Parallel Synchronization-Free Approximate Data Structure Construction (Full Version)

Martin Rinard, MIT CSAIL

Abstract

We present approximate data structures with synchronization-free construction algorithms. The data races present in these algorithms may cause them to drop inserted or appended elements. Nevertheless, the algorithms 1) do not crash and 2) may produce a data structure that is accurate enough for its clients to use successfully. We advocate an approach in which the approximate data structures are composed of basic tree and array building blocks with associated synchronization-free construction algorithms. This approach enables developers to reuse the construction algorithms, which have been engineered to execute successfully in parallel contexts despite the presence of data races, without having to understand the complex details of why they execute successfully. We present C++ templates for several such building blocks.

We evaluate the end-to-end accuracy and performance consequences of our approach by building a space-subdivision tree for the Barnes-Hut N-body simulation out of our presented tree and array building blocks. In the context of the Barnes-Hut computation, the resulting approximate data structure construction algorithms eliminate synchronization overhead and potential anomalies such as excessive serialization and deadlock. Our experimental results show that the algorithm exhibits good performance (running 14 times faster on 16 cores than the sequential version) and good accuracy (the accuracy loss is four orders of magnitude less than the accuracy gain associated with increasing the accuracy of the Barnes-Hut center of mass approximation by 20%).

1. Introduction

Many computations (for example, video and image processing computations, modern internet search and information retrieval, and many scientific computations) are designed to produce only an approximation to an ideal output. Because such computations have the flexibility to produce any one of a range of acceptably accurate outputs, they can often productively skip tasks [20, 21] or loop iterations [9, 15, 16, 23, 29], or approximately memoize functions [4] as long as these transformations do not unacceptably degrade the accuracy of the output. Potential benefits of applying these and other approximate transformations include reduced power consumption [9, 13–16, 23, 29], increased performance [9, 13–16, 23, 29], and the ability to automatically adapt to changing conditions in the underlying computational platform [23].

1.1 Approximate Data Structures

In this paper we shift the focus to the data structures that the computations manipulate — we present synchronization-free parallel algorithms for building approximate data structures. These algorithms work with data structures composed of tree and array building blocks. Examples of such data structures include search trees, array lists, hash tables, linked lists, and space-subdivision trees. The algorithms insert elements at the leaves of trees or append elements at the end of arrays. Because they use only primitive reads and writes, they are free of synchronization mechanisms such as mutual exclusion locks or compare and swap. As a result, they execute without synchronization overhead or undesirable synchronization anomalies such as excessive serialization or deadlock. Moreover, unlike much early research in the field [12], the algorithms do not use reads and writes to synthesize higher-level synchronization constructs.

Now, it may not be clear how to implement correct data structure construction algorithms without sychronization. Indeed, we do not attempt to do so — the data races in our algorithms may drop inserted elements and violate some of the natural data structure consistency properties. But these algorithms 1) do not crash, 2) produce a data structure that is consistent enough for its clients to use successfully, and 3) produce a data structure that contains enough of the inserted elements so that its clients can deliver acceptably accurate outputs. In effect, we eliminate synchronization by leveraging the end-to-end ability of the client to tolerate some imprecision in the approximate data structure that the algorithm produces.

1.2 Building Blocks

Our approximate data structure construction algorithms have the advantage that they are, essentially, standard sequential algorithms that have been engineered to execute successfully without synchronization in parallel contexts despite the presence of data races. The reasons for this successful execution can be quite involved. We therefore advocate the development of general data structure building blocks (such as reusable tree and array components) with associated approximate construction algorithms. Because these building blocks encapsulate the reasoning required to obtain successful synchronization-free construction algorithms, developers can build their data structures out of these building blocks and reuse the construction algorithms without needing to understand the details of why the algorithms operate successfully without synchronization despite the presence of data races. We present several such building blocks implemented as C++ templates.

1.3 Advantages and Disadvantages

Our unsynchronized approximate data structures are standard, familiar sequential implementations that execute successfully in parallel contexts. They are completely free of the complex and potentially confusing synchronization primitives (for example, mutual exclusion locks, atomic transactions, and wait-free updates) that complicate the development of synchronized parallel code. Our unsynchronized approach can therefore offer the following advantages:

An abridged version of this paper appears in 5th USENIX Workshop on Hot Topics in Parallelism, June 24-25, San Jose, CA.

- **Enhanced Correctness and Trustworthiness:** There is no exposure to coding errors in complex synchronization primitives or in the complex, intellectually challenging parallel data structure implementations that use them.
- **Enhanced Portability:** There is no reliance on specialized synchronization primitives that may not be widely implemented across platforms.
- **Simplicity and Ease of Development:** There is no need to learn, use, or rely on complex synchronization primitives — our unsynchronized data structures use only standard read and write operations.

As a result, we anticipate that the simplicity, familiarity, and transparency of our approach may make synchronization-free approximate data structures easier for developers to understand, trust, and use than their complex, intimidating synchronized counterparts.

A potential disadvantage is that developers have been repeatedly told that unsynchronized concurrent accesses to shared data are dangerous. If developers internalize this rigid, incomplete, but currently widespread perception, they may find unsynchronized data structures emotionally difficult to accept even when they are the superior alternative for the task at hand.

1.4 Case Study

In general, we expect the acceptability of the approximate data structures to depend on end-to-end effects such as 1) how frequently the application's data structure construction workload elicits interactions that drop elements and 2) the effect that any dropped elements have on the accuracy of the result that the application produces. We therefore perform a case study that evaluates the performance and accuracy consequences of using our building blocks for the space-subdivision tree in the Barnes-Hut N-body computation [1, 24]. This computation simulates a system of N interacting bodies (such as molecules, stars, or galaxies). At each step of the simulation, the computation computes the forces acting on each body, then uses these forces to update the positions, velocities, and accelerations of the bodies.

