34], *Amber 12* [9] and *Tinker 6.0* [29]. Table II summarizes some important properties of these programs. We have also reported the running times and energy values computed by the naïve serial implementations of Equations 2 and 4.

Package	GB-Model	Parallelism
Gromacs 4.5.3 [18]	HCT [17]	Distributed (MPI)
NAMD 2.9 [34]	OBC [28]	Distributed (MPI)
Amber 12 [9]	HCT	Distributed (MPI)
Tinker 6.0 [29]	STILL [16]	Shared (OpenMP)
GBr^6 [35]	STILL	Serial
Name	GB-Model	Parallelism
OCT_{CILK}	STILL	Shared (`cilk++`)
OCT_{MPI}	STILL	Distributed (MPI)
$OCT_{MPI+CILK}$	STILL	Distributed (MPI+`cilk++`)
Naïve	STILL	Serial

Table II: Packages with GB models and types of parallelism used.

A. Dealing with NUMA Effect

Note that to reduce the impact of *NUMA* (Non-uniform memory architecture) on Intel machines, we ran all the MPI programs with *ibrun tacc_affinity*, which is basically a wrapper around the *mpirun* or *mpiexec*, and it fixes the affinity of the processes to the cores, sockets and caches to reduce overall cache misses. On the other hand, `cilk++` does not provide any thread affinity manager. The `cilk++` work-stealing scheduler allows a thread to steal from any other thread. However, by stealing the oldest entry from a deque (least recently used data), it tries to reduce the number of cache misses. On Lonestar4, each machine was dual socket, and we launched one process with 6 threads on each socket for the $OCT_{MPI+CILK}$ program, which bounded those 6 threads only to one socket and alleviated the NUMA effect.

B. Scalability

Figures 5 and 6 show the scalability of our OCT_{MPI} and $OCT_{MPI+CILK}$ implementations from which we observe how the running time decreases and speedup increases with the number of cores. We ran this experiment on the Blue Tongue Virus (BTV) that has 6 million atoms and more than 3 million quadrature points. Since for smaller number of cores (or processes), each core needs to handle a comparatively larger data segment, the segment may not fit in the cache fully at the same time leading to more cache misses. However, as the number of cores or processes increases, because of the balanced work division, each core will work only on a smaller portion of data which can easily fit into the cache. For OCT_{MPI} program we ran 12 processes in each compute node, and for $OCT_{MPI+CILK}$ program we ran 2 processes each with 6 threads each. For each configuration, we ran all programs 20 times and plotted the minimum and maximum running times in the Figure 6. We observe that the minimum running time of $OCT_{MPI+CILK}$ is always smaller than the minimum running time of OCT_{MPI} after the core count reaches 180, whereas we always (independent of core count) see the opposite for the maximum running times. As the OCT_{MPI} program has 6 times more processes than $OCT_{MPI+CILK}$, the communication overhead of OCT_{MPI} was more than $OCT_{MPI+CILK}$. Similarly, the memory overhead was also more in OCT_{MPI}. For these reasons $OCT_{MPI+CILK}$ eventually ran faster than OCT_{MPI} . For BTV, when run on a single node with 12 cores, $OCT_{MPI+CILK}$ (2 processes, each with 6 threads) took approximately $1.4GB$ of memory, whereas $OCT_{MPI+CILK}$ (12 processes, each with 1 thread) occupied $8.2GB$, which is 5.86 times more than that of $OCT_{MPI+CILK}$ (as expected). This ratio continues to hold as we increase the number of compute nodes.

