Efficient Maintenance and Update of Non-bonded Lists in Macromolecular Simulations

Supplemental Material

Rezaul Chowdhury,† Dmitri Beglov,‡ Mohammad Moghadasi,¶ Ioannis Ch. Paschalidis,§ Pirooz Vakili,∥ Sandor Vajda,‡ Chandrajit Bajaj,⊥ and Dima Kozakov*,‡

Computer Science Department, Stony Brook University, Department of Biomedical Engineering, Division of Systems Engineering, Department of Electrical and Computer Engineering, Department of Mechanical Engineering, Boston University, Boston, USA, and Department of Computer Science, University of Texas at Austin

E-mail: midas@bu.edu

1 Supplementary methods

1.1 Updates description

Here we demonstrate how to efficiently maintain a \((\mathcal{K},2)\)-admissible octree under the conformational changes of the underlying molecular system, during the course of simulation. While it is
possible to maintain a \((K, \alpha)\)-admissible octree within the same asymptotic bounds as a \((K, 2)\)-admissible octree for any \(\alpha > 1\), fixing \(\alpha\) to 2 makes the exposition simpler.

While we will update a \((K, 2)\)-admissible octree using the same idea as described above, the advantage of using a \((K, 2)\)-admissible octree instead of a \((K, 1)\)-admissible one is that the former allows one to obtain a good amortized bound for the update operations. An amortized analysis looks at all operations performed on a data structure, and proves a worst-case upper bound on the total cost of the operations. If the worst-case cost of \(n\) operations is \(F(n)\), the amortized cost of each operation is given by \(f(n) = F(n)/n\). An amortized analysis can capture scenarios in which some of the operations performed on the data structure are expensive\(^a\), but enough low-cost operations are also applied on it to average out the cost of the expensive ones. A worst-case analysis fails to identify such situations.

Suppose we are given a \((K, 2)\)-admissible octree \(T\), and we want to move the center of an atom that already belongs to \(T\) from \(p\) to \(p'\). We find the shortest path through \(T\) that goes from the leaf already containing \(p\) to the leaf that should contain \(p'\) (we need to create the leaf for \(p'\) if it does not already exist). Only the nodes along this path may lose \((K, 2)\)-admissibility due to the removal of \(p\) and addition of \(p'\). We find the furthest proper ancestor of \(p\) with fewer than \(K/2\) atoms, and if such a node exists we contract it to a leaf. We also check if the leaf to which we have added \(p'\) now has more than \(2K\) atoms, and if so, we expand it to a \((K, 2)\)-admissible subtree. The pseudocode is given in Supplementary Figure 3.

2 Supplementary Discussion

2.1 Cache Complexity

Memory in modern computers is typically organized in a hierarchy with registers at the lowest level followed by several levels of caches (L1, L2, and possibly L3) and the RAM. The access time

\(^a\)Consider, for example, when a point in a \((K, \alpha)\)-admissible octree moves to a leaf already containing \(\alpha K\) points, and so the leaf needs to be decomposed into smaller nodes in order to maintain \((K, \alpha)\)-admissibility.
and size of each level increases with its depth. An algorithm is said to have incurred a cache miss at any given cache level if a data item referenced by the algorithm does not reside in that level and must be fetched from a higher and slower level of the memory hierarchy. Cache misses often cause processors to stall while waiting for data to be transferred from slower memory/cache levels. The cache efficiency of an algorithm is measured in terms of the number of cache misses it incurs at each cache level (the fewer the better).

An octree is a cache-friendly data structure. Whenever interaction computation function loads a node $u$ into the cache it computes as many interactions as possible involving the atoms in that node, and it does so recursively. This temporal locality in computation reduces the number of cache misses. On the other hand, energy evaluations involving nblists do not have any temporal locality as no nblist location is accessed more than a constant number of times during the evaluation.