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 compute the force acting on each body. It approximates the combined forces from multiple distant bodies as the force from the center of mass of the distant bodies as stored in the root of the subtree that includes these distant bodies. This approximation reduces the complexity of the force computation algorithm from N^2 to $N \log N$. We implement the space-subdivision tree itself as a hybrid data structure whose leaves use an array to store multiple inserted bodies.

1.5 Experimental Results

Because the approximate space-subdivision tree construction algorithm is unsynchronized, it may produce a tree that does not contain some of the inserted bodies. The net effect is that the force computation algorithm operates as if those bodies did not exist at that step. Our results show that, in practice, less than 0.0003% of the inserted bodies are dropped. The effect of these dropped bodies on the overall accuracy of the computation is negligible. Specifically, the effect on the computed body positions is four orders of magnitude less than the effect of increasing the accuracy of the center of mass approximation in the force calcuation phase by 20%.

The unsynchronized algorithm exhibits good parallel performance (on 16 cores, it runs over 14 times faster than the sequential tree construction algorithm) and runs over an order of magnitude faster than a version that uses standard tree locking to eliminate dropped bodies. It also runs 5% and 10% faster than sophisticated parallel implementations that use either fine-grained mutual exclusion locks or compare and swap operations, in combination with strategically placed retry operations, to eliminate dropped bodies.

1.6 Scope

There are deep conceptual connections between approximate data structures and techniques such as task skipping [20], loop perforation [9, 16, 23], early phase termination [21], infinite loop exit [3, 11], reduction sampling [29], and approximate parallel compilation [13, 17]. All of these techniques remove parts of the computation to produce a *perforated computation* — i.e., a computation with pieces missing. In the case of approximate data structures, there is a cascade effect — removing synchronization drops inserted bodies from the tree, which, in turn, has the effect of eliminating the dropped bodies from the force computation.

Many successful perforations target computations that combine multiple items to obtain a composite result — adding up numbers to obtain a sum, inserting elements to obtain a space-subdivision tree. One potential explanation for why applications tolerate perforation is that the perforations exploit a redundancy inherent in the multiple contributions. In effect, the original computations were overengineered, perhaps because developers try to manage the cognitive complexity of developing complex software by conservatively producing computations that they can easily see will produce a (potentially overaccurate) result.

Tolerating some inaccuracy is a prerequisite for the use of approximate data structures. Computations that are already inherently approximate are therefore promising candidates for approximate data structures. Note that this is a broad and growing class — any computation that processes noisy data from real-world sensors, employs standard approximations such as discretization, or is designed to please a subjective human consumer (for example, search, entertainment, or gaming applications), falls into this class. Applications such as video or dynamic web page construction may be especially promising — the results are immediately consumed and the effect of any approximations move quickly into the past and are forgotten. Traditional computations such as relational databases or compilers, with their hard notions of correctness, may be less promising candidates.

In our experience, many developers view banking as an application domain that does not tolerate imprecision and is therefore not appropriate for approximate data structures or algorithms. In practice, the integrity of the banking system depends on periodic reconciliations, which examine transactions, balance accounts, and catch and correct any errors. Other operations can, and do, exhibit errors. Because of the error tolerance that reconciliation provides, approximate computing may be completely appropriate in some banking operations as long as it preserves the integrity of the reconciliation process. As this example illustrates, approximate computing may be applicable in a broader range of application domains than many researchers and developers currently envision.

1.7 Contributions

This paper makes the following contributions:

- **Data Structure Construction Algorithms:** It presents approximate synchronization-free parallel data structure construction algorithms. These algorithms have no synchronization overhead and are free of synchronization-related anomalies such as excessive serialization and deadlock.

  Because these algorithms contain data races, they may drop inserted or appended elements and may produce data structures that do not satisfy all of the natural consistency properties. The produced data structures are, nevertheless, acceptable for approximate computations that can tolerate dropped elements.

- **Data Structure Building Blocks:** The data structure construction algorithms operate on general classes of data structures composed from tree and array building blocks. Examples of such data structures include hash tables, search trees, array lists, linked lists, and space-subdivision trees. We present C++ templates (with associated data structure construction algorithms) that can be instantiated and combined to implement such data structures.
- **Reasoning Approach:** It shows how to reason about the properties that the algorithms do preserve. This reasoning proceeds by induction to show that each individual write preserves the relevant consistency properties.
- **Semantic Data Races:** Data races are usually defined as unsynchronized accesses to shared data. Our position is that this concept is less useful than the concept of *semantic data races* — classifying parallel interactions according to the effect they have on specific data structure consistency properties (see Section 2.11). This concept makes it possible to distinguish productively between acceptable and unacceptable unsynchronized updates to shared data structures (based on the consistency properties that the updates do or do not preserve).
- **Evaluation Methodology:** Approximate data structures are acceptable only if they enable the client to operate acceptably. We therefore propose a methodology that evaluates approximate data structures in the context of a complete application. We evaluate their acceptability by comparing their effects with those of other approximate computing techniques (such as changing application-specific accuracy parameters) that increase the accuracy. If the accuracy decreases from the approximate data structures are negligible in comparison with the obtained accuracy increases, the approximate data structures are acceptable in the context of the application.
- **Experimental Results:** It presents experimental results that characterize the performance and accuracy of the unsynchronized tree construction algorithms in the context of the Barnes-Hut N-body simulation algorithm. These results show that the algorithm exhibits good parallel performance (on 16 cores, running 14 times faster than the sequential algorithm) and good accuracy (the effect on the body positions is four orders of magnitude less than increasing an accuracy parameter by 20%). The algorithm runs an order of magnitude faster than a synchronized version that uses standard tree locking and 5% to 10% faster than more sophisticated synchronized versions that use fine-grained locking or compare and swap instructions along with strategically placed retry operations.

2. Data Structures

We advocate the development of data structure building blocks with associated construction algorithms. These building blocks can then be composed to obtain a variety of data structures. We present tree and array building blocks as C++ templates.