C. Running Time and Speedup

Next we ran OCT_{MPI} and $OCT_{MPI+CILK}$ on a 12-core machine for the ZDock benchmark molecules, and compared their performance with that of OCT_{CILK}. Note that the algorithms underlying OCT_{MPI} and $OCT_{MPI+CILK}$ were different from the one used by OCT_{CILK}. All these algorithms were run with approximation parameters set to 0.9 (Born Radii) and 0.9 (E_{pol}), respectively. We used approximate math for computing square root and power functions. No vectorization was used. We observed that OCT_{CILK} showed better performance than both OCT_{MPI} and $OCT_{MPI+CILK}$ for molecules with less than 2500 atoms, since for small molecules the communication cost dominated computation cost. The OCT_{MPI} implementation was significantly faster than OCT_{CILK} for molecules with greater than 2500 atoms, because for larger molecules computation costs beaten communication cost, and the differences in running times increased with the size of the molecules. The OCT_{MPI} implementation was also slightly faster than $OCT_{MPI+CILK}$ for molecules with less than 7500 atoms. After molecule size 7500, both OCT_{MPI} and $OCT_{MPI+CILK}$ showed similar performance. As OCT_{MPI} was using almost 6 times more memory than $OCT_{MPI+CILK}$, the difference in performance diminishes with the size of the molecule. MPI turns out to be

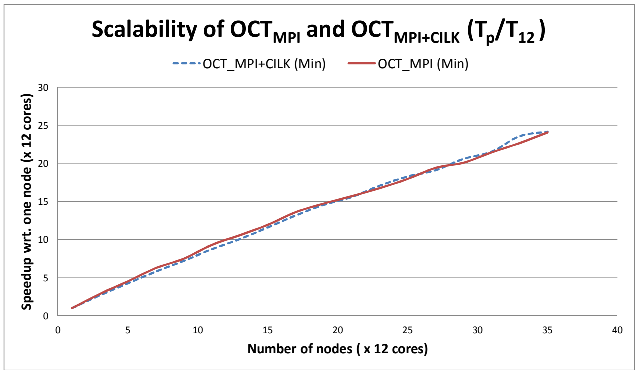


Figure 5: Speedup w.r.t. running time on one node (12 cores).

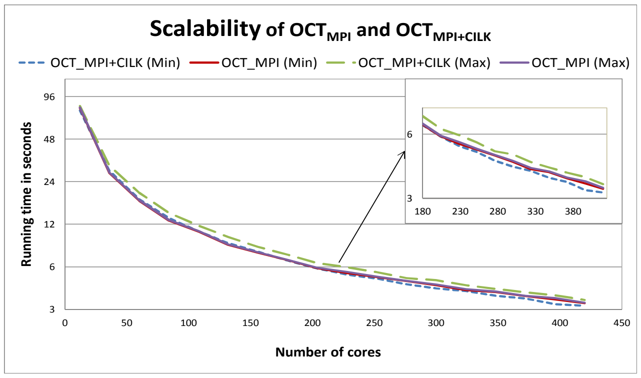


Figure 6: Scalability with increasing number of cores.

more optimized compared to the `cilk++` implementation[5] and `cilk++` does not maintain thread affinity. There is an additional overhead of interfacing cilk++ and MPI. These overheads of $OCT_{MPI+CILK}$ were prominent for smaller molecules and became less dominant as the size of the molecule increased.

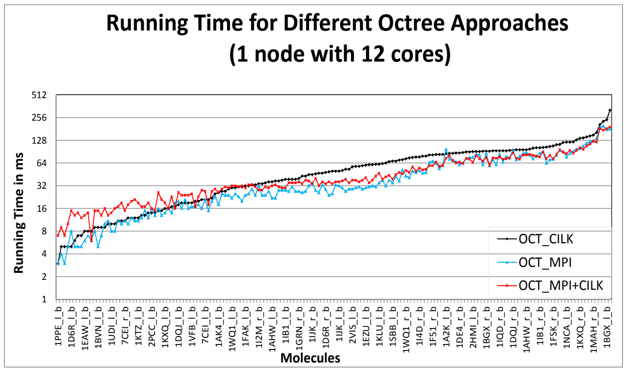
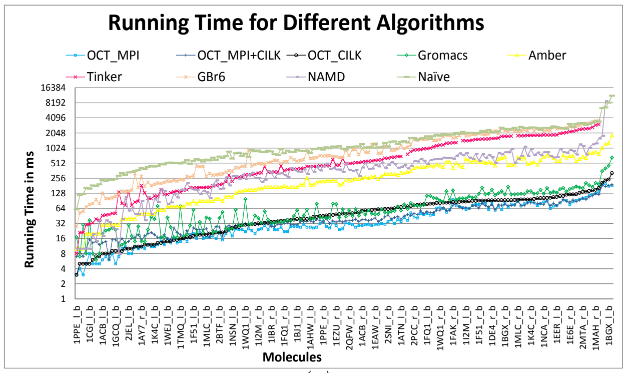


