Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2012-005 February 23, 2012

A Lossy, Synchronization-Free, Race-Full, But Still Acceptably Accurate Parallel Space-Subdivision Tree Construction Algorithm

Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

A Lossy, Synchronization-Free, Race-Full, But Still Acceptably Accurate Parallel Space-Subdivision Tree Construction Algorithm

Martin C. Rinard
MIT EECS and CSAIL
{rinard}@csail.mit.edu

Abstract

We present a new synchronization-free space-subdivision tree construction algorithm. Despite data races, this algorithm produces trees that are consistent enough for the client Barnes-Hut center of mass and force computation phases to use successfully. Our performance results show that eliminating synchronization improves the performance of the parallel algorithm by approximately 20%. End-to-end accuracy results show that the resulting partial data structure corruption has a neglible effect on the overall accuracy of the Barnes-Hut N-body simulation.

We note that many data structure manipulation algorithms use many of the same basic operations (linked data structure updates and array insertions) as our tree construction algorithm. We therefore anticipate that the basic principles the we develop in this paper may effectively guide future efforts in this area.

1. Introduction

Many computations (for example, lossy video and image processing computations, information retrieval computations, and many scientific computations) are designed to produce only an approximation to an (often inherently unrealizable) ideal output. Because such computations have the flexibility to produce any one of a range of acceptably accurate outputs, they can productively tolerate otherwise unacceptable events such as skipped loop iterations [11, 17, 18, 29, 37], skipped tasks[21, 22], or data races [15, 16] as long as these events do not unacceptably degrade the accuracy of the output. Potential benefits of exploiting this flexibility include reduced power consumption [11, 15–18, 29, 37], increased performance [11, 15–18, 29, 37], simpler and more powerful automatic parallelization techniques [15, 16], and the elimination of synchronization between parallel threads [15, 16].

We present a parallel tree construction algorithm that uses only primitive reads and writes — it uses no specialized synchronization mechanisms at all (even though multiple parallel updates to the tree under construction may interfere). Moreover, unlike much early research in the field [14], the algorithm does not use reads and writes to synthesize higher-level synchronization constructs.

Now, it may not be clear how to implement a correct parallel tree construction algorithm without sychronization. Indeed, we do not attempt to do so. We instead develop an algorithm that 1) does not crash, 2) produces a tree that is consistent enough for its clients to use successfully, and 3) contains enough of the inserted elements so that its clients can produce an acceptably accurate output. In effect, we eliminate synchronization by leveraging the end-to-end ability of the client computations to tolerate some imprecision and corruption in the tree that the algorithm produces.

1.1 Barnes-Hut N-Body Simulation

We use our algorithm for the tree construction phase of the Barnes-Hut N-body simulation computation [1, 30]. This computation simulates a system of N interacting bodies (such as stars or galaxies). Instead of computing the force acting on each body with the straightforward pairwise N^2 algorithm, Barnes-Hut instead inserts the N bodies into a space-subdivision tree, computes the center of mass at each node of the tree, then uses the tree to approximate the combined forces from multiple distant bodies as the force from the center of mass of the distant bodies.

Barnes-Hut is a mainstay of parallel computing benchmark suites [12, 13, 15, 16, 25–27, 33]. Because the force computation phase consumes the majority of the computation time in sequential executions, many parallelization efforts have focused on that phase (even though the tree construction phase can easily become the performance bottleneck in parallel implementations [28]). But unlike the force computation phase (in which the parallel computations for the different bodies are independent and therefore data-race free), the parallel tasks in the tree construction phase may interfere, specifically when bodies inserted by different parallel tasks fall into the same region of the tree.

In the absence of synchronization, the parallel tree construction algorithm will therefore generate data races that complicate its ability to execute successfully. But because the tree construction algorithm is a component of the larger N-body simulation, it is also possible to obtain an end-to-end measure of the effect that these data races have on the overall accuracy of the computation. We use this measure to determine the acceptability of our tree construction algorithm for this computation.

1.2 Key Ideas

As one might anticipate, our starting sequential tree construction algorithm crashes when executed in parallel. The first step is to modify the algorithm to ensure that it does not crash. Two key ideas are central to this modification (Section 4).

- **Link At End:** Some operations replace part of the tree with a new subtree. The starting algorithm links the subtree in place before it was fully constructed, enabling fatal data races as parallel tasks interact with the subtree construction algorithm.

  The modified algorithm first creates the new subtree locally without interference, then executes a last instruction that links the subtree in place. This technique ensures that parallel tasks observe only fully constructed subtrees.
- **Local Read, Check, and Index:** Some operations insert items into shared arrays. These operations read the last element index associated with the array two times: first to to check if the array is full and (if not) then again to determine where to insert the

item. The resulting data races can cause out of bounds accesses that crash the algorithm.

The modified array insert operations read the last element index into a local variable at the start of the operation. The algorithm then uses this local variable for both the check and to determine where to insert the item. This technique eliminates out of bounds array accesses.

With these modifications, we obtain an unsynchronized algorithm that does not crash, but may produce a tree that does not contain all of the inserted elements. We next modify the algorithm to reduce (but not completely eliminate) the number of missing inserted elements (Section 5).

- **Final Check and Retry:** Some operations check a condition, construct a new subtree, then link the new subtree into place. While the new subtree is under construction, other parallel tasks may insert items into the same region of the tree. The instruction that links the new subtree in place then removes these items.

  The modified algorithm performs a final check to see if the condition still holds. If not, it cancels the instruction that links the subtree in place and retries the operation in the updated tree state. This technique reduces the "window of vulnerability" between the check and link, thereby increasing the number of inserted items that appear in the final tree.
- **Data Structure Repair:** Because of (non-fatal) data races in the modified algorithm, it is possible for the tree to enter a somewhat inconsistent state in which one of the final checks always fails. We eliminate the resulting infinite loop with a data structure repair algorithm that eliminates the inconsistency and enables the final check to eventually succeed.

1.3 Consistency Properties

Our unsynchronized algorithms contain data races that may drop inserted bodies from the tree and leave the tree in a partially corrupted state. Nevertheless, the clients (the center of mass and force computation phases) are still able to use the tree to successfully compute the force acting on each body in the system.

To explain why the races do not interfere with the ability of the clients to use the tree successfully, we identify several important consistency properties that the tree satisfies in spite of the data races (Section 2). For example, the tree contains a subset of the inserted bodies, with no body appearing more than once, and the internal nodes of the data structure form a tree. These properties enable the clients to traverse the tree without crashing and to compute an acceptably accurate approximation to the force acting on each body.

1.4 Experimental Results

Our experimental results show that our unsynchronized tree construction algorithms exhibit good parallel performance — on 16 cores they run almost 8 times faster than the original sequential tree construction algorithm and approximately 20% faster than the corresponding synchronized version.

We evaluate the acceptability of the trees that our algorithm produces by measuring how they affect the accuracy of the body positions that the Barnes-Hut N-body simulation computes. We compare versions that use our unsynchronized algorithms with 1) a synchronized algorithm that produces a fully consistent tree and 2) a hyperaccurate algorithm that goes deeper into the space-subdivision tree during the force computation phase before it applies the center of mass approximation.

Our results show that the body positions from the synchronized and unsynchronized versions are much closer to each other than they are to the body positions from the hyperaccurate version. In fact, the the synchronized and unsynchronized body positions are typically two orders of magnitude closer to each other than they are to the hyperaccurate body positions.

Together, these results show that eliminating synchronization improves the performance while preserving the ability of our unsynchronized algorithms to produce acceptably accurate and consistent space-subdivision trees.