2.1 Approximate Tree Data Structure

Our tree construction algorithm works with trees that contain internal nodes and external (leaf) nodes. Internal nodes reference other tree nodes; external nodes reference one or more inserted elements. To insert an element, the algorithm traverses the tree from the root to find (or, if necessary, create) the external node into which to insert the element. If the external node is full, it creates a new internal node to take the place of the external node. It then divides the elements in the external node into the new internal node and links the new internal node into the tree to take the place of the external node.

```
 1:template <typename E, class T,
 2:  class I, int N, class X, typename P>
 3:class internal : public T {
 4:  public: T *children[N];
 5:  void insert(E e, P p) {
 6:    int i;
 7:    T *t;
 8:    X *x;
 9:    i = index(e);
10:    t = children[i];
11:    if (t == NULL) {
12:      x = new (p) X((I *) this, i, e, p);
13:      children[i] = x;
14:    } else if (t->isInternal()) {
15:      ((I *) t)->insert(e, p);
16:    } else {
17:      x = (X *) t;
18:      if (!x->insert(e, p)) {
19:        I *c = new (p) I((I *) this, i, p);
20:        x->divide(c, p);
21:        children[i] = (T *) c;
22:        insert(e, p);
23:      }
24:    }
25:  }
26:  bool isInternal() { return true; }
27:  virtual int index(E e) = 0;
28:  ...
29:};
30:template <typename E, class T,
31:  class I, int N, class X, typename P>
32:class external : public T {
33:  public:
34:  bool isInternal() { return false; }
35:  virtual void divide(I *t, P p) = 0;
36:  virtual bool insert(E e, P p) = 0;
37   ...
38:};
39:template <typename E, class T, class I, int N,
40:  class X, typename P>
41:class tree {
42:  public: virtual bool isExternal() = 0;
43:};
```

Figure 1. Internal Tree Template and Tree Insert Algorithm

Figure 1 presents a C++ template that implements the insertion algorithm. In this template `E` is the type of the elements in the data structure, `T` is the type of the superclass of internal and external tree nodes, `I` is the internal node class, `N` is the number of references to child nodes in each internal node, `X` is the external (leaf) node class (instances of this class must implement the `divide` and `insert` methods as declared in the abstract `external` template, lines 35 to 36), and `P` is the type of a parameter that is threaded through the insertion algorithm. The program instantiates this template to obtain the internal node class of the data structure that the application uses.

The `insert` method (line 5) implements a standard tree insertion algorithm. It invokes the `index` method to determine the next child along the insertion path for the inserted element `e` (line 9). If the child is `NULL` (line 11), it creates a new external node that contains the inserted element `e` (line 14) and links the node into the tree (line 15). If the child is an internal node (line 14), it recursively descends the tree (line 15). Otherwise, the child is an external node.

```
 1:template <typename E, int M>
 2:class list {
 3:  public: int next; E elements[M];
 4:  virtual bool append(E e) {
 5:    int i = next;
 6:    if (M <= i) return false;
 7:    elements[i] = e;
 8:    next = i + 1;
 9:    return true;
10:  }
11:  ...
12:};
```

Figure 2. Array List Template and Append Algorithm

The algorithm attempts to insert the element `e` into the child external node (line 18). If the insertion fails (because the external node is full), it divides the elements in the external node into a new subtree (lines 19 to 20), links the new subtree into place (line 21), then retries the insertion (line 22).

Consider what may happen when multiple threads insert elements in parallel. There are multiple opportunities for the insertions to interfere in ways that drop elements. For example, multiple threads may encounter a `NULL` child, then start parallel computations to build new external nodes to hold the elements inserted in that location in the tree. As the computations finish and link the external nodes into the tree, they may overwrite references to previously linked external nodes (dropping all of the elements in those nodes). But even though these interactions may prevent the tree from containing all of the inserted elements, the tree still preserves key consistency properties required for clients to use it successfully (see Section 2.6.2).

We note that this algorithm may insert elements into the same external node concurrently. It is the responsibility of the external node insertion algorithm to operate acceptably in the face of such concurrent insertions. The approximate array list append algorithm (see Section 2.2) and approximate extensible array append algorithm (see Section 2.3) do so.

2.2 Approximate Array List Data Structure

Figure 2 presents a C++ template that implements an array list append algorithm. The template parameters are `E`, the type of the elements in the array list, and `M`, the number of elements in the list. The algorithm first checks to see if there is room in the array list for the appended element (line 6). If so, it inserts the element into the array (lines 7 to 8) and returns `true` (indicating that the append was successful). Otherwise it returns `false` (indicating that the array was full).

We note that there are multiple opportunities for parallel executions of the `append` operation to interfere in ways that drop elements. For example, parallel executions of lines 7 and 8 may overwrite references to concurrently appended elements.

2.3 Approximate Extensible Array List Data Structure

Figure 3 presents a C++ template that implements an append algorithm for an extensible array list. Researchers have previously developed a non-template version of this data structure and associated construction algorithm [19]. This data structure dynamically extends the array to accomodate more elements when the current array becomes full. The template paramters are `E`, the type of the elements in the array list, `M`, the number of elements in the initial list, and `P`, the type of a parameter threaded through the parallel computation. The append algorithm is similar to the append algorithm

```
template <typename E, typename P>
class helper {
  public:
    int next, last;
    E *elements;
  helper(int n, P p) {
    next = 0; last = n;
    elements = new (p) E[n];
  }
  helper(helper<E,P> *h, int n, P p) {
    next = h->next; last = n;
    elements = new (p) E[n];
    for (int i = 0; i < h->last; i++) {
      elements[i] = h->elements[i];
    }
  }
  ...
};

template <typename E, int M, typename P>
class extensible {
  public: helper<E,P> *h;
  extensible(P p) {
    h = new (p) helper<E,P>(M,p);
  }
  virtual bool append(E e, P p) {
    int i = h->next;
    int l = h->last;
    if (l <= i) {
      h = new (p) helper<E,P>(h, l * 2, p);
    }
    h->elements[i] = e;
    h->next = i + 1;
    return true;
  }
  ...
};
```

Figure 3. Extensible Array List Template and Append Algorithm

for fixed-sized array lists (see Section 2.2), except that it extends the array if the current array is full.