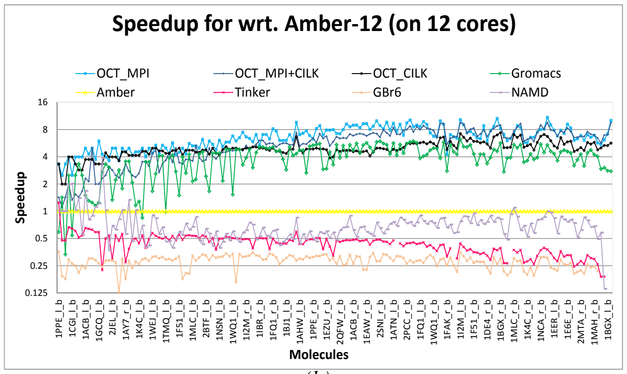
Figure 7: Performance Comparison of Different Octree Based Algorithms (Results are sorted by the OCT_{CILK} time).

Note that Gromacs also has a shared-memory implementation of GB-energy, and we observed that for Gromacs, too, the distributed-memory implementation was slightly faster

[5]We have used cilk-4.5.4, which is a predecessor of Intel cilk plus, and Intel cilk plus is likely to be much better optimized than cilk-4.5.4.



(a)



(b)

Figure 8: Performance Comparison of Different Algorithms. Results are sorted by molecule size.

than the shared-memory implementation. Hence, in the rest of this section, we only compare the MPI based distributed-memory implementation of Gromacs.

For comparison purposes, we ran all programs mentioned in Table II on a 12-core machine (single compute node). For the distributed implementations (NAMD, Gromacs, Amber, OCT_{MPI}), we ran 12 different MPI processes on these 12 cores. For NAMD we were not able to find any way to compute only the GB-energy. So, we first computed the total electrostatic potential with GB energy turned on, and then computed the electrostatic energy with GB energy turned off, and took the difference to retrieve actual GB energy. We also took the difference of running times of these two runs to get the time of GB energy computation. We took the average of 10 runs to reduce noise. Figure 8 shows the performance of different algorithms. From the plot of running times for GB-energy (including Born radii), we observe that overall OCT_{MPI} and $OCT_{MPI+CILK}$ perform the best among all algorithms. The differences in performance among Gromacs, OCT_{MPI} and $OCT_{MPI+CILK}$ become prominent as the size of the molecule increases. On the other hand, *Amber* was much slower than both OCT_{MPI} and Gromacs but faster than NAMD, Tinker and GBr^6. Experiments show that Tinker is slightly faster than GBr^6. We can get a glimpse of the speedup achieved by these programs on 12 cores of one compute node (1 core for GBr^6) compared to *Amber*. Figure 8 (b) shows that OCT_{MPI} achieves a speedup of approximately 11 w.r.t. *Amber* for a molecule of size 16,301 using only 12 cores, whereas Gromacs achieves a speedup of $\sim$ 2.7 for the same molecule (although the maximum speedup achieved by Gromacs is 6.2 for a molecule with 2260 atoms). The maximum speedup achieved by *NAMD*, Tinker and GBr^6 for the ZDock benchmark molecules are 1.1, 2.1 and 1.14, respectively.