It can be shown that for large globular proteins nblists-based energy evaluation causes $\mathcal{O}(nd^3)$ cache misses, compared to only $\mathcal{O}\left(\frac{nd^2}{B\sqrt{C}} + d + \mathcal{O}(1)\right)$ cache misses incurred by ACCUINTER, where $C$ is the size of the cache and $B$ is the cache-line size.

Supplementary Figure 6 compares the cache performance of the octree-based minimizer with that of the nblists-based minimizer for various proteins and distance cutoffs. Clearly, nblists almost always incur more cache misses than octrees, and their performance w.r.t. octrees degrade as the distance cutoff increases. For 20Å cutoff nblists incur 7 to 10 times more cache misses compared to octrees.

### 2.2 Parallelization, Load Balancing and Vectorization

Recall from the experimental results presented in the main paper that the octree-based minimizer spends most of its time computing the energy values, and the update time is almost negligible (e.g., for 2CGT it’s only 1-5% of the update time of the nblists-based minimizer). Fortunately, the octree-based recursive energy computation routine given in Supplementary Figure Figure 3 is amenable to efficient parallelization. A parallel (multithreaded) version for shared-memory multicore machines is shown in Supplementary Figure Figure 4 which is based on the observation
that many of the recursive calls in Supplementary Figure 3 can be executed in parallel. However, at the time of computing the interaction energy between two atoms the routine also updates each of the two atoms with the force exerted on it by the other. As a result, if a thread computes the interaction between atoms \(a\) and \(b\) in parallel with another thread computing the interaction between atoms \(a\) and \(c\), a race condition may arise if both threads attempt to update the force acting on \(a\) simultaneously and thus leading to a possibly incorrectly updated force vector for \(a\). Hence, if \(\text{PARACCUMINTER}(u, v, d, \text{INTER})\) makes two parallel recursive calls \(\text{PARACCUMINTER}(u', v', d, \text{INTER})\) and \(\text{PARACCUMINTER}(u'', v'', d, \text{INTER})\) then it must make sure that both \(u'\) and \(v'\) are completely disjoint from \(u''\) and \(v''\). Indeed, \(\text{PARACCUMINTER}\) divides the recursive calls into disjoint groups such that each recursive call in a given group updates a set of atoms completely disjoint from the sets updated by all other recursive calls in that group. All recursive calls in the same group can then be executed in parallel. For example, steps 7–24 of Supplementary Figure 4 shows one way of grouping and thus parallelizing the \(8 \times 8 = 64\) possible recursive calls when both \(u\) and \(v\) are non-leaves.

The \(\text{Cilk}^1\) concurrency platform can be used to implement and execute task parallel recursive divide-and-conquer algorithms, such as \(\text{PARACCUMINTER}\), on modern multicore machines efficiently. For load balancing \(\text{Cilk}\) uses a randomized work-stealing scheduler with strong theoretical performance guarantees. Let \(T_P\) denote the running time of an algorithm on \(P\) processing cores, and let \(Q_P\) be the total number (sum) of cache misses it incurs on those cores. Then \(T_1\) denotes the work done by the algorithm, and \(T_\infty\) denotes its span (i.e., the best parallel running time one can expect from the algorithm). Now given a recursive divide-and-conquer algorithm with work \(T_1\), span \(T_\infty\) and serial cache complexity \(Q_1\), \(\text{Cilk}\)'s scheduler guarantees a parallel running time of \(T_P \leq \frac{T_1}{P} + \mathcal{O}(T_\infty)\) (w.h.p.\(^b\)) and a total cache complexity of \(Q_P \leq Q_1 + \mathcal{O}(PCT_\infty)\) (w.h.p.) on \(P\) processing cores each with a private cache of size \(C^1\).\(^3\) We show in Supplementary Section 3 (Lemma 3.2) that for a \((\mathcal{H}, 2)\)-admissible octree storing a molecular system containing \(n\) atoms, \(\text{ACCUMINTER} / \text{PARACCUMINTER}\) performs \(T_1(n) = \mathcal{O}(nd^2 \left(d + \mathcal{H}^{1/3}\right))\) work for