This data structure uses a separate `helper` class that contains the `next`, `last`, and `elements` variables that implement the extensible array. Instances of this `helper` class satisfy the invariant that `elements[i]` is always in bounds if `i<last`. An alternate strategy that eliminated the `helper` class by storing the `next`, `last`, and `elements` directly in the `extensible` instances would work for sequential executions but fail for parallel executions — data races could violate the invariant that `elements[i]` is always in bounds if `i<last`, in which case the `append` algorithm would perform out of bounds accesses.

This data structure therefore illustrates that, although a strength of the approximate data structures presented in this paper is the fact that they very closely resemble standard (and relatively simple) sequential implementations, they have been engineered to avoid pitfalls associated with closely related data structures that are correct in sequential contexts but crash when executed without synchronization in parallel contexts.

2.4 Array Lists As External Tree Nodes

The building blocks presented in Sections 2.1, 2.2, and 2.3 are designed to work together as components of hybrid data structures built from arbitrary combinations of trees and arrays.

```
 1:template <typename E, int M, class T,
 2:  class I, int N, class X, typename P>
 3:class externalList : public list<E,M>,
 4:  public external<E,T,I,N,X,P> {
 5:  public:
 6:  externalList(I *t, int i, E e, P p) {
 7:    append(e);
 8:  }
 9:  virtual bool insert(E e, P p) {
10:    return append(e);
11:  }
12:  virtual void divide(I *n, P p) {
13:    for (int i=0; i < list<E,M>::next; i++) {
14:      n->insert(list<E,M>::elements[i], p);
15:    }
16:  }
17:  ...
18:};
```

Figure 4. Array List As External Tree Node Template

Figure 4 presents a template for implementing external `tree` nodes as array list nodes. The `externalList` template inherits from both the `list` and `external` templates to obtain a template with the `list` functionality that can operate as an external tree node. The `insert` operation simply invokes the array list `append` operation. The `divide` operation iterates over the elements in the array list and inserts them into the internal node that is the root of the new subtree. In both cases the operations execute without interference on objects that are not (yet) accessible to other threads.

The template in Figure 4 implements the leaves of the tree with fixed-size array lists. Inheriting from the `extensible` template instead of the `list` template would produce a `externalExtensible` template that uses extensible arrays to implement the leaves of the tree.

2.5 Hash Tables

Because hash tables are typically implemented as a combination of (a single-level) tree and array data structures, it is possible to build an approximate hash table out of our tree and array building blocks. Figure 5 shows how to instantiate the `internal` and `externalExtensible` templates to obtain an approximate hash table. The hash array is implemented as a single-level tree. Each array element `children[i]` references an extensible array `bucket` object that holds the inserted elements that fall into that bucket. The `index` method uses a hash function to compute the bucket for each inserted element. The `insert` and `append` algorithms, which perform the hash table insertions, are encapsulated within the `internal` and `extensible` templates.

2.6 Barnes-Hut Algorithm

The Barnes-Hut N-body simulation algorithm works with a hierarchical space-subdivision tree. 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.

Figure 6 presents the class declarations for the objects that implement the Barnes-Hut space-subdivision tree. The `cell` class implements the internal nodes in the tree, the `leaf` class implements

```
class parent : public
  tree<int, parent, hash, 100, bucket, int> {
  ...
};
class bucket : public
  externalExtensible<int, parent, hash, 100,
                     bucket, 5, int> {
  ...
};
class hash : public
  internal<int, hb, hash, 100, bucket, int> {
  public:
    int index(int i) { ... }
    ...
};
```

Figure 5. Hash Table Classes

```
class body { ... };
class cell : public
  internal<body *, node, cell, 8, leaf, int> {
  int index(body *b) { ... }
  ...
};
class leaf : public
  externalList<body *, 10, node, cell, 8, leaf, int>
  { ... };
class node : public
  tree<body *, node, cell, 8, leaf, int>
  { ... };
```

Figure 6. Classes for Barnes-Hut Space Subdivision Tree

the external nodes in the tree. Both of these classes inherit from the `node` class. The `body` class implements the bodies in the simulation. It contains fields that store values such as the position, velocity, and acceleration of the body.

As Figure 6 indicates, the tree construction algorithms are inherited from the `internal` and `externalList` template classes. The only new method required to implement the tree construction algorithm is the `index(b)` method, which computes the index of the octant of the current cell into which the body `b` falls.

The parallel tree construction algorithm divides the bodies among the parallel threads. For each of its bodies `b`, each thread `p` invokes `c->insert(b,p)` (here `c` is the root of the space-subdivision tree) to insert the body into the tree. The computation allocates any required `cell` or `leaf` objects from independent allocation pools associated with thread `p`.

2.6.1 Natural Consistency Properties

Some natural consistency properties of the tree include:

- **Property L1 (Leaf In Bounds):** The `next` field in each `leaf` object that appears in the tree is at least `0` and at most `M` (the number of elements in the `elements` array).
- **Property L2 (Leaf Prefix):** For each `leaf` object `l` that appears in the tree and for all `i` at least `0` and less than `l->next`, `l->elements[i]` references a `body` object (and is therefore not `NULL`).
- **Property L3 (Leaf Postfix):** For each `leaf` object `l` that appears in the tree and for all `i` at least `l->next` and less than `M`, `l->elements[i]` is `NULL`;

- **Property T1 (Tree):** The `cell`, `leaf`, and `body` objects reachable from the root form a tree, where the edges of the tree consist of the references stored in the `children` fields of `cell` objects and the `elements` fields of `leaf` objects. Note that this property ensures that each `body` object appears at most once in the tree.
- **Property B1 (Body Soundness):** If a `body` object appears in the tree (i.e., is referenced by a `leaf` object reachable from the root), then it was inserted into the tree.
- **Property B2 (Body Completeness):** Each `body` object that was inserted into the tree actually appears in the tree.
- **Property O1 (Octant Inclusion):** Each `leaf` object that appears in the tree 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 `body` objects that each `leaf` object references fall within that region.