1.5 Contributions

This paper makes the following contributions:

- **Algorithm:** It presents new unsynchronized algorithms for building space-subdivision trees in parallel.
- **Evaluation Methodology:** The unsynchronized algorithms contain data races and may therefore produce somewhat corrupted trees. We evaluate the acceptability of these trees by comparing the results produced with these trees to results produced 1) with fully consistent trees and 2) by a hyperaccurate version of the algorithm.
- **Experimental Results:** It presents experimental results that characterize the performance and accuracy of the unsynchronized tree construction algorithms. These results show that the algorithm exhibits good parallel performance (speedup of approximately 8 on 16 cores) and good accuracy (corresponding bodies in the unsynchronized and synchronized versions are typically two orders of magnitude closer to each other than they are to corresponding bodies in the hyperaccurate version).

The basic operations in the tree construction algorithm (linked data structure manipulations and array insertions) are shared by a broad range of algorithms that manipulate data structures. We therefore anticipate that the basic principles that the tree construction algorithm illustrates may help guide developers of other unsynchronized or only partially synchronized algorithms.

2. Data Structures

The Barnes-Hut N-body simulation algorithm works with a hierarchical space-subdivision tree. In our implementation, this tree contains *bodies* (the N bodies in the simulation), *cells* (each of which corresponds to a spatial region within the space-subdivision tree), and *leaves* (each of which stores a set of bodies that are located within the same leaf region of the tree).

In the hierarchical structure of the tree, the regions are nested — each cell is divided into eight octants. Each cell therefore contains eight references to either a hierarchically nested cell or leaf, each of which corresponds to one of the octants in the parent cell's region.

2.1 Class Declarations

Figure 1 presents the C++ class declarations for the data structures that our algorithm uses to represent the space-subdivision tree.[1] We omit fields that are not relevant to the tree construction algorithm. Instances of the class `body` represent the N bodies in the simulation. Each body has an instance variable `position` that records its current position in the simulation. Instances of the class `cell` represent cells in the space-subdivision tree. Each cell has an array `children` that contains references to the (cell or leaf) data structures that represents its hierarchically nested octants. Instances of the class `leaf` represent leaves in the space-subdivision tree. Each leaf has an array `bodies` that contains references to the bodies in the leaf node. The instance variable `numberOfBodies` stores how many leaves the `bodies` array references.

[1] The data structures and algorithms in this paper were all derived from an implementation of the Barnes-Hut algorithm in the QuickStep benchmark suite [15, 16].

```
class body {
  vector position;
  ...
};

class node {
  public:
    int cellOrLeaf;
    int level;
  ...
};

class cell : public node {
 public:
   node *children[NUMBEROFOCTANTS];
   ...
};

class leaf : public node {
  public:
   int numberOfBodies;
   body *bodies[MAXBODIES];
  ...
};
```

Figure 1. Space Subdivision Tree Data Structure Declarations

Both the `cell` and `leaf` classes are subclasses of the `node` class, which contains a flag `cellOrLeaf` indicating whether the instance is a cell or leaf and a field `level` indicating the level at which the cell or leaf appears in the tree.

2.2 Consistency Properties

We first discuss consistency properties of a final space-subdivision tree produced by a tree construction algorithm that starts with an empty root cell, then inserts all N bodies in the simulation into the tree. These consistency properties are relevant primarily because they influence the ability of the Barnes-Hut center of mass and force computation phases to use the tree to deliver an acceptable approximation to the forces acting on each body.

2.2.1 Natural Consistency Properties

Some natural consistency properties of the tree include:

- **Tree:** The `cell`, `leaf`, and `body` objects form a tree, where the edges of the tree consist of the references stored in the `children` fields of the cell objects and the `bodies` fields of the leaf objects. Note that this property ensures that each body appears at most once in the tree.
- **Body Inclusion:** If a body appears in the tree (i.e., is referenced by a `leaf` object reachable from the root), then it was inserted into the tree by the tree construction algorithm. Conversely, each body that was inserted into the tree actually appears in the tree.
- **Octant Inclusion:** Each `leaf` object corresponds to a region of space, with the region defined by the path to the leaf from the root cell object. The positions of all of the bodies that the leaf object references must be within that region.
- **Leaf Representation:** The `numberOfBodies` field in each leaf object is at least zero and less than `MAXBODIES`. For all `i` at least zero and less than `numberOfBodies`, `bodies[i]` references a body object (and is therefore not `NULL`). For all `i` at least `numberOfBodies` and less than `MAXBODIES`, `bodies[i]` is `NULL`.
- **Level:** The `level` of the root object is zero. The `level` of all other cell and leaf objects is the level of the enclosing cell object plus one.
- **cellOrLeaf:** The `cellOrLeaf` field in cell objects is `CELL`; the `cellOrLeaf` field in leaf objects is `LEAF`.

Together, we call these properties the *Natural Properties* of the tree construction algorithm. The original sequential tree construction algorithm produces a tree that satisfies these properties.

2.2.2 Relaxed Consistency Properties

We note that the above properties are stronger than the center of mass and force computation phases actually require to produce an acceptable approximation to the forces acting on the bodies. And in fact, our parallel synchronization-free algorithm may not (and in practice does not) produce a tree that satisfies the above Natural Properties. Specifically, it may produce a tree that violates the Body Inclusion and Leaf Representation properties.

We first consider what requirements the tree must satisfy so that the center of mass and force computation phases do not crash. The following Relaxed Body Inclusion and Relaxed Leaf Representation properties, which replace the Body Inclusion and Leaf Representation properties, respectively, are sufficient to ensure that the center of mass and force computation phases do not crash. Our parallel synchronization-free algorithm produces a tree that satisfies these relaxed properties. It also produces a tree that satisfies the remaining Tree, Octant Inclusion, Level, and cellOrLeaf properties.

- **Relaxed Body Inclusion:** If a body appears in the tree (i.e., is referenced by a `leaf` object reachable from the root), then it was inserted into the tree by the tree construction algorithm. If a body was inserted into the tree by the tree construction algorithm, it may or may not appear in the tree.
- **Relaxed Leaf Representation:** The `numberOfBodies` field in each leaf object is at least zero and less than `MAXBODIES`. For all `i` at least zero and less than `numberOfBodies`, `bodies[i]` references a body object (and is therefore not `NULL`). For all `i` at least `numberOfBodies` and less than `MAXBODIES`, `bodies[i]` may or many not be `NULL`.

Together, we call these properties the *Relaxed Properties*. Our parallel tree construction algorithm will produce a tree that satisfies these properties.

We note that all of these properties are hard logical correctness properties of the type that standard data structure reasoning systems work with [34, 35]. One way to establish that the algorithm always produces a tree with these properties is to reason about all possible executions of the program to show that all executions produce a tree that satisfies these properties.

In addition to these properties, our parallel synchronization-free algorithm must satisfy the following Accuracy property:

- **Accuracy:** The tree contains sufficiently many bodies so that the force computation phase produces a sufficiently accurate result.

In contrast to the hard logical consistency properties that we discuss above, we do not establish that our algorithm preserves this property by reasoning about all possible executions of the algorithm. Indeed, this approach would fail, because some of the possible executions violate this property.

We instead reason empirically about this property by observing executions of the program on representative inputs. Specifically, we compare the results that the program produces when it uses our parallel synchronization-free algorithm with results that we know to be accurate. We use this comparison to evaluate the accuracy of the results from the parallel synchronization-free algorithm.