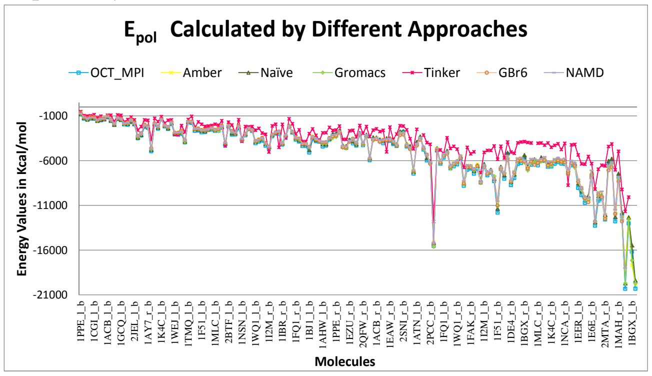
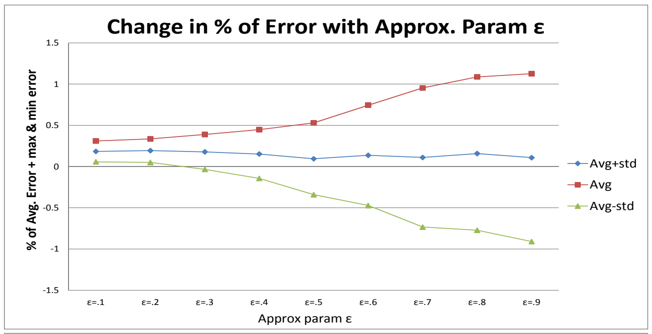


Figure 9: Energy Value Computed by Different Algorithms.



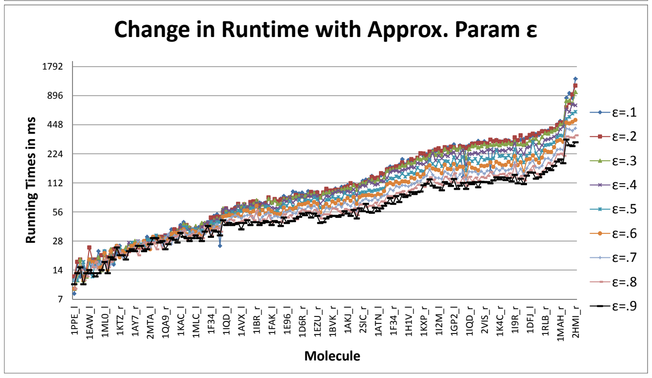


Figure 10: Performance Change of $OCT_{MPI+CILK}$ with Approximation Parameter; Born Radius ϵ is fixed at 0.9 and E_{pol} ϵ varies.

D. Energy Value

Figure 9 plots the GB-energy values for the ZDock benchmark molecules calculated by different algorithms mentioned in Table II. The energy values computed by Amber, GBr^6, Gromacs, NAMD and OCT_{MPI} match closely with GB-energy computed by the naïve approach. Energy values reported by Tinker were around 70% of the naïve energy. All octree based algorithms reported approximately the same energy value. We have observed that Tinker and GBr^6 do not work for larger molecules ($> 12k$ and $> 13k$ respectively) as they run out of memory.

E. Change in Error and Running Time with Approximation Parameter

Recall that the octree based algorithms are tunable, because we can change the error in result by changing the approximation parameters. An increase in approximation parameter ϵ increases error in energy value and decreases running time. However, for small molecules, running times do not depend on ϵ at all. Figure 10 shows the impact of approximation parameter on our distributed-shared-memory algorithm's percentage of error in energy value and running time. The distributed-memory algorithm also follows the same trend. For this experiment, we kept the approximation parameter of Born Radii calculation fixed at 0.9 and varied the approximation parameter of E_{pol} from 0.1 to 0.9. We ran the $OCT_{MPI+CILK}$ implementation on all protein molecules of the ZDock benchmark suite. Approximate math was turned "off". Turning approximate math "on" shifted the error by $4-5\%$ and decreased the running times by a factor of 1.42 on average (Figure 7 vs. Figure 10). We collected the average and standard deviation of percentage of error for E_{pol}, and plotted these avg. $\pm$ std. for all molecules.

F. Scalability with Larger Molecule

We also ran all octree-based implementations and *Amber* on the Cucumber Mosaic Virus (CMV) shell consisting of 509,640 atoms and 1,929,128 quadrature points. GBr^6 and Tinker ran out of memory for CMV. We were able to run Gromacs and NAMD on CMV only for cutoff values up to 2 and 60, respectively, which are not reasonable cutoff values for such a large molecule. For CMV, OCT_{MPI} and $OCT_{MPI+CILK}$ achieved a speedup of more than $400-500$ using only 12 cores of a single compute node and $300-400$ times speedup using 144 cores (12 compute nodes each running 12-threads internally) w.r.t. Amber, while the errors w.r.t. the naïve energy were still less than 1% [6]. Note that we get such a high speedup because of three levels of acceleration: (a) from parallelism, (b) from two levels of approximations in calculations (in Born Radii and E_{pol}), and (c) from using the cache-friendly octree data structure.

