Parallel Synchronization-Free Approximate Data Structure Construction

Martin Rinard, MIT CSAIL

Abstract

We present approximate data structures with construction algorithms that execute without synchronization. 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 details of why they execute successfully.

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. The resulting approximate data structure construction algorithm eliminates synchronization overhead and anomalies such as excessive serialization and deadlock. 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, many video and image processing computations, modern internet search and information retrieval, and many scientific computations) exhibit the flexibility to produce a range of acceptably accurate outputs. Researchers have exploited this flexibility to develop techniques that profitably trade off accuracy in return for increased performance, reduced power consumption, or the ability to adapt to changing conditions in the underlying computational platform [4, 9, 13–16, 21, 22, 24, 28].

1.1 Approximate Data Structure Building Blocks

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 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.

Approximation: 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.

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 reusable data structure building blocks 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.

1.2 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:

- **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.

1.3 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 evaluate the performance and accuracy consequences of using our building blocks for the space-subdivision tree in the Barnes-Hut N-body computation [1, 25]. 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.

1.4 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.

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 [21], loop perforation [9, 16, 24], early phase termination [22], infinite loop exit [3, 11], reduction sampling [28], 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. Approximations that occur infrequently