Together, we call these properties the *Natural Properties* of the tree. If the tree construction algorithm executes sequentially, it produces a tree that satisfies these properties.

2.6.2 Synchronization-Free 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 synchronization-free algorithm may not (and in practice does not) produce a tree that satisfies all of the above Natural Properties. Specifically, it may produce a tree that violates Property B2 (the algorithm may drop inserted bodies) and Property L3 (some parallel interactions may reset the `next` field back into the middle of the `elements` array).

We first consider what requirements the tree must satisfy so that the center of mass and force computation phases do not crash. It turns out that Properties T1, B1, O1, L1, and L2 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 properties.

We note that all of these properties are hard logical correctness properties of the type that standard data structure reasoning systems work with [27, 28]. 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 property:

- **Property A1 (Accuracy):** The tree contains sufficiently many `body` objects 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. 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. 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.

2.7 Memory Model

When we reason about parallel executions, we assume the executions take place on a parallel machine that implements individual reads and writes atomically. We also assume that writes become visible to other cores in the order in which they are performed. The computational platform on which we run our experiments (Intel Xeon E7340) implements a memory consistency model that satisfies these constraints.

Under the C++11 standard, if a parallel program performs conflicting memory accesses that are not ordered by synchronization operations or explicitly identified atomic read or write instructions, the meaning of the program is undefined. When using a compiler that implements this standard, we would identify the writes on lines `13` and `21`, Figure 1 and lines `7` and `8`, Figure 2 as atomic writes. We would also identify the reads on line `10`, Figure 1, line `5`, Figure 2, and and line `14`, Figure 4 (as well as any reads to `cell` or `leaf` fields perfomed in other methods executed during the parallel tree construction, the `index` method, for example) as atomic reads. The semantics of this program is defined under the C++11 standard. Moreover, when instructed to compile the program using an appropriate weak memory consistency model, an appropriately competent C++11 compiler should generate substantially the same instruction stream with substantially the same performance as our current implementation for the Intel Xeon E7340 platform.

2.8 Acceptability Argument

We next show that the tree construction algorithm preserves Properties T1, B1, O1, L1, and L2. We consider each property in turn and provide an argument that each tree assignment that may affect the property preserves the property. The argument proceeds by induction — we assume that the tree satisfies the properties before the assignment, then show that each assignment preserves the property.

- **Property L1 (Leaf In Bounds):** The `next` field is initialized to `0` in the `list` constructor. Line `8`, Figure 2 is the only other line that changes `next`. The check at line `6`, Figure 2 ensures that `next` is at most `M`. By the induction hypothesis, at line `7`, Figure 2 the index `i` is at least `0`, so `next` is set to a value that is at least `1`.
- **Property L2 (Leaf Prefix)**: We first show that the assignment `elements[i] = e` at line `7`, Figure 2 (the only assignment that changes `elements[i]`) always sets `elements[i]` to reference a non-`NULL` body `b`.

  There are two cases. The first case is the initial insertion of the body `b` into the tree via an invocation of `c->insert(b, p)`, where `c` is the root of the tree. In this case `b` is not `NULL` because the algorithm only inserts non-`NULL` bodies into the tree.

  The second case occurs via an execution of the `divide` method (line `12`, Figure 4), which inserts the bodies from a full `leaf` object into the subtree that will replace that `leaf` object in the tree. All of the inserted body objects `b` come from `elements[i]` (line `14`, Figure 4), where `i` is at least `0` and less than `next` (see the loop termination condition at line `13` of Figure 4). By the induction hypothesis all of these `body` objects are not `NULL`.

  We next show that the assignment `next = i + 1` (line `8`, Figure 2) preserves Property L2. By the induction hypothesis we must only show that `elements[i]` is non-`NULL` after the assignment executes. The preceding assignment at line `7`, Figure2) ensures that this is the case.
- **Property T1 (Tree):** The tree contains `cell`, `leaf`, and `body` objects. We show that there is at most one path to each such object from the root:
  - `cell` **Objects:** The algorithm creates a reference to a `cell` object only at line `21`, Figure 1. In this case the `cell` object is newly created at line `19`, Figure 1. The only path to this new `cell` object must therefore go through the parent object that contains the `children` array that references the new

cellobject. By the induction hypothesis there is at most one path to this parent object and therefore at most one path to the new cell object.

- leaf **Objects:** The algorithm creates a reference to a leaf object only at line 13, Figure 1. In this case the leaf object is newly created at line 12, Figure 1. The only path to this new leaf object must therefore go through the parent object that contains the children array that references the new leafobject. By the induction hypothesis there is at most one path to this parent object and therefore at most one path to the new leaf object.
- body **Objects:** For each body object b, the algorithm invokes c->insert(b,p), where c is the root of the tree, exactly once. During this call it creates the first reference to the body, either inside the leaf constructor (line 12, Figure 1) or at line 7, Figure 2 via the call to x->insert(e,p) (line 18, Figure 1).

  The algorithm may subsequently create a new reference to b when the leaf object x that references b becomes full and the algorithm invokes x->divide(c,p) to create a new subtree c that holds the referenced body objects. When the call to divide completes, the subtree rooted at c (which is not yet reachable) may contain a reference to b. This reference becomes reachable when the assignment at line 21 of Figure 1 creates a path to the new cell object created at line 19 of Figure 1. Because this assignment overwrites the old reference from children[i] to x through which b was reachable in the original tree, it preserves the invariant that there is at most one path to b.