\(^b\)For an input of size \(n\), an event \(E\) occurs w.h.p. (with high probability) if, for any \(\alpha \geq 1\) and \(c\) independent of \(n\), \(Pr(E) \geq 1 - \frac{c}{n^{\alpha}}\). Observe that the larger the value of \(n\), the closer \(Pr(E)\) is to 1, and \(\lim_{n \to \infty} Pr(E) = 1\).
distance cutoff $d$ assuming that each pairwise interaction can be computed in constant time. We compute an upper bound on the span $T_\infty(n)$ of PARACCUMINTER on a molecular system of size $n$ as follows. For simplicity, we assume that the atoms of the molecule are uniformly distributed inside a box of size $h \times h \times h$. Clearly, $h = \Theta \left( n^{\frac{1}{3}} \right)$, and all leaves of the octree constructed for such molecular system will be in the same level. Let $T'_\infty(m)$ be the span of PARACCUMINTER when both $u$ and $v$ are the same octree node of size $m \times m \times m$, and let $T''_\infty(m)$ be its span when $u$ and $v$ are disjoint octree nodes of size $m \times m \times m$ each. Then clearly, for a suitable constant $c$ that depends on the size of the leaves, we have:

$$
T'_\infty(m) \leq \begin{cases} 
\Theta(1) & \text{if } m \leq c, \\
T'_\infty \left( \frac{m}{2} \right) + 14T''_\infty \left( \frac{m}{2} \right) + \Theta(1) & \text{otherwise.}
\end{cases}
$$

$$
T''_\infty(m) \leq \begin{cases} 
\Theta(1) & \text{if } m \leq c, \\
8T''_\infty \left( \frac{m}{2} \right) + \Theta(1) & \text{otherwise.}
\end{cases}
$$

The recurrence for $T'_\infty$ is based on the 15 groups of parallel recursive calls made in steps 11–21 of PARACCUMINTER, and that for $T''_\infty$ is based on the 8 groups in steps 23–24. Solving the recurrences we get: $T''_\infty(m) = \mathcal{O} \left( m^3 \right)$ and $T'_\infty(m) = \mathcal{O} \left( m^3 \right)$. Clearly, $T_\infty(n) = \mathcal{O} \left( T'_\infty(m) \right) = \mathcal{O} \left( T'_\infty \left( \Theta \left( n^{\frac{1}{3}} \right) \right) \right) = \mathcal{O} \left( n \right)$.

Therefore, when run on $P$ processing cores under Cilk’s work-stealing scheduler PARACCUMINTER will terminate in $T_P = \mathcal{O} \left( T_P + T_\infty \right) = \mathcal{O} \left( n^{d/2} \left( d + \mathcal{X}^{1/3} \right) + n \right)$ parallel time (w.h.p.).

In order to extract parallelism even from a single core one can vectorize the base case summation in step 3 of PARACCUMINTER. Observe that when $\mathcal{T}_A = \mathcal{T}_B$ one can evaluate the sum by iterating through every $p \in u$ in the outer loop and computing $\sum_{q \in v \cap \text{dist}(p,q) \leq d \wedge p < q} \text{INTER}(p,q)$ in the inner loop. Similarly for $\mathcal{T}_A \neq \mathcal{T}_B$. Assuming that the atoms in each leaf are stored in a linear array, the inner loop given above can be vectorized easily. The vectorization can be made even more efficient by observing that if $d$ is significantly longer than the diameter (i.e., diagonal length) of a leaf, one can often drop the $\text{dist}(p,q) \leq d$ predicate from the inner loop by checking if $\text{dist}(u,v) + d_u + d_v < d$ holds outside the outer loop, where $d_u$ (resp. $d_v$) is the diameter of node $u$.

Though not as important as parallelizing the energy computation function, parallelization of the update function given in the Supplementary Figure 5 is not very difficult either. Parallel execution proceeds in two phases. In the first phase, one considers all atoms that have changed positions in parallel, and marks all octree nodes that may need to be updated. In the second phase, marked
nodes in independent parts of the octree are updated in parallel.