2.2.3 Execution Properties

The properties discussed in the previous two sections relate only to the final tree that the algorithm produces. But to reason about the operation of the tree construction algorithm itself, we must identify properties that the tree satisfies while it is under construction. Our primary purpose is to ensure that the tree construction algorithm itself does not crash.

As the algorithm operates, it will create and destroy references between cells, leaves, and bodies. To state consistency properties that hold during the execution of the tree construction algorithm, we consider *all* references that ever existed at any point during the execution of the algorithm and *all* cells and leaves that existed at any point during the execution of the algorithm (whether or not these cells and leaves are reachable with the references currently stored in the tree).

We define the *nodes* of the *full space-subdivision graph* to be all of the cell, leaf, and body objects that ever existed during the execution of the algorithm and the *edges* of the full space-subdivision graph to be all of the references ever stored in `children` fields of cell objects or `bodies` fields of leaf objects.

At all points during the execution of the tree construction algorithm, the full space-subdivision graph satisfies the Relaxed Body Inclusion, Octant Inclusion, Relaxed Leaf Representation, Level, and cellOrLeaf properties discussed above in Sections 2.2.1 and 2.2.2. The following two properties also hold:

- **Cell Tree:** The set of references ever stored in any `children` field of any cell object that ever existed during the execution algorithm is a tree.
- **Body Isolation:** No leaf object references more than one body object.

Together, we call this set of properties the *Execution Properties*. As we will see, the Execution Properties properties are 1) preserved by all operations of the parallel synchronization-free tree construction algorithm, 2) sufficient to ensure that this algorithm will not crash, and 3) ensure that the algorithm will produce a final tree that satisfies the Relaxed Properties.

Note that it is possible for multiple leaf objects, specifically an old leaf object that has been unlinked from the tree and one or more new leaf objects that may have been subsequently linked into the tree, to reference the same body object. Also note that the Octant Inclusion property ensures that each body is referenced by at most one leaf object in the final tree (because different leaf objects in the final tree correspond to disjoint regions).

3. Original Sequential Algorithm

Figure 2 presents the (slightly simplified and reconstructed) original sequential algorithm for inserting a single body `b` with integer coordinates `x` into the space-subdivision tree rooted at the cell `c` (the body `b` has double coordinates, but the insertion algorithm uses an integer version of these coordinates).

The `insert` algorithm first determines the octant of the cell `c` into which the body `b` falls (line 6). If this octant is free, the algorithm allocates a new `leaf` object to hold the bodies in the octant (lines 9–10), links the leaf into the enclosing cell (line 11), then inserts the body into the leaf (lines 12–13). We assume that new object allocations take place atomically and without interacting with computations running on other processors. We achieve this property by providing each processor with its own pools of cells and leaves from which it can allocate without interference.

If the tree contains a leaf for the octant, the algorithm checks if the leaf is full of bodies (line 17) and, if not, inserts the body into the leaf (lines 21–22). If the leaf is full, the algorithm invokes the `divide` algorithm to replace the leaf with a new cell (line 18, algorithm in Figure 3), then retries the insert (line 19).

```
1 :void insert(cell *c, body *b, int *x) {
2 :  int i;
3 :  node *n;
4 :  leaf *l;
5 :
6 :  i = octant(x,c->level);
7 :  n = c->children[i];
8 :  if (n == NULL) {
9 :    l = >newleaf();
10:    l->level = c->level >> 1;
11:    c->children[i] = l;
12:    l->bodies[l->numberOfBodies] = b;
13:    l->numberOfBodies++;
14:  } else {
15:    if (n->cellOrLeaf == LEAF) {
16:      l = (leaf*) n;
17:      if (l->numberOfBodies == MAXBODIES) {
18:        divide(c, l, i);
19:        insert(c, b, x);
20:      } else {
21:        l->bodies[l->numberOfBodies] = b;
22:        l->numberOfBodies++;
23:      }
24:    } else {
25:      insert((cell *) n, b, x);
26:    }
27:  }
28:}
```

Figure 2. Original Sequential Insert Algorithm

```
1 :void divide(cell *c, leaf *l, int i) {
2 :  cell *newc;
3 :  leaf *newl;
4 :  body *p;
5 :
6 :  int index, x[NDIM];
7 :
8 :  newc = newcell();
9 :  newc->level = (c->level) >> 1;
10:  c->children[i] = newc;
11:
12:  for (int j = 0; j < l->numberOfBodies; j++) {
13:    p = l->bodies[j];
14:    getCoordinates(p, x);
15:    index = octant(x, l->level);
16:    newl = (leaf*) (newc->children[index]);
17:    if (newl == NULL) {
18:      newl = newleaf();
19:      newl->level = (newc->level) >> 1;
20:      newc->children[index] = newl;
21:    }
22:    newl->bodies[newl->numberOfBodies] = p;
23:    newl->numberOfBodies++;
24:  }
25:}
```

Figure 3. Original Sequential Divide Algorithm

If the tree contains a cell for the octant, the algorithm calls itself recursively to step down the tree towards the leaf into which it will eventually insert the body `b` (line 25).

Figure 3 presents the original divide algorithm. The algorithm allocates a new cell (lines 8-9), links it into the tree (line 10, disconnecting the leaf `l` that `c->children[i]` previously referenced), then inserts the bodies from the leaf `l` into the subtree rooted at the new cell `newc` (lines 12-23).

3.1 Fatal Data Races In Unsynchronized Parallel Version

We consider an unsynchronized parallel version of this algorithm that invokes the `insert` algorithm for the N bodies in the system in parallel with no additional synchronization. There are multiple opportunities for data races. Consider, for example, a case when many bodies (specifically, more than `MAXBODIES`) fall into the same leaf and are inserted in parallel. The insertions of these bodies may hit the check at line 17 of Figure 2 at the same time, with all of the checks finding that there is room to insert the body. The insertions next proceed to lines 21-22 of Figure 2, potentially overflowing the `bodies` array and causing out of bounds accesses.

There are also other opportunities for race conditions to overflow the `bodies` array, specifically interactions between the leaf insertions at lines 22-23 of Figure 3 and lines 21-22 of Figure 2. One of the causes of these fatal race conditions is the fact that the new cell and new leaves in the original `divide` algorithm in Figure 3 are linked into the tree (and therefore made reachable to insertions of other bodies) before they are fully constructed.

We have implemented and tested this parallel algorithm. We note that, in practice, these race conditions are always fatal — the parallel version of this tree construction algorithm always crashes.

3.2 Tree Locking

A standard way to synchronize tree insertion algorithms is to use *tree locking* — to lock each cell as the algorithm walks down the tree to find the location at which to insert the body. We have implemented and tested a version of this standard parallel tree construction algorithm. Although correct, it is hopelessly inefficient for this task — it always takes longer to execute than the original sequential tree construction algorithm, with the execution time growing as the number of cores executing the computation increases (see Section 6).

4. First Parallel Algorithm

Figure 4 presents our first unsynchronized parallel `insert` algorithm. This algorithm executes without crashing and preserves the Execution Properties from Section 2.2.3. There are two main differences from the original sequential algorithm. The first main difference is that the statement that links the new leaf into the tree (line 13, Figure 4) has been moved until after the statements that insert the body into the new leaf (lines 11-12, Figure 4). Because the new leaf is inaccessible to other parallel threads until it is linked into the tree, the insertion of the body into the leaf executes atomically without violating the bounds of the `bodies` array. In terms of our Execution Properties, the insertion preserves the Relaxed Leaf Representation property (which is the key property that executing the original sequential algorithm in parallel may violate).