- **Property B1 (Body Soundness):** The tree construction algorithm starts with an empty tree, then invokes the insert procedure for each body in the simulation. The invoked code only creates cell and leaf objects, not bodies. All bodies that appear in the tree are therefore inserted by the tree construction algorithm.
- **Property O1 (Octant Inclusion):** The initial insertion of a body b into the tree traverses a path to an external node, then inserts b into that node. At each step the algorithm invokes the index method (line 9, Figure 1 to determine the appropriate octant at that level into which to insert the body b.

  The initial path may be extended when the leaf object that references b becomes full and the algorithm inserts another cell object c into the path. At this step, the algorithm again invokes the index method to determine the correct octant in c into which to insert the body b.

2.9 Key Concepts

Our algorithms have the advantage that they are essentially clean, easily-understandable sequential algorithms that execute acceptably when run in parallel without synchronization. But as the above acceptability argument illustrates, the acceptability of the algorithm relies on two key structuring techniques:

- **Link At End:** Some data structure updates link a new leaf or subtree into the tree. Because our algorithms link the leaf or subtree into place only after they fully initialize and construct the new leaf or subtree, they ensure that parallel threads only encounter fully constructed leaves or subtrees that they can access without crashing.
- **Local Read, Check, and Index:** One natural way to code the append operation (Figure 2) would read next (which references the next available array element) three times: first to to check if the array is full (line 6, Figure 2), then again to determine where to insert the item (line 7, Figure 2), then again to increment next (line 8, Figure 2). If coded this way, the resulting data races could cause out of bounds accesses that crash the computation.

  The append operation in Figure 2 avoids such data races by reading next into a local variable at the start of the operation, then using this local variable for all checks, indexing, and update operations. This technique eliminates out of bounds accesses even in the presence of data races associated with unsynchronized parallel executions.

2.10 Final Check and Data Structure Repair

It is possible to view the tree insertion and array append algorithms as composed of fine-grained updates, each of which first performs a check to determine which action to perform, next (in general) executes a sequence of instructions that construct a new part of the data structure, then executes one or more write instructions to commit the update. The unsynchronized algorithm drops bodies because writes from parallel threads may change the checked state so that the check would no longer produce the same result if executed at the time when the writes that commit the update execute. We call the time between the check and the commit the *window of vulnerability* because it is during this time that state changes may interfere with the atomicity of the check and the update.

We use a technique, *final check*, to shrink (but not eliminate) the window of vulnerability. Just before the commit, the final check performs part or all of the check again to determine if it should still perform the update. If not, the algorithm discards the action and retries the insert or append. 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. Figure 7 presents a tree insert algorithm that contains final checks. Figure 8 presents an array append algorithm with final checks. If a final check fails, both algorithms immediately retry.

The final check at line 6, Figure 8 may infinite loop if Property L3 (Leaf Postfix) is violated (so that elements[next] is non-NULL). A data structure repair algorithm (lines 12 to 19, Figure 8) eliminates the infinite loop by repairing the data structure to restore Property L3.

2.11 Semantic Data Races

Of course, even though the algorithm preserves Properties L1, L2, T1, B1, and O1, it still contains data races (because it performs conflicting parallel accesses to shared data without synchronization). But instead of focusing on a low-level implementation detail that may have little or nothing to do with the acceptability, we instead focus on the semantic consequences of interactions between parallel threads.

We make this concept precise as follows. We say that there is a *semantic 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 2.6.1.

Data races are traditionally defined as unsynchronized conflicting accesses to shared data. Our semantic approach to defining data races, in contrast, makes it possible to distinguish different kinds of data races based on the properties that they do and do not violate. Our unsynchronized parallel tree construction algorithms, for example, may violate Properties L3 and B2, but do not violate the other properties. Because the clients of the tree construction algorithm can operate acceptably with trees that violate Properties L3 and B2, these data races are acceptable. Other data races are unacceptable because they may cause the data structure to violate crit-

```
 1:void insert(E e, P p) {
 2:  int i;
 3:  T *t;
 4:  X *x;
 5:  i = index(e);
 6:  t = children[i];
 7:  if (t == NULL) {
 8:    x = new (p) X((I *) this, i, e, p);
 9:    // final check
10:    if (children[i] == NULL) {
11:      children[i] = x;
12:    } else {
13:      insert(e, p);
14:    }
15:  } else if (t->isInternal()) {
16:    ((I *) t)->insert(e, p);
17:  } else {
18:    x = (X *) t;
19:    if (!x->insert(e, p)) {
20:      I *c = new (p) I((I *) this, i, p);
21:      x->divide(c, p);
22:      // final check
23:      if (children[i] == t) {
24:        children[i] = (T *) c;
25:      }
26:      insert(e, p);
27:    }
28:  }
29:}
```

Figure 7. Tree Insert Algorithm With Final Check

```
 1:virtual bool append(E e) {
 2:  while (true) {
 3:    int i = next;
 4:    if (M <= i) return false;
 5:    // final check
 6:    if (elements[i] == NULL) {
 7:      elements[i] = e;
 8:      next = i + 1;
 9:      return true;
10:    } else {
11:      // data structure repair
12:      while (true) {
13:        if ((elements[i] == NULL) ||
14:            (M <= i)) {
15:          next = i;
16:          break;
17:        }
18:        i++;
19:      }
20:    }
21:  }
22:}
```

Figure 8. Append Algorithm With Final Check And Data Structure Repair

ical properties that must hold for clients (and indeed, for the tree construction algorithm itself) to execute successfully.

This semantic approach to data races therefore provides the conceptual framework we need to understand why our unsynchronized parallel tree construction algorithms produce an acceptable result. Specifically, these algorithms are acceptable because their data races preserve critical properties that must be true for the computation and its clients to execute successfully.

2.12 Atomic Updates

We discuss data races further by identifying specific updates in the Final Check version. Each update 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 write that the algorithm performs if the check succeeds). If each update executes atomically, the algorithm does not drop bodies and satisfies all of the Natural Properties from Section 2.6.1.