### 2.2.1 Experimental Results

We implemented our parallel octree algorithms in C with Intel’s Cilk Plus extension for parallelization, and compiled using ICC 13.0.1. We vectorized the base cases (step 3), and compiled our code with “-O3 -ip -parallel -AVX -xhost” optimization parameters. We ran our simulations on a multicore machine equipped with two 8-core 2.7GHz E5-2680 Intel Xeon (Sandy Bridge) processors and 32GB RAM. For best vectorization results we had to increase $\mathcal{H}$ to 150.

![Figure 1](a) vdW  
(b) vdW + Coul

Figure 1: Speedup factors achieved by PARACCUMINTER (w.r.t. its serial running time without vectorization) when it is run on a multicore machine with two 8-core ($2 \times 8 = 16$ cores in total) Intel Sandy Bridge processors with and without vectorization.

Supplementary Figure 1 plots the speedup factors achieved by PARACCUMINTER (w.r.t. its serial running time without vectorization) for various distance cutoffs from 9Å to 20Å when it is run on 16 cores with and without vectorization. While Figure 1(a) shows results for vdW energy only, Figure 1(a) plots results when both vdW and Coul are used. The plots show that the parallel version without vectorization consistently achieves a factor 12 speedup over its serial baseline. With vectorization of the parallel code the speedup factor reaches 16 for $d = 9$Å which increases to almost 20 when $d = 20$Å.

---

\(^c^\)GCC versions 4.7 and above also support Cilk extensions  
\(^d^\)a single compute node of the Stampede Supercomputing Cluster
3 Proofs

Lemma 3.1. A \((\mathcal{K}, \alpha)\)-admissible octree \(\mathcal{T}\) storing molecular system containing \(n\) atoms can be constructed in \(O(n \log n)\) time, where \(\mathcal{K}\) is a positive integer and \(\alpha \geq 1\).

**Proof.** An octree is constructed based only on the Cartesian coordinates of the atoms in the given molecular system\(^6\). Since all atoms in the molecular system form a single spatially connected component in the sense that they can never be split into two groups completely and widely separated from each other in space\(^7\), the entire molecular system must fit inside a bounding box (i.e., cube) of dimension (i.e., length of a side) \(O(n)\). As in an octree the children of a node \(u\) are formed by decomposing the cube corresponding to \(u\) into 8 smaller subcubes, the height of \(\mathcal{T}\) is clearly \(O(\log n)\). Thus construction of \(\mathcal{T}\) (see \textsc{constructAdmissibleOctree} and \textsc{expandNode} in Supplementary Figure 2) involves scanning each atom of the molecular system at most \(O(\log n)\) times (i.e., once for each level of recursion of \textsc{expandNode}). Hence the worst-case cost of constructing \(\mathcal{T}\) is \(O(n \log n)\). □

Lemma 3.2. Given a \((\mathcal{K}, 2)\)-admissible octree storing a molecular system containing \(n\) atoms, all pairwise interactions between atoms within distance \(d\) can be computed in \(O\left(nd^2 \left(\delta d + \mathcal{K}^{1/3}\right)\right)\) time, where \(\delta\) is the time to compute one pairwise interaction, and \(\mathcal{K}\) is a positive integer.

**Proof.** Function \textsc{accumulate} given in Supplementary Figure 3 shows how to compute all pairwise interactions between atoms within a given distance cutoff. For any given atom \(q\), let \(\mathcal{S}_q\) be the sphere of radius \(d\) centered at the center of \(q\). Since each atom has a small volume bounded by a constant, the number of atoms with centers strictly inside \(\mathcal{S}_q\) will be bounded by \(O(d^3)\). \textsc{accumulate} will compute pairwise interactions of \(q\) with all these \(O(d^3)\) atoms. It will also consider atoms in all fringe nodes, i.e., octree leaves that lie only partly inside \(\mathcal{S}_q\). \textsc{accumulate} will have the worst running time provided each fringe node contains the largest number of atoms.