The second main difference is that the check for a full leaf (lines 17-18, Figure 4) and insertion of the body `b` into leaf if it is not full (lines 22-23, Figure 4) are coded to avoid violating the bounds of the `bodies` array even in the presence of interference in the form of data races with other parallel insertions. The code sequence stores the number of bodies currently in the leaf into the variable `p` (line 17, Figure 4). It then uses `p` to check if the leaf is full (line 18, Figure 4). If it is not, it places the reference to the new

```
1 :void insert(cell *c, body *b, int *x, int t) {
2 :  int i, p;
3 :  node *n;
4 :  leaf *l;
5 :
6 :  i = octant(x,c->level);
7 :  n = c->children[i];
8 :  if (n == NULL) {
9 :    l = newleaf(t);
10:    l->level = c->level >> 1;
11:    l->bodies[l->numberOfBodies] = b;
12:    l->numberOfBodies++;
13:    c->children[i] = l;
14:  } else {
15:    if (n->cellOrLeaf == LEAF) {
16:      l = (leaf*) n;
17:      p = l->numberOfBodies;
18:      if (p == MAXBODIES) {
19:        divide(c, l, i, t);
20:        insert(c, b, x, t);
21:      } else {
22:        l->bodies[p] = b;
23:        l->numberOfBodies = p+1;
24:      }
25:    } else {
26:      insert((cell *) n, b, x, t);
27:    }
28:  }
29:}
```

Figure 4. First Parallel Insert Algorithm

```
1 :void divide(cell *c, leaf *l, int i, int t) {
2 :  cell *newc;
3 :  leaf *newl;
4 :  body *p;
5 :
6 :  int index, x[NDIM];
7 :
8 :  newc = newcell(t);
9 :  newc->level = (c->level) >> 1;
10:
11:  for (int j = 0; j < l->numberOfBodies; j++) {
12:    p = l->bodies[j];
13:    getCoordinates(p, x);
14:    index = octant(x, l->level);
15:    newl = (leaf*) (newc->children[index]);
16:    if (newl == NULL) {
17:      newl = newleaf(t);
18:      newl->level = (newc->level) >> 1;
19:      newc->children[index] = newl;
20:    }
21:    newl->bodies[newl->numberOfBodies] = p;
22:    newl->numberOfBodies++;
23:  }
24:  c->children[i] = newc;
25:}
```

Figure 5. First Parallel Divide Algorithm

body in element p of the bodies array (line 22, Figure 4) and sets l->numberOfBodies to reference the next element after p (line 23 Figure 4). This approach eliminates the multiple reads to the numberOfBodies field in the original algorithm from Figure 2 and ensures that the insert algorithm does not violate the bounds of the bodies array.

We also add the thread id t of the thread executing the insert as a parameter to insert and divide. The leaf allocation procedure newleaf(t) and cell allocation procedure newcell(t) now take this thread id t as a parameter. Each thread has its own allocation pool of leaves and cells so that it can allocate leaves and cells locally without synchronizing with other threads. The thread id t tells the allocation procedures which pool to allocate from.

Figure 5 presents the first parallel divide algorithm (the insert algorithm in Figure 4 invokes this algorithm at line 19). The main difference between this algorithm and the original divide algorithm from Figure 3 is that the statement that links the new cell into the tree (line 24 in Figure 5) has been moved until after the new cell has been fully constructed. This change ensures that the cell is not reachable to other parallel insertions while it is under construction (lines 8-23, Figure 5). The construction therefore executes atomically and preserves all of the Execution Properties.

4.1 Survivable Data Races

Of course, even though this algorithm preserves the Execution Properties, it still contains potential data races. We make this concept precise for this algorithm as follows. We say that there is a *data race* when the parallel execution causes sequences of instructions from insertions of different bodies to interleave in a way that the parallel execution produces a tree that violates one or more of the Natural Properties from Section 2.2.1. When we reason about parallel executions we assume the executions take place on a parallel machine that implements sequential consistency.

4.2 Tree Construction Operations

We discuss the potential data races further by first dividing the tree construction algorithm into *operations* whose atomic execution renders the algorithm free of data races. Each operation consists of a *check* (which examines part of the data structure to determine if a given condition holds, in which case we say that the check succeeds) and an *action* (a data structure update that the algorithm performs if the check succeeds).

- **New Leaf:** The check for a null octant reference (lines 7-8, Figure 4) in combination with the action that creates a new leaf to hold the inserted body b (lines 9-13, Figure 4).
- **Insert Body:** The check that determines that the next octant down is a leaf l with room to insert the body b (lines 7-8 and 15-18, Figure 4) in combination with the action that inserts the body b in the next available element of the bodies array in the leaf l.
- **Divide Leaf:** The check that determines that the next octant down is a leaf l that is full and therefore does not have room to insert the body b (lines 7-8 and 15-18, Figure 4) in combination with the action that replaces the leaf l with a cell containing the bodies in the leaf l (the call to divide, line 19, Figure 4).
- **Retry:** The recursive call to insert that retries the insert after the call to divide replaces the leaf l with a new cell.
- **Downward Step:** The check that determines that the next octant down is a cell (lines 7-8 and 15, Figure 4) in combination with the action that calls insert recursively on cell n for that octant (line 26, Figure 4, strictly speaking not an action because it does not update the data structure).

We next analyze the effects of potential data races. We consider data races associated with two parallel insertions (data races involving more operations typically have similar effects):

- **New Leaf vs. New Leaf:** It is possible for the check at line 8 of Figure 4 to succeed in both insertions. In this case both insertions will allocate a new leaf containing the inserted body and link the new leaf into the tree at line 13 of Figure 4. One of the new leaves will be discarded and the final tree will not contain the corresponding body.
- **New Leaf vs. Insert Body:** Because the New Leaf operation links the new leaf into the tree as its last step, it is not possible for the New Leaf operation to interleave with an Insert Body operation on the same leaf. So there is no data race associated with this combination of operations.
- **New Leaf vs. Divide Leaf:** It is possible for the check at line 8 of Figure 4 to succeed in one insertion, then other insert operations allocate a leaf at that same position in the tree, then fill that leaf with bodies. Then the Divide Leaf operation attempts to replace that leaf with a cell. The instructions at line 24 of Figure 5 (which links the cell into the tree) and line 13 of Figure 4 race. If the instruction from the Divide Leaf opertion that inserts the cell executes first, the subsequent instruction from the New Leaf operation will remove the cell (and all of the bodies that it contains) from the tree. If, on the other hand, the instruction from the New Leaf operation executes first, the subsequent instruction from the Divide Leaf operation will remove the new leaf (and the body that it contains) from the tree. In both cases the final tree will not contain some of the inserted bodies.
- **Insert Body vs. Insert Body:** It is possible for the check at lines 7-8 and lines 15-18 of Figure 4 to determine that there is room in the leaf l in both insertions. Both insertions proceed on to place the body in the leaf l at lines 22-23 of Figure 4. Both insertions have the same value of p. One of the insertions will execute first, the other will execute second and remove the first body from the tree. The final tree will not contain this first body.
- **Insert Body vs. Divide Leaf:** When the divide operation replaces a leaf with a cell, it iterates over all of the bodies in the leaf to reinsert them into the tree rooted at the new cell (lines 11-23, Figure 5). After the loop terminates, it is possible for an Insert Body operation to insert another body b into the replaced leaf node. The assignment at line 24 of Figure 5 will then remove the leaf (and the body b that it contains) from the tree. The final tree will not contain the body b.
- **Divide Leaf vs. Divide Leaf** It is possible for two insertions to both determine (via the check at lines 7-8 and 15-18, Figure 4) that the leaf l at the current octant is full. In this case both insertions will invoke the divide algorithm to replace the leaf l with a new cell. In this case the tree will temporarily contain the cell from the first operation to execute line 24 of Figure 5 (which links the new cell into the tree), then contain the cell from the second operation to execute this line.