- **New Leaf:** The check for a `NULL` child reference (line `10`, Figure 7) in combination with the write that links the new leaf into place (line `11`, Figure 7).
- **New Cell:** The check that the reference to the `leaf` object `t` in `children[i]` (line `23`, Figure 7) has not changed since the test at line `15`, Figure 7 in combination with the write that links the new `cell` object `c` into the tree.
- **Insert Body:** The check that the slot at `elements[i]` is `NULL` and therefore available (line `6`, Figure 8) in combination with the write that links the element `e` into the array list (line `7`, Figure 8).

We have developed two versions of the algorithm that execute these operations atomically (Section 3 presents results from these versions): one uses mutual exclusion locks, the other uses compare and swap instructions. Even with these synchronization operations, these versions still contain data races. But because they preserve the Natural Properties, they do not contain semantic data races (although the analysis required to verify that this is true is non-trivial). We therefore view semantic data races as a more productive and appropriate concept for reasoning about the potential interactions in this algorithm than the traditional concept of data races.

3. Experimental Results

We present results from a parallel Barnes-Hut computation that uses the algorithms described in Sections 2.1, 2.2, and 2.10 to build its space-subdivision tree. We implement the computation in C++ using the `pthreads` threads package. At each step of the simulation 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.

For comparison purposes, we implemented several different versions of the algorithm:

- **TL (Tree Locking):** This version locks each node in the tree before it accesses the node. As it descends the tree, it releases the lock on the parent node and acquires the lock on the child node. Tree locking is a standard way to synchronize tree updates.
- **UL (Update Locking:)** This version uses mutual exclusion locks to make the updates identified in Section 2.12 execute atomically. It satisfies all of the Natural Properties and does not drop bodies.
- **HA (Hyperaccurate):** The Update Locking 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 — the smaller the `tol` parameter, the deeper the phase goes into the tree before it applies the center of mass approximation. We use this version to evaluate the accuracy consequences of dropping bodies from the space-subdivision tree.

- **CAS (Compare And Swap)**: This version uses compare and swap instructions to make the updates identified in Section 2.12 execute atomically. It satisfies all of the Natural Properties and does not drop bodies.
- **FP (First Parallel)**: The synchronization-free approximate versions in Figures 1 and 2.
- **FC (Final Check)**: The synchronization-free approximate versions in Figures 7 and 8 that use final checks to shrink the window of vulnerability.

Note that the Tree Locking, Update Locking, and Compare and Swap versions all compute the same result. We run all versions on a 16 core 2.4 GHz Intel Xeon E7340 machine with 16 GB of RAM running Debian Linux kernel version 2.6.27. We report results from executions that simulate 100 steps of a system with 256K bodies. We initialize the positions and velocities of the 256K bodies to psuedorandom numbers.

3.1 Accuracy

We define the distance metric Δ_Y^X between two versions X and Y of the algorithm as follows:

$$\Delta_Y^X = \sum_{0 \le i < N} d(\mathtt{b}_i^X, \mathtt{b}_i^Y)$$

Here $\mathtt{b}_i^X$ is the final position of the ith body at the end of the simulation that uses version X of the tree construction algorithm (and similarly for $\mathtt{b}_i^Y$), $d(\mathtt{b}_i^X, \mathtt{b}_i^Y)$ is the Euclidean distance between the final positions of corresponding bodies in the two simulations, and N is the number of bodies in the simulation.

We evaluate the accuracy of a given version X by comparing its final body positions with those computed by the Hyperaccurate (HA) version. The minimum Δ_X^{HA}, over all versions $X \neq$ HA, all executions, and all number of cores, is 3041.48.

We next use the distance metric Δ_Y^X to evaluate the inaccuracy that the use of approximate data structures introduces. Specifically, we compute $\Delta_{FP}^{HA} - \Delta_{UL}^{HA}$ and $\Delta_{FC}^{HA} - \Delta_{UL}^{HA}$ as the *additional inaccuracy metric* for the First Parallel and Final Check versions, respectively. These differences quantify the additional inaccuracy introduced by the use of approximate data structure construction algorithms in these versions. We compare these differences to Δ_{UL}^{HA}.

Table 1 presents the maximum (over the eight runs) additional inaccuracy metric for the First Parallel and Final Check versions as a function of the number of cores executing the computation. These numbers show that the accuracy loss introduced by the use of approximate data structures is four orders of magnitude smaller than the accuracy gain obtained by increasing the accuracy of the center of mass approximation (0.26 in comparison with 3041.48). This fact supports the acceptability of the presented approximate data structure construction algorithms. The rationale is that, in comparison with the Hyperaccurate version, all other versions (including the approximate synchronization-free versions) compute results with essentially identical accuracy.

3.2 Final Check Effectiveness

Table 2 reports the maximum (over all eight executions) of the sum (over all 100 simulation steps) of the number of bodies that the First Parallel and Final Check versions drop. Note that there are 256K*100 inserted bodies in total. These numbers show that the number of dropped bodies is very small — even running on 16 cores with no final check to reduce the number of dropped bodies, in our eight runs the First Parallel algorithm drops at most 730 of the bodies it inserts. Other versions drop significantly fewer bodies. These numbers also show that the final check is effective in reducing the number of dropped bodies by a factor of 3 to 6 depending on the number of cores executing the computation.

	Number of Cores				
Version	1	2	4	8	16
FP	0.00	0.05	0.57	0.26	-0.63
FC	0.00	0.02	0.03	0.09	-0.11

Table 1. Maximum additional inaccuracy metric for First Parallel ($\Delta_{FP}^{HA} - \Delta_{UL}^{HA}$) and Final Check ($\Delta_{FC}^{HA} - \Delta_{UL}^{HA}$) versions. Compare with the minimum Δ_{UL}^{HA} = 3041.48.

	Number of Cores				
Version	1	2	4	8	16
FP	0	62	80	202	730
FC	0	13	24	34	159

Table 2. Number of dropped bodies (out of 256K * 100 total inserted bodies) for First Parallel (FP) and Final Check (FC) versions.