---

\(^6\)No chemical connectivity information is used during octree construction.

\(^7\)A molecular system cannot be split into two atom groups \(G_1\) and \(G_2\) such that each atom in \(G_1\) is far away (say, more than 10Å away) from every atom of \(G_2\). In other words, there must exist at least one atom in \(G_1\) that is close to (say, less than 10Å away from) at least one atom in \(G_2\).
in the smallest possible volume. But each leaf can contain at most $2^K$ atoms. The smallest octree leaf containing $2^K$ atoms will cover a cubic region of side $l = \Theta\left(\frac{K}{3}\right)$. Since $S_q$ has a surface area of $\Theta\left(d^2\right)$ and each of $q$’s fringe nodes partially intersects $S_q$, the number of such fringe nodes with side $l$ will be $\Theta\left(\frac{d^2}{l^2}\right)$, and the total number of atoms inside those nodes will be $\Theta\left(\frac{d^2}{l^2} \times l^3\right) = \Theta\left(d^2l\right)$. Hence, the total number of atoms inside all of $q$’s fringe nodes with a side length $l$ or smaller is $\Theta\left(d^2 \sum_{h=0}^{\log_2(l)} \left(\frac{l}{2}\right)\right) = \Theta\left(d^2, K^{1/3}\right)$. Thus the time spent by $\text{ACCUMINTER}$ for computing interactions involving any given atom is $\Theta\left(d^3 + d^2K^{1/3}\right)$ from which the claimed bound for the entire molecular system follows. ■

Lemma 3.3. A $(K,2)$-admissible octree storing a molecular system containing $n$ atoms supports updates of atomic positions in $\Theta\left(\log n\right)$ amortized time each, where $K$ is a positive integer.

Proof. Function $\text{UPDATEADMISSIBLEOCTREE}$ given in Supplementary Figure 5 updates an atom’s position in a $(K,2)$-admissible octree. It calls two other functions named $\text{CONTRACTNODE}$ (in Supplementary Figure 5) and $\text{EXPANDNODE}$ (see Supplementary Figure 2) as subroutines.

We will use the potential method for amortized analysis. Let $\{o_0...o_i...o_f\}$ be a sequence of operations on data structure, with $S_i$ is the state of data structure before the operation $o_i$, and $S_{i+1}$ is the state of the data structure after operation $o_i$. Let $\Phi$ is such function, that amortized time for operation $o_i$ is defined to be $T_{\text{amort}}(o_i) = T_{\text{actual}}(o_i) + (\Phi(S_{i+1}) - \Phi(S_i))$. $\Phi(S_0) = 0$. We can estimate the upper bound of actual time of the execution. Intuitively, inexpensive data structural operations build up potential energy in $\Phi$, and expensive operations use the energy accumulated in $\Phi$ to pay for their own costs. For example, consider a sequence of operations performed on a data structure, and suppose we would like to show that each such operation has an amortized cost of $a$. Now if $c_i$ is the actual cost of the $i$-th operation in the sequence and $c_i < a$ (an inexpensive operation) then $a - c_i$ to will be added to the
potential energy already stored in $\Phi$. However, if $c_i > a$ (an expensive operation), the excess cost $c_i - a$ will be paid for by $\Phi$. If we can find a function $\Phi$ that can pay for all expensive operations (i.e., with actual cost $> a$) in the sequence without ever falling below 0 then we can conclude that the amortized cost of each operation in the sequence is $a$.

We define the potential of a $(\mathcal{K}, 2)$-admissible octree $\mathcal{T}$ as:

$$\Phi(\mathcal{T}) = \sum_{u \in \mathcal{T}} k(\mathcal{N}(u) - \mathcal{K}) \log n + \sum_{u \in \mathcal{T}} k(\mathcal{K} - \mathcal{N}(u)),$$

where, $k$ is a positive constant.