The effect of all of these data races is to remove inserted bodies from the tree. For this reason the parallel algorithm satisfies the Relaxed Body Inclusion property from Section 2.2.2 rather than the stricter Body Inclusion property from Section 2.2.1. Note that we do not consider any races involving Retry or Downward Step operations as these operations do not modify the data structure and the effect of any interaction with other operations will show up via other considered data races.

4.3 One Three Way Data Race

We next discuss one three-way data race: the Insert Body vs. Insert Body vs. Insert Body data race. In this race, the first two Insert Body operations obtain the same value of `p` and will therefore insert their bodies into the same element `p` of the `bodies` array. The first Insert Body operation completes and, at line 23 of Figure 4, changes `l->numberOfBodies` to index the next free element of the `bodies` array.

Before the second Insert Body operation executes the instruction at line 23 of Figure 4, the third Insert Body operation executes to completion. At that point the second Insert Body operation executes line 23 of Figure 4, setting `l->numberOfBodies` back to its value before the execution of the third Insert Body operation.

With this data race 1) the `bodies` array does not reference one of the bodies from the first two Insert Body operations (the operation that executed line 22 in Figure 4 second overwrote this reference) and 2) the `bodies` array does reference the body from the third operation, but at index `l->numberOfBodies` in the `bodies` array. It is for this reason that we work with the Relaxed Leaf Representation property from Section 2.2.2 rather than the more strict Leaf Representation property from Section 2.2.1.

Note that a subsequent insert into the leaf may overwrite the reference inserted by the third Insert Body operation. Even if no such insert occurs and the reference remains in the array, any computation that uses `l->numberOfBodies` to traverse the references in the `bodies` array will terminate the traversal before it accesses the reference. The net effect is that, with this data race, the final tree does not contain two of the three inserted bodies.

5. Final Parallel Algorithm

The number of bodies that data races remove from the final tree depends on the frequency with which the data races occur. While our final algorithm does not eliminate data races altogether, it uses a technique (which we call *final check*) to reduce the data race frequency. The technique operates as follows.

The tree construction operations follow the general pattern of first performing a check to determine which action to perform, next executing a sequence of instructions that to construct a new part of the data structure (the Insert Body operation does not execute such a sequence), then executing one or more instructions to commit its updates from the action into the main tree data structure. All of the data races change the result of the check so that the check would no longer produce the same result if executed at the time when the operation committed the updates from its action. We call the time between the check and the commit the *window of vulnerability*.

Our final check technique shrinks (but does not eliminate) the window of vulnerability by performing part or all of the check again just before the instructions that commit the action (if the final check retries the entire first check, we say that it is a *full final check*, otherwise we say that it is a *partial final check*).

If the final check produces a different result than the first check, the algorithm discards its action and retries the insert. It is possible to retry immediately or defer the retry until later, for example by storing the body in a data structure for later processing. We present a final algorithm that retries immediately; we have also implemented an algorithm that defers the retries until each thread has attempted to insert all of its initially assigned set of bodies.

Figure 6 presents the final parallel tree construction algorithm. Figure 7 presents the final parallel tree divide algorithm. The final checks occur at lines 14 and 30 of Figure 6 and line 25 of Figure 7. The retries (immediate in this case) appear at lines 18, 27, 46 in Figure 6 (note that the retry at line 27 is present in the first parallel algorithm in Section 4).

```
1 :void insert(cell *c, body *b, int *x, int t) {
2 :  int i, p;
3 :  node *n;
4 :  leaf *l;
5 :
6 :  i = octant(x,c->level);
7 :  n = c->children[i];
8 :  if (n == NULL) {
9 :    l = newleaf(t);
10:   l->level = c->level >> 1;
11:   l->bodies[l->numberOfBodies] = b;
12:   l->numberOfBodies++;
13:   // final check (full)
14:   if (c->children[i] == NULL) {
15:     c->children[i] = l;
16:   } else {
17:     // retry
18:     insert(c, b, x, t);
19:   }
20: } else {
21:   if (n->cellOrLeaf == LEAF) {
22:     l = (leaf*) n;
23:     p = l->numberOfBodies;
24:     if (p == MAXBODIES) {
25:       divide(c, l, i, t);
26:       // retry after divide
27:       insert(c, b, x, t);
28:     } else {
29:       // final check (partial)
30:       if (l->bodies[p] == NULL) {
31:         l->bodies[p] = b;
32:         l->numberOfBodies = p+1;
33:       } else {
34:         // data structure repair
35:         // to avoid infinite loop
36:         int j = l->numberOfBodies;
37:         while (true) {
38:           if ((l->bodies[j] == NULL) ||
39:               (j == MAXBODIES)) {
40:             l->numberOfBodies = j;
41:             break;
42:           }
43:           j++;
44:         }
45:         // retry
46:         insert(c, b, x, t);
47:       }
48:     }
49:   } else {
50:     insert((cell *) n, b, x, t);
51:   }
52: }
53:}
```

Figure 6. Final Parallel Insert Algorithm

```
1 :void divide(cell *c, leaf *l, int i, int t) {
2 :  cell *newc;
3 :  leaf *newl;
4 :  body *p;
5 :
6 :  int index, x[NDIM];
7 :
8 :  newc = newcell(t);
9 :  newc->level = (c->level) >> 1;
10:
11:  for (int j = 0; j < l->numberOfBodies; j++) {
12:    p = l->bodies[j];
13:    getCoordinates(p, x);
14:    index = octant(x, l->level);
15:    newl = (leaf*) (newc->children[index]);
16:    if (newl == NULL) {
17:      newl = newleaf(t);
18:      newl->level = (newc->level) >> 1;
19:      newc->children[index] = newl;
20:    }
21:    newl->bodies[newl->numberOfBodies] = p;
22:    newl->numberOfBodies++;
23:  }
24:  // final check (full)
25:  if (c->children[i] == l) {
26:    c->children[i] = newc;
27:  }
28:  // caller retries regardless of whether
29:  // final check succeeds or not
30:}
```

Figure 7. Final Parallel Divide Algorithm

5.1 Full vs. Partial Checks

The final checks at lines 13 of Figure 6 and 25 of Figure 7 are both full final checks — if the check succeeds, all parts of the tree data structure that the first check accessed are in the same state (so the first check, if rerun, would give the same result). The final check at line 30 of Figure 6, however, is partial — it does not check if `c->children[i]` still references the leaf `l`. Replacing line 30 in Figure 6 with the following line:

```
30:        if (((node *) l == c->children[i]) &&
               (l->bodies[p] == NULL)) {
```

makes the final check full instead of partial.

5.2 Full Final Checks and Atomicity

We have implemented a synchronized version of the algorithm in which 1) all operations use full final checks and 2) each cell object `c` contains a mutual exclusion lock that the algorithm uses to make all operations execute atomically (recall that each operation consists of a check plus an action, see Section 4.2). This synchronized version acquires the lock in the accessed cell object `c` just before the check and releases the lock immediately after the action. This version has no data races and always produces a tree that satisfies the Natural Properties from Section 2.2.1.