To summarize, our octree based polarization energy approximation algorithms run faster than *Amber*, Gromacs, NAMD, Tinker and GBr^6, and can handle molecules with millions of atoms which cannot be handled by most of the other implementations. The octree-based approaches show

[6]At present, Amber does not support concurrent execution of more than 256 cores.

Program	12 Cores (Time)	144 Cores (Time)	Speedup wrt Amber using 12 Cores	Speedup wrt Amber using 144 Cores	Energy Value Kcal/Mol (10^6)	% of Difference with Naïve
OCT$_{\text{CILK}}$	12.5s	X	187	X	-1.48	-0.95
Amber	39min	3.3min	1	1	-1.44	2.2
OCT$_{\text{MPI+CILK}}$	4.8s	0.61s	488	325	-1.47	-0.07
OCT$_{\text{MPI}}$	4.5s	0.46s	520	430	-1.47	-0.07

Figure 11: Scalability on a large molecule (Cucumber Mosaic Virus shell).

very good speedup and scalability with the number of cores and molecule size.

VI. Conclusion

In this work we have presented a hybrid distributed-shared-memory parallel octree based approximation algorithm for approximating polarization energy of protein molecules, and provided detailed performance comparison with Gromacs, NAMD, Amber, Tinker and GBr^6. We have shown that our octree-based approaches perform the best among all and achieve a speedup of ~ 400 for molecules with half a million atoms w.r.t. to the popular MD package Amber. We have also shown that the distributed-shared-memory implementation of our algorithm performs slightly better than the distributed-memory implementation for larger molecules. We believe that distributed-shared-memory parallelism is the right approach for implementing high performance MD simulations. We also believe that octree is the right data-structure to use in MD packages instead of nonbonded lists that cause most MD packages to run out of memory for very large molecules. Although our octree based algorithms perform better than others without explicit dynamic load balancing (except the one provided by `cilk++`), we are planning to incorporate explicit dynamics load balancing techniques such as work-stealing to improve the performance even further. Distributing data as well as computation is also an interesting approach to explore.

VII. Acknowledgement

We thank Qin Zhang for his help in preparing the input files. We also thank Prof. Chandrajit L. Bajaj for his support and advice as always. We are thankful to Mark Abraham, Justin Lemkul and Szilárd Páll for their help with Gromacs, and to Michael Schnieders for helping with Tinker.

References

[1] N. Baker, M. Holst, and F. Wang. Adaptive multilevel finite element solution of the Poisson-Boltzmann equation II: Refinement at solvent-accessible surfaces in biomolecular systems. *Journal of Computational Chemistry*, 21(15):1343–1352, 2000.

[2] R. Blumofe et al. Cilk: An efficient multithreaded runtime system. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp: 207–216, 1995.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[4] J. Board et al. Accelerated molecular dynamics simulation with the parallel fast multipole algorithm. *Chemical Physics Letters*, 198(1):89–94, 1992.

[5] P . Campbell et al. Dynamic octree load balancing using space-filling curves. *Technical Report, Department of Computer Science, Williams College*, 2003.

[6] R. Chowdhury and C. Bajaj. Multi-level grid algorithms for faster molecular energetics. *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pp: 147–152, 2010.

[7] R. Chowdhury and C. Bajaj. Multi-level grid algorithms for faster molecular energetics. *Journal Version (under review)*, 2013.

[8] R. Chowdhury et al. Space-efficient maintenance of nonbonded lists for flexible molecules using dynamic octrees. *Journal Version (under review)*, 2013.

[9] D. Case et al. AMBER 12. *University of California, San Francisco.*, 2012.

[10] Z. Duan and R. Krasny. An adaptive treecode for computing nonbonded potential energy in classical molecular systems. *Journal of Computational Chemistry*, 22(2):184–195, 2001.

[11] D. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21(6):1129–1148, 1985.