Let us first consider the amortized cost of ContractNode($u$) where $u$ is an internal node with $\mathcal{N}(u) < \mathcal{K}/2$. The actual cost of performing this operation is $O(\mathcal{N}(u)) = O(\mathcal{K})$ as it involves mainly a depth-first traversal of the subtree rooted at $u$. The drop in potential due to this operation is at least $k\mathcal{K}/2$ as at least one internal node (i.e., $u$) with fewer than $\mathcal{K}/2$ atoms disappears. The disappearance and appearance of the leaf nodes do not contribute to the total potential as none of them has more than $\mathcal{K}/2$ atoms. Thus the amortized cost of this operation is $O(\mathcal{K}) - k\mathcal{K}/2 \leq 0$ for sufficiently large $k$.

Next we consider the amortized cost of ExpandNode($u$, $\mathcal{K}$, 1) where $u$ is a leaf with $\mathcal{N}(u) > 2\mathcal{K}$. The actual cost of performing this operation is $O(\mathcal{N}(u) \log n) = O(\mathcal{K} \log n)$. The potential drop is $k\mathcal{K} \log n$ as one leaf (i.e., $u$) with more than $2\mathcal{K}$ atoms disappears. Since this leaf is replaced with a $(\mathcal{K}, 1)$-admissible subtree none of the nodes in that subtree contributes to the change in potential. Thus the amortized cost of this operation is $O(\mathcal{K} \log n) - k\mathcal{K} \log n \leq 0$ provided $k$ is sufficiently large.

We are now in a position to compute the amortized cost of UpdateAdmissibleOctree. We have already shown that the amortized cost of lines 11–12 (i.e., ContractNode and ExpandNode) is zero. So we can concentrate on lines 1–10. The actual cost of lines 2, 3 and 9 is
\( O(\log n) \) as each performs work proportional to no more than the height of \( \mathcal{T} \). The other lines perform \( O(1) \) work. Since lines 1–10 change the number of atoms in at most 2 leaves and \( O(\log n) \) internal nodes by at most 1, the increase in potential is \( O(\log n) \). Hence, the amortized cost of this update operation is \( O(\log n) \).

The following corollary says that when the updates are local the update costs can be much smaller than one is predicted by Lemma 3.3.

**Corollary 3.1. [Local Updates]** If a sequence of updates performed on a \((g,2)\)-admissible octree \( \mathcal{T} \) can be decomposed into \( k \geq 1 \) disjoint subsequences \( S_1, S_2, \ldots, S_k \) such that all updates in \( S_i \) perform atomic movements only within a subtree \( \mathcal{T}_i \) of \( \mathcal{T} \) and all such \( \mathcal{T}_i \)s are disjoint, then the updates in \( S_i \) take \( O(h_i) \) amortized time each, where \( h_i \) is the height of \( \mathcal{T}_i \).

**Proof.** The corollary follows by applying Lemma 3.3 on each \( \mathcal{T}_i \) separately as each of them is also a \((g,2)\)-admissible octree.

**Lemma 3.4.** A contracted \((g,\alpha)\)-admissible octree \( \mathcal{T} \) storing molecular system containing \( n \) atoms uses \( \Theta(n) \) space, where \( g \) is a positive integer and \( \alpha \geq 1 \).

**Proof.** All atoms are stored in the leaves using a total of \( \Theta(n) \) space. Since each leaf stores at least 1 atom, \( \mathcal{T} \) has \( O(n) \) leaves. Also since each internal node has at least 2 children, the number of internal nodes in \( \mathcal{T} \) is \( O(n) \). But each node uses only a constant amount of extra space. Thus the total space occupied by \( \mathcal{T} \) is \( \Theta(n) \).

**References**

(1) Leiserson, C. E. The Cilk++ concurrency platform. 2009.

(2) Arora, N. S.; Blumofe, R. D.; Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. 1998.

(3) Acar, U. A.; Blelloch, G. E.; Blumofe, R. D. The data locality of work stealing. 2000.

\(^8\) A subsequence can be extracted from a given sequence by removing zero or more elements from the sequence without changing the order of the remaining elements
(4) TACC, Texas Advanced Computing Center. https://www.tacc.utexas.edu/.