5.3 Data Structure Repair

Lines 37-44 of Figure 6 implement a data structure repair algorithm that enables the algorithm to avoid an infinite loop. Recall that the three way data race discussed in Section 4.3 may leave a leaf `l` in a state in which `bodies[l->numberOfBodies]` is not `NULL`. In this case the final check at line 30 will always fail and the insertion will infinite loop. The data structure repair algorithm avoids this infinite loop by reseting `l->numberOfBodies` so that either 1) `bodies[l->numberOfBodies]` is `NULL`, so that the insertion will succeed in the retry (in the absence of interference from other parallel insertions), or 2) if all entries of the `bodies` array are non-`NULL`, `l->numberOfBodies` is set to `MAXBODIES` so that the retry will (again, in the absence of interference from other parallel insertions) invoke the `divide` algorithm to create space in the tree for the body. In all cases the data structure repair enables the insertion to make progress and avoid the infinite loop.

5.4 Iterative Extension to Insert Missing Bodies

It is possible to extend the final algorithm to produce a tree that contains all of the inserted bodies as follows. This extension traverses the final tree to determine which, if any, bodies are missing. It then retries the insertion of the missing bodies, iterating until the tree contains all of the N bodies in the system. A drawback of this approach is the time required to determine which bodies are missing. An efficient algorithm for this task would make this approach viable.

6. Experimental Results

We implemented the tree construction algorithms described in Sections 4 and 5 in C++ using the `pthreads` threads package. This version executes both the tree construction phase and the force computation phase in parallel. In the tree construction phase each thread inserts a block of N/T bodies into the tree, where N is the number of bodies in the system and T is the number of parallel threads. Similarly, each thread in the force computation phase computes the forces acting on a block of N/T bodies. The threads in each parallel implementation execute a barrier (`pthread_barrier_wait`) before and after the tree construction and force computation phases.

We report results for the following versions of the algorithm:

- **Original Sequential:** The original sequential version of the algorithm with no parallelization overhead.
- **Tree Locking:** A parallel algorithm that uses standard tree locking to synchronize tree insertions. Each cell contains a mutual exclusion lock (`pthread_mutex_t`). Each insertion uses standard `pthreads` mutual exclusion primitives to acquire (`pthread_mutex_lock`) and release (`pthread_mutex_unlock`) the lock on each cell that it visits as it walks down the tree.
- **First Parallel:** The synchronization-free first parallel algorithm described in Section 4.
- **Final Check:** The synchronization-free parallel algorithm described in Section 5 that performs partial final checks.
- **Synchronized:** The synchronized parallel algorithm described in Section 5.2. Each cell contains a mutual exclusion lock. The algorithm uses this mutual exclusion lock to make the each operation (which consists of a full final check and an action) execute atomically. This version executes without data races and always produces a tree that satisfies the Natural Properties (see Section 2.2.1).
- **Hyperaccurate:** The Synchronized version above, but running with an smaller `tol` parameter (the original `tol` parameter divided by 1.25). With this parameter, the force computation phase goes deeper into the space-subdivision tree before it approximates the effect of multiple distant bodies with their center of mass. We use this version to evaluate the accuracy consequences of dropping bodies from the space-subdivision tree as compared with making the force computation phase more accurate.

We refer to the First Parallel and Final Check versions as the Unsynchronized versions. We run all versions on a 16 core 2.4 GHz

	Number of Cores				
Version	1	2	4	8	16
Tree Locking	17.21 (0.50)	19.35 (0.44)	18.35 (0.46)	23.46 (0.26)	39.38 (0.22)
First Parallel	8.49 (1.00)	5.56 (1.53)	3.25 (2.62)	1.64 (5.20)	1.01 (7.77)
Final Check	8.44 (1.01)	5.38 (1.59)	3.25 (2.63)	1.72 (4.96)	1.11 (7.70)
Synchronized	9.64 (0.88)	6.33 (1.35)	3.79 (2.25)	2.01 (4.24)	1.323 (6.44)

Table 1. Performance numbers for tree construction phase. Each entry is of the form entry of the form X (Y) where X is the execution time and Y is the speedup.

	Number of Cores				
Version	1	2	4	8	16
Parallel	529.06 (1.00)	264.60 (2.00)	112.70 (4.70)	69.47 (7.62)	44.85 (11.80)
Hyperaccurate	632.34 (0.84)	317.00 (1.67)	137.03 (3.87)	82.78 (6.39)	52.29 (10.12)

Table 2. Performance numbers for force computation phase. Each entry is of the form X (Y) where X is the execution time and Y is the speedup.

	Number of Cores				
Version	1	2	4	8	16
Tree Locking	17.21, 0.00 (0.0%)	38.43, 0.26 (0.6%)	70.68, 2.73 (3.7%)	255.88, 3.75 (1.4%)	613.339, 11.83 (1.9%)
First Parallel	8.49, 0.00 (0.0%)	10.85, 0.29 (2.6%)	12.02, 0.97 (7.5%)	12.55, 0.56 (4.3%)	16.055, 1.62 (9.2%)
Final Check	8.44, 0.0 (0.0%)	10.57, 0.17 (1.6%)	12.01, 0.98 (7.5%)	12.62, 1.05 (7.7%)	16.23, 1.60 (9.0%)
Synchronized	9.69, 0.0 (0.0%)	12.75, 0.16 (1.2%)	14.04, 1.18 (7.8%)	15.15, 1.09 (6.7%)	18.99, 2.36 (11.1%)

Table 3. Working and waiting times for tree construction phase. Each entry is of the form X, Y (Z%) where X is the total working time summed over all cores, Y is the total barrier waiting time summed over all cores, and Z is the percentage of total execution time spent barrier waiting.

Intel Xeon E7340 machine with 16 GB of RAM running Debian Linux kernel version 2.6.27. We report results from executions that simulate 200 steps of a system with 128K bodies.

6.1 Performance

Table 1 presents performance numbers for the different space-subdivision tree construction algorithms. There is a row in the table for each version. The first column in each row identifies the version of the algorithm. The following columns present timing results from running the algorithm on different numbers of cores. Each entry is of the form X (Y), where X is the tree construction time and Y is the speedup over the original sequential tree construction algorithm (running with no parallelization overhead). The number at the top of each column identifies the number of cores running the computation (we report results for 1, 2, 4, 8, and 16 core executions).

We note that the Tree Locking version exhibits poor performance — it takes twice as long to run as the original version on one core, and the performance decreases as the number of cores increases. We attribute the poor performance to a combination of synchronization overhead and bottlenecks associated with locking the top cells in the tree.

The First Parallel and Final Check versions exhibit almost identical performance with good performance for all numbers of cores. The Synchronized version also scales reasonably well, but has reduced performance in comparison with the First Parallel and Final Check versions. We attribute this reduced performance to synchronization overhead (note the increase in the single core execution time relative to the Original Sequential version).

For comparison purposes Table 2 presents performance numbers for the parallel force computation phase. We note that this phase is much more computationally expensive (and therefore consumes more of the execution time) than the tree construction phase. It also scales better than the tree construction phase as the number of cores increases — the force computation phase is known to be more amenable to parallelization than the tree construction phase [30]. The numbers in Table 2 also illustrate the effect of increasing the precision of the force computation phase by decreasing the `tol` parameter.

We note that these executions use a small value for the `tol` parameter. This parameter determines how high in the tree the force computation phase terminates the traversal and approximates the remote force with the center of mass below the current cell in the traversal. Smaller versions provide more accurate approximations but cause the force computation phase to run longer.