[12] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing* (2nd Edition). Addison-Wesley, 2003.

[13] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal Of Computational Physics*, 135(2):280–292, 1997.

[14] T. Grycuk. Deficiency of the Coulomb-field approximation in the Generalized Born model: An improved formula for Born radii evaluation. *The Journal of Chemical Physics*, 119(9):4817–4826, 2003.

[15] M. Gilson, M. Davis, B. Luty, and J. McCammon. Computation of electrostatic forces on solvated molecules using the Poisson-Boltzmann equation. *The Journal of Physical Chemistry*, 97(14):3591–3600, 1993.

[16] R. Hawley, W. Still, A. Tempczyk, and T. Hendrickson. Semianalytical treatment of solvation for molecular mechanics and dynamics. *Journal of the American Chemical Society*, 112(16):6127–6129, 1990.

[17] G. Hawkins, C. Cramer, and D. Truhlar. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges from a dielectric medium. *The Journal of Physical Chemistry*, 100(51):19824–19839, 1996.

[18] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, 2008.

[19] M. Holst, N. Baker, and F. Wang. Adaptive multilevel finite element solution of the Poisson-Boltzmann equation I: Algorithms and examples. *Journal of Computational Chemistry*, 21(15):1319–1342, 2000.

[20] C. Jackins and S. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, 1980.

[21] I. Jeffrey and V. Okhmatovskil. Effect of multilayered substrate on the Barnes-Hut center-of-charge clustering approximation: Half-space case study. *Proceedings of the 12th IEEE Workshop on Signal Propagation on Interconnects*, pp. 1–4, 2008.

[22] C. Janssen and I. Nielsen. *Parallel Computing in Quantum Chemistry*. CRC, 2008.

[23] L. Kale. Charm++. *Encyclopedia of Parallel Computing*, Springer Verlag, 2011.

[24] P. Li, H. Johnston, and R. Krasny. A Cartesian treecode for screened Coulomb interactions. *Journal of Computational Physics*, 228(10):3858–3868, 2009.

[25] B. Lu, D. Zhang, and J. McCammon. Computation of electrostatic forces between solvated molecules determined by the Poisson-Boltzmann equation using a boundary element method. *The Journal of Chemical Physics*, 122(21):214102, 2005.

[26] C. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[27] J. Mitchell. Multipole-based calculation of the polarization energy. *Theoretica Chimica Acta*, 94(5):287–294, 1996.

[28] A. Onufriev, D. Bashford, and D. Case. Exploring protein native states and large-scale conformational changes with a modified generalized Born model. *Proteins*, 55(2):383–394, 2004.

[29] W. Ponder et al. Tinker molecular dynamics package, 2012.

[30] R. Petrella, I. Andricioaei, B. Brooks, and M. Karplus. An improved method for nonbonded list generation: Rapid determination of near-neighbor pairs. *Journal of Computational Chemistry*, 24(2):222-231, 2003.

[31] J. Phillips et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781-1802, 2005.

[32] C. Sosa, T. Hewitt, M. Lee, and D. Case, Vectorization of the generalized Born model for molecular dynamics on shared-memory computers. *Journal of Molecular Structure: THEOCHEM*, 549(1–2):193–201, 2001.

[33] T. Simonson and A. Bruenger. Solvation free energies estimated from macroscopic continuum theory: An accuracy assessment. *The Journal of Physical Chemistry*, 98(17):4683–4694, 1994.

[34] D. Tanner, K. Chan, J. Phillips, and K. Schulten. Parallel generalized Born implicit solvent calculations with NAMD. *Journal of Chemical Theory and Computation*, 7(11):3635-3642, 2011.

[35] H. Tjong and H. Zhou. GBr(6): A parameterization-free, accurate, analytical Generalized Born method. *Journal of Physical Chemistry B*, 111(11):3055-3061, 2007.

[36] Weisstein and W. Eric. “Gaussian quadrature.” from mathworld–a wolfram web resource. http://mathworld.wolfram.com/GaussianQuadrature.html.

[37] Z. Xu. Treecode algorithm for pairwise electrostatic interactions with solvent-solute polarization. *Physical Review E*, 81(2):020902, 2010.