(5) Cormen, T.; Leiserson, C.; Rivest, R.; Stein, C. Introduction to Algorithms, 3rd ed.; The MIT Press, 2009.
Figure 2: Recursive construction of a \((\mathcal{X}, \alpha)\)-admissible octree from a given set \(P\) of points in 3D, where \(\mathcal{X}\) is a positive integer and \(\alpha\) is a number not smaller than 1.

Figure 3: Accumulating \(\sum_{a \in \mathcal{A}, b \in \mathcal{B} \atop \text{dist}(a, b) \leq d} \text{INTER}(a, b)\) recursively, where \(a \in \mathcal{T}_A\) and \(b \in \mathcal{T}_B\) are two octree nodes (possibly \(\mathcal{T}_A = \mathcal{T}_B\)), and \(\text{INTER}\) is a user-specified function for computing the interaction between a pair of atoms. The initial call is \(\text{ACCUMINTER}(\text{ROOT}(\mathcal{T}_A), \text{ROOT}(\mathcal{T}_B))\) for the accumulation of all pairwise interactions within distance cutoff \(d\).
PARACCUMINTER( u, v, d, INTER ) \{parallel recursive accumulation of pairwise interactions\}

(Inputs are octree nodes \( u \in \mathcal{T}_A \) and \( v \in \mathcal{T}_B \) (possibly \( \mathcal{T}_A = \mathcal{T}_B \)), and output is \( \sum_{u \in \mathcal{S} \land v \in \mathcal{S} \land d(u,v) \leq d} \text{INTER}( u, v ) \), where \text{INTER} is a user-specified function for computing the interaction between a pair of points. By \text{CHILD}(x) we denote the set of non-empty octree nodes obtained by subdividing node \( x \). Each of the eight octants of \( x \) is assigned a unique index in [0, 7] based on its location in \( x \) relative to other octants, e.g., 0: top-front-right, 1: top-back-right, 2: top-back-left, 3: top-front-left, 4: bottom-front-right, 5: bottom-back-right, 6: bottom-back-left and 7: bottom-front-left. For each \( x' \in \text{CHILD}(x) \), function OCTANTID(x, x') returns the unique octant index of \( x' \) inside \( x \).

1. if \( u = \text{NIL} \) or \( v = \text{NIL} \) then return 0 \{if \( u \) and/or \( v \) do/does not exist \( \Rightarrow \) no interactions\}

2. if \( \text{dist}( u, v ) > d \) then return 0 \{surface-to-surface distance \( > d \Rightarrow \) no interactions\}

3. else if \( \text{LEAF}(u) \land \text{LEAF}(v) \) then return \( \sum_{p \in \mathcal{S} \land q \in \mathcal{S} \land \text{dist}(p,q) \leq d} \text{INTER}( p, q ) \) \{iterative computation \( \lor \) \text{vectorized}\}

4. else if \( \text{LEAF}(u) \) then return \( \sum_{v' \in \text{CHILD}(v)} \text{PARACCUMINTER}( u, v', d, \text{INTER} ) \) \{recurse on \( v \)\}

5. else if \( \text{LEAF}(v) \) then return \( \sum_{u' \in \text{CHILD}(u)} \text{PARACCUMINTER}( u', v, d, \text{INTER} ) \) \{recurse on \( u \)\}

6. else \{recurse on \( u \) and \( v \) \( \land \) \text{parallelized}\}

7. array \( u\text{child}[0 : 7] = \{ \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL} \} \), \( v\text{child}[0 : 7] = \{ \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL}, \text{NIL} \} \)

8. \( \forall u' \in \text{CHILD}(u) \text{;} \text{uchild}[ \text{OCTANTID}(u, u') ] \leftarrow u' \), \( \forall v' \in \text{CHILD}(v) \text{;} \text{vchild}[ \text{OCTANTID}(v, v') ] \leftarrow v' \)