Table 3 presents the total working and barrier waiting times for different versions of the tree construction phase running with different numbers of cores. Each entry is of the form X, Y (Z%), where X is the total time spent inserting bodies (summed over all cores), Y is the total time spent waiting at the barrier at the end of the tree construction phase (again, summed over all cores), and Z is the percentage of time spent waiting at the barrier. We note that, as the number of cores increases, the working time for the First Parallel and Final Check versions increases. We attribute this increase to memory system effects. We also note that because the computation spends a small (typically much less than 10%) of its time waiting at the barrier, it balances the computational load effectively. For these reasons, we identify the primary source of performance degradation for these two versions as memory system effects.

6.2 Accuracy

Table 4 reports the cumulative number of dropped bodies for the First Parallel and Final Check versions. Each number is the total number of bodies dropped from the space-subdivision tree over all 200 steps. We note two properties:

- **Low Drop Rate:** Over the course of the simulation, each algorithm inserts a total of 128K * 200 bodies into the 200 con-

	Number of Cores				
Version	1	2	4	8	16
First Parallel	0	3,689	11,262	28,877	52,864
Final Check	0	769	1,986	3,764	7,703

Table 4. Cumulative number of dropped bodies after 200 simulation steps.

	Number of Cores				
Version	1	2	4	8	16
First Parallel	1.02%, 0.000%	1.02%, 0.002%	1.02%, 0.013%	1.02%, 0.036%	1.02%, 0.045%
Final Check	1.02%, 0.000%	1.02%, 0.001%	1.02%, 0.002%	1.02%, 0.007%	1.02%, 0.012%
Synchronized	1.02%, 0.000%	1.02%, 0.000%	1.02%, 0.000%	1.02%, 0.000%	1.02%, 0.000%

Table 5. Distances between body positions computed by different versions of the simulation after 200 simulation steps. Each entry is of the form X, Y where X is Φ_H^S and Y is either Φ_S^{FP} (First Parallel row), Φ_S^{FC} (Final Check row), or Φ_S^S = 0.000% (Synchronized row).

structed trees. Even running on 16 cores with no final check to reduce the number of dropped bodies, the First Parallel algorithm drops only 0.2% of the bodies it inserts. Other versions drop significantly fewer bodies.

- **Final Check Effectiveness:** The final check algorithm is effective at reducing the number of dropped bodies, specifically by a factor of between 5 and 7 depending on the number of cores executing the simulation.

We next consider the effect of the dropped bodies on the overall accuracy of the simulation. Starting from the same state, we run the Hyperaccurate, Synchronized, and (in different runs) either the First Parallel or the Final Check versions in lockstep. Each step in each simulation produces a three-dimensional position for each body. We compute the following measures of how much the simulations differ in the results that they produce:

- Δ_H^S : The sum of the distances between corresponding bodies in two simulations: one that uses the Hyperaccurate algorithm and one that uses the Synchronized algorithm. This quantity provides an estimate of how much accuracy is lost because of the center of mass approximation in the force computation phase.
- Δ_S^{FP} : The sum of the distances between corresponding bodies in two simulations: one that uses the Synchronized algorithm and one that uses the First Parallel algorithm.
- Δ_S^{FC} : The sum of the distances between corresponding bodies in two simulations: one that uses the Synchronized algorithm and one that uses the Final Check algorithm.

We also compute percentage differences Φ_H^S, Φ_S^{FP}, Φ_S^{FC}, Φ_H^{FP}, and Φ_H^{FC}, which are the corresponding Δs as a percentage of the distance between the far corners of the bounding box of the bodies in the corresponding step of the Synchronized simulation. So to compute a Φ, we first obtain the corresponding Δ by summing up the distances between corresponding bodies, divide the sum by the distance between the far corners of the bounding box, then multiply by 100. Table 5 reports the different Φs for the positions of the bodies after 200 simulation steps.

We note that by the triangle inequality, $\Phi_H^{FP} \leq \Phi_H^S + \Phi_S^{FP}$, $\Phi_H^{FP} \leq \Phi_H^S + \Phi_S^{FP}$, and similarly for the corresponding Δs. We also note that while the triangle inequality provides an upper bound on Φ_H^{FC} and Φ_H^{FP}, the actual Φ_H^{FC} and Φ_H^{FP} that we compute from the computed body positions are significantly smaller than this upper bound. Specifically, they are the same as $\Phi_H^S = 1.02\%$ to three significant digits.

We highlight the following facts:

- **Hyperaccurate Differences:** Both the Synchronized and Unsynchronized versions produce results that differ from the Hyperaccurate results by essentially the same amount: 1.02% (at three significant digits of precision).
- **Synchronized vs. Unsynchronized Differences:** The results from the Synchronized and Unsynchronized versions differ from each other by at most hundredths of percentage points. This is a very small difference, especially in comparison with the differences between the Hyperaccurate and Synchronized results, which are typically two orders of magnitude larger.
- **First Parallel vs. Final Check Differences:** The reduced number of dropped bodies that the Final Check version produces translates into more accurate results, specifically results that are three to six times closer to the Synchronized result at four processors and above.

These facts support the acceptability of the Unsynchronized versions. The rationale is that, in comparison with the Hyperaccurate version, the Synchronized and Unsynchronized versions provide results with essentially identical accuracy. Moreover, the Synchronized and Unsynchronized versions provide approximations that are typically two orders of magnitude closer to each other than they are to the more accurate Hyperaccurate results. Even if we use the loose triangle inequality bound, for the Synchronized version to be acceptable and the Final Check version unacceptable, it must be the case that a result that differs from the Hyperaccurate result by 1.02% is acceptable, but a result that differs by 1.032% is not acceptable. The corresponding numbers for the First Parallel version are 1.02% and 1.065%.

6.3 Accuracy Over Time

To better understand the effect of the Unsynchronized versions on the accuracy of the simulation as the number of steps increases, we recorded Δ_H^S, Δ_S^{FP}, and Δ_S^{FC} for each of the 200 steps s of the simulation. Plotting these points reveals that they form linear curves characterized by the following equations:

$$\Delta_H^S(s) = 245731s + 90867$$

$$\Delta_S^{FP}(s) = 10880s + 11883$$

$$\Delta_S^{FC}(s) = 2949.5s + 8731.5$$

These equations show that the difference between the Hyperaccurate and Synchronized versions grows more than twenty times as fast as the difference between the Synchronized and Unsynchronized versions as the number of simulation steps increases. Our interpretation of this result is that the Unsynchronized versions will remain as acceptably accurate as the Synchronized version even for long simulations with many steps.

We note that in our simulation, the system is expanding faster than the Δs are growing. So even though the Δs increase as the simulation executes more steps, the Φs decrease.

7. Related Work

Researchers have invested significant time and effort in obtaining parallel versions of hierarchical N-body simulations, including parallel space-subdivision tree construction algorithms [28, 30]. Even though (depending on the parameterization) the force computation phase may consume more than 97% of the total sequential execution time the tree construction phase can still become the major performance bottleneck in the parallel execution [28].

Synchronization is identified as necessary to preserve the correctness of parallel tree construction algorithms [30], with the synchronization overhead driving the design of an algorithm that first constructs local trees on each processor, then merges the local trees to obtain the final tree. Because the local tree construction algorithms execute without synchronization (the merge algorithm must still synchronize its updates as it builds the final tree) this approach reduces, but does not eliminate, the synchronization required to build the tree. The algorithm presented in this paper, in contrast, executes without synchronization and may therefore produce partially corrupted trees. Our results show that these trees are still acceptable in the broader context of the N-body simulation.