	Number of Cores				
Version	1	2	4	8	16
FC	1.00	1.88	3.41	7.32	14.15
FP	1.00	1.87	3.38	7.28	14.02
CAS	0.95	1.75	3.23	6.86	13.21
UL	0.90	1.67	3.02	6.51	12.58
TL	0.83	0.87	0.58	0.40	0.39

Table 3. Speedup Numbers for Barnes-Hut Tree Construction

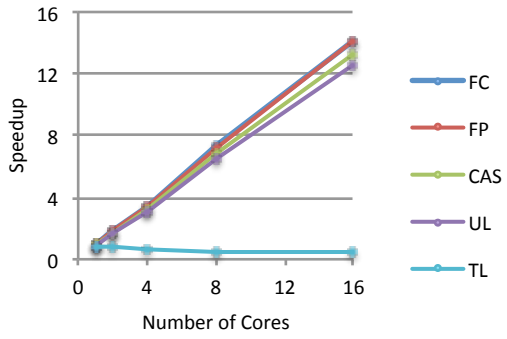

Figure 9. Speedup Curves for Barnes-Hut Tree Construction

3.3 Accuracy For Long Simulations

To better understand the effect of the First Parallel and Final Check versions on the accuracy of the simulation as the number of steps increases, we computed Δ_H^S, Δ_S^{FP}, and Δ_S^{FC} after each of the 100 steps s of the simulation. Plotting these points reveals that they form curves characterized by the following equations:

$$\Delta_{UL}^{HA}(s) = 0.3s^2 + 1.4s$$

$$\Delta_{FP}^{UL}(s) = 0.002s^2 + 0.04s$$

$$\Delta_{FC}^{UL}(s) = 0.0004s^2 + 0.008s$$

These equations show that the difference between the Hyperaccurate and Update Locking versions grows substantially faster than the difference the between Update Locking and First Parallel/Final Check versions. Our interpretation of this fact is that the First Parallel/Final Check versions will remain as acceptably accurate as the Update Locking version even for long simulations with many steps.

3.4 Performance

Table 3 presents speedup numbers for the different versions; Figure 9 presents the corresponding speedup curves. The X axis plots the number of threads executing the computation. The Y axis plots the mean speedup (the running time of the parallel version divided by the running time of the sequential version, which executes without parallelization overhead) over eight executions of each version (each of which starts with different body positions and velocities).

The Tree Locking version exhibits extremely poor parallel performance — 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 remaining versions scale well, with the First Parallel and Final Check versions running between 5% to 10% faster than the versions that contain synchronization. We attribute the performance difference to the synchroinzation overhead in these versions.

4. Related Work

To date, approximate computing techniqes such as task skipping [20, 21], loop perforation [9, 15, 16, 23, 29], and approximate function memoization [4] have focused on the computation (as opposed to the data structures that the computation may manipulate). This research, in contrast, focuses on data structure construction (although the effect may be conceptually similar in that our approximate data structures may drop inserted or appended elements).

The design and implementation of sophisticated wait-free data structures has been an active research area for some years [7]. Our research differs in several ways. First, our synchronization-free algorithms we use only reads and writes (wait-free data structures typically rely on complex synchronization primitives such as compare and swap or, more generally, transactional memory [8]). Second, we do not aspire to provide a data structure that satisfies all of the standard correctness properties. Third, our algorithms are essentially clean, easily-understandable sequential algorithms that have been carefully designed to execute in parallel without crashing. Researchers have also designed a similarly simple unsynchronized accumlator and extensible array with a final check [19]. Wait-free implementations are typically significantly more complex than their sequential counterparts.

In recent years researchers have acknowledged the need to relax traditional data structure correctness guarantees, typically by relaxing the order in which operations such as queue insertions and removals can complete [10, 22]. Our research differs in that it delivers approximate data structures (which may drop inserted or appended elements) as opposed to relaxed data structures (which may relax the order in which operations execute but do not propose to drop elements).

Synchronization has been identified as necessary to preserve the correctness of parallel space-subdivision tree construction algorithms [24]. The desire to eliminate synchronization overhead motivated 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 drop inserted bodies. Our results show that these trees are still acceptably accurate.

The Cilkchess parallel chess program uses concurrently accessed transposition tables [6]. 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) [6]. Like Cilkchess, our parallel tree insertion algorithm improves performance by purposefully eliminating synchronization and therefore contains acceptable data races.

The QuickStep compiler automatically generates parallel code for a wide range of loops in both standard imperative and object-oriented programs [13, 14]. Unlike other parallelizing compilers, but like the approximate data structure construction algorithms presented in this paper, QuickStep is designed to generate parallel programs that may contain data races that acceptably change the output of the program.

The race-and-repair project has developed a synchronization-free parallel hash table insertion algorithm [26]. 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 (and a previous technical report [18]) presents an algorithm that works with clients that simply use the tree as produced 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, 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 approximate semantics of the synchronization-free algorithm has acceptable accuracy consequences for this computation.

Chaotic relaxation runs iterative solvers without synchronization [2, 5, 25]. 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.

5. 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 presenting general building blocks (with associated encapsulated approximate data structure construction algorithms) that can be composed to obtain approximate data structures built from trees, fixed-size arrays, and extensible arrays. We use these building blocks to obtain an approximate 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] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. FSE, 2011.

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

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

[7] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. May 1993.

[9] 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.

[10] Christoph M. Kirsch and Hannes Payer. Incorrect systems: it's not the problem, it's the solution. In *DAC*, 2012.

[11] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin C. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *OOPSLA*, 2012.

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

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

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

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

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

[17] Sasa Misailovic, Stelios Sidiroglou, and Martin Rinard. Dancing with uncertainty. RACES Workshop, 2012.

[18] Martin Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, February 2012.

[19] Martin Rinard. Unsynchronized techniques for approximate parallel computing. RACES Workshop, 2012.

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

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

[22] Nir Shavit. Data structures in the multicore age. *CACM*, 54(3), 2011.

[23] 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.

[24] 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.

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

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

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

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

[29] 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.