9. array \( s[0 : 7] = \{ 0, 0, 0, 0, 0, 0, 0, 0 \} \)

10. if \( u = v \) then \{can happen when \( \mathcal{T}_A = \mathcal{T}_B \)\}

\begin{verbatim}
11.   parallel \( \forall i \in [0, 7] \colon s[i] \leftarrow \text{PARACCUMINTER}( \text{uchild}[i], \text{vchild}[i], d, \text{INTER} ) \\
12.     for j \leftarrow 0 \text{ to } 3 \text{ do} \\
13.       parallel \( \forall i \in [0, 3] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[i], \text{vchild}[4 + (i + j) \text{ mod } 4], d, \text{INTER} ) \\
14.     for j \leftarrow 0 \text{ to } 3 \text{ do} \\
15.       parallel \( \forall i \in [0, 3] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[4 + (i + j) \text{ mod } 4], \text{vchild}[i], d, \text{INTER} ) \\
16.     for j \leftarrow 0 \text{ to } 1 \text{ do} \\
17.       parallel \( \forall i \in [0, 1, 4, 5] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[i], \text{vchild}[i + 1 + (j + 1) \text{ mod } 2], d, \text{INTER} ) \\
18.     for j \leftarrow 0 \text{ to } 1 \text{ do} \\
19.       parallel \( \forall i \in [0, 1, 4, 5] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[i + 1 + (j + 1) \text{ mod } 2], \text{vchild}[i], d, \text{INTER} ) \\
20.     parallel \( \forall i \in [0, 2, 4, 6] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[i], \text{vchild}[i + 1], d, \text{INTER} ) \\
21.     parallel \( \forall i \in [0, 2, 4, 6] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[i + 1], \text{vchild}[i], d, \text{INTER} ) \\
22.   else \{ u \text{ and } v \text{ are disjoint} \} \\
23.     for j \leftarrow 0 \text{ to } 7 \text{ do} \\
24.       parallel \( \forall i \in [0, 7] \colon s[i] \leftarrow s[i] + \text{PARACCUMINTER}( \text{uchild}[i], \text{vchild}[(i + j) \text{ mod } 8], d, \text{INTER} ) \\
25.   return \sum_{0 \leq i \leq 7} s[i] \\
\end{verbatim}

Figure 4: Parallel recursive accumulation of \( \sum_{u \in \mathcal{S} \land v \in \mathcal{S} \land \text{dist}(u,v) \leq d} \text{INTER}( u, v ) \), where \( u \in \mathcal{T}_A \) and \( v \in \mathcal{T}_B \) are two octree nodes (possibly \( \mathcal{T}_A = \mathcal{T}_B \)), and \text{INTER} is a user-specified function for computing the interaction between a pair of atoms. The initial call is \text{ACCUMINTER}( \text{ROOT}(\mathcal{T}_A), \text{ROOT}(\mathcal{T}_B) \) for the accumulation of all pairwise interactions within distance cutoff \( d \).
Figure 5: Replacing a point \( p \) with a new point \( p' \) in a \((\mathcal{X}, 2)\)-admissible octree while maintaining its \((\mathcal{X}, 2)\)-admissibility.
Figure 6: Plot (a) shows the ratio of the number of L1 misses incurred by the nblists-based LBFGS minimizer to that incurred by the octree-based LBFGS minimizer for various proteins and distance cutoff values, while plot (b) shows the same for L2 misses. LBFGS minimized only vdw energy, and executed 100 minimization steps.
Figure 7: Comparison of the update time values involving nblists and octrees: (a) vDW energy for various distance cutoffs, where both data structures use the same cutoff value. (b) vDW plus hBond energy, where nblists use the same cutoff value for both types of energy while octrees use a fixed 3Å cutoff for hBond and the same cutoff as nblists for vDW. (c) vDW plus hBond plus Coul, where nblists use the same cutoff value for all types of energy while octrees use a fixed 3Å cutoff for hBond, a fixed 12Å cutoff for vDW, and the same cutoff as nblists for Coul.