The QuickStep compiler automatically generates parallel code for a wide range of loops in both standard imperative and object-oriented programs [15, 16]. Unlike other parallelizing compilers, QuickStep is designed to generate parallel programs that may contain data races that change the output of the program. QuickStep uses training runs to distinguish acceptable data races from unacceptable data races. If the generated parallel program satisfies a probabilistic accuracy bound, the race is acceptable, otherwise it is unacceptable. QuickStep eliminates unacceptable races by either inserting synchronization that eliminates the race, replicating data to eliminate the race, or by simply generating sequential code for the loop that contains the data race. Like QuickStep, the research presented in this paper is designed to produce parallel programs with data races that acceptably change the output of the program.

The Cilkchess parallel chess program uses concurrently accessed transposition tables [5]. Standard semantics require synchronization to ensure that the accesses execute atomically. The developers of Cilkchess determined, however, that the probability of losing a match because of the synchronization overhead was larger than the probability of losing a match because of unsynchronized accesses corrupting the transposition table. They therefore left the parallel accesses unsynchronized (so that Cilkchess contains data races) [5]. Like Cilkchess, our parallel tree insertion algorithm improves performance by purposefully eliminating synchronization and therefore contains acceptable data races.

We are aware of an effort to develop a synchronization-free parallel Delauney triangulation algorithm [36]. Like the parallel space-subdivision tree algorithm presented in this paper, the triangulation algorithm contained data races that could corrupt the data structure and data structure repair algorithms designed to repair enough of the inconsistencies to produce an acceptably consistent final triangulation in spite of the data races. One of the challenges associated with this research was obtaining and interfacing with a client that could provide a suitable accuracy metric for evaluating the acceptability of the final data structure that the algorithm produced.

The race-and-repair project has developed a synchronization-free parallel hash table insertion algorithm [32]. Like our parallel tree construction algorithm, this algorithm may drop inserted entries. An envisioned higher layer in the system recovers from any errors that the absence of inserted elements may cause.

This paper presents an algorithm that works with clients that simply use the tree as produced by the algorithm with no higher layer to deal with dropped bodies (and no need for such a higher layer). Because we evaluate the algorithm in the context of a complete computation (the Barnes-Hut N-body simulation), we develop an end-to-end accuracy measure and use that measure to evaluate the overall end-to-end acceptability of the algorithm. This measure enables us to determine that the relaxed semantics of the synchronization-free algorithm have acceptable accuracy consequences for this computation.

Chaotic relaxation runs iterative solvers without synchronization [2, 4, 31]. Convergence theorems prove that the computation will still converge even in the presence data races. The performance impact depends on the specific problem at hand — some converge faster with chaotic relaxation, others more slowly.

Task skipping [21], early phase termination [22], and loop perforation [11, 17, 18, 29, 37] all increase performance by skipping parts of the computation. All of these techniques may (and typically do) change the output that the transformed program produces. All use training runs to characterize the accuracy consequences of applying the transformation. All accept transformations that produce acceptably small changes to the output. The algorithm presented in this paper is conceptually similar in that it may drop inserted bodies (instead of parts of the computation) to acceptably change the output that the computation produces. It differs in that the mechanism that changes the output is data races, not skipped parts of the computation.

Data structure repair [6–10] enables systems to recover from errors that damage the data structures with which that the system works. The Final Check algorithm presented in this paper contains a similar data structure repair mechanism that detects and eliminates a certain kind of data structure corruption. In this case the goal is to recover from the data race that introduced the corruption and avoid an infinite loop (see Section 5.3).

Cyclic memory allocation [19] eliminates memory leaks by preallocating a fixed-size buffer, then cyclically allocating memory out of that buffer. It is possible for new memory to overlay previously allocated memory that is still live. Like the data races in the tree construction algorithm presented in this paper, one observed effect is the acceptable deletion of elements of linked data structures.

Acceptability-oriented computing [20] is a set of techniques that are designed to keep a system operating acceptably in the face of errors or other anomalies that may perturb the execution of the system. One of the key techniques is to do nothing if the consequences of the errors or anomalies are acceptable. The algorithm presented in this paper employs a technique (data structure repair) that keeps the algorithm operating acceptably in the face of data structure corruption errors that might otherwise cause the program to infinite loop. It also identifies the data races as having other acceptable consequences (specifically, dropping inserted bodies) that it does not try to avoid or eliminate.

It is possible to view the unsynchronized algorithms presented in this paper as imperfect but acceptable implementations of the perfect synchronized algorithm. In this view the data races are the source of the imperfection. Other researchers have demonstrated that programs are often able to productively tolerate other kinds of errors [24], for example off by one errors in loop bounds. The acceptability of our unsynchronized tree construction algorithms provides additional evidence that software systems can often productively tolerate a surprisingly large range of errors.

Jolt [3] implements a technique that recognizes and escapes infinite loops. Failure-oblivious computing [23] returns a sequence of values for out of bounds accesses that is designed to enable a program to exit an infinite loop that reads beyond the end of an array or buffer. These techniques, along with the data structure repair

technique that eliminates the infinite loop in the Final Check algorithm presented in this paper, highlight the importance of infinite loop elimination techniques.

8. Conclusion

Since the inception of the field, developers of parallel algorithms have used synchronization to ensure that their algorithms execute correctly. In contrast, the basic premise of this paper is that parallel algorithms, to the extent that they need to contain any synchronization at all, need contain only enough synchronization to ensure that they execute correctly enough to generate an acceptably accurate result.

We show how this premise works out in practice by developing a synchronization-free parallel space-subdivision tree construction algorithm. Even though this algorithm contains data races, it produces trees that are consistent enough for the Barnes-Hut N-body simulation to use successfully. Our experimental results demonstrate the performance benefits and acceptable accuracy consequences of this approach.

References

[1] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[2] G. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 25:225–244, April 1998.

[3] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *ECOOP*, pages 609–633, 2011.

[4] D. Chazan and W. Mirankar. Chaotic relaxation. *Linear Algebra and its Applications*, 2:119–222, April 1969.

[5] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[6] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, pages 233–244, 2006.

[7] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[8] Brian Demsky and Martin C. Rinard. Static specification analysis for termination of specification-based data structure repair. In *ISSRE*, pages 71–84, 2003.

[9] Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005.

[10] Brian Demsky and Martin C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12), 2006.

[11] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, September 2009.

[12] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76, 2009.

[13] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[14] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[15] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems*. "to appear".

[16] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, August 2010.

[17] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[18] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin C. Rinard. Quality of service profiling. In *ICSE (1)*, pages 25–34, 2010.

[19] Huu Hai Nguyen and Martin C. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*, pages 15–30, 2007.

[20] Martin C. Rinard. Acceptability-oriented computing. In *OOPSLA Companion*, pages 221–239, 2003.

[21] Martin C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[22] Martin C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, pages 369–386, 2007.

[23] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[24] Martin C. Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In *OOPSLA Companion*, pages 21–30, 2005.

[25] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *PLDI*, pages 54–67, 1996.

[26] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, 1997.

[27] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *OSDI*, pages 101–114, 1994.

[28] Hongzhang Shan and Jaswinder Pal Singh. Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 475–484, 1998.

[29] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[30] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal Of Parallel and Distributed Computing*, 27:118–141, 1995.

[31] J. Strikwerda. A convergence theorem for chaotic asynchronous relaxation. *Linear Algebra and its Applications*, 253:15–24, March 1997.

[32] D. Ungar. Presentation at OOPSLA 2011, November 2011.

[33] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[34] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

[35] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351, 2009.

[36] Z. Zhang and M. Rinard. Synchronization-Free Parallel Delauney Triangulation Experiments, September 2010.

[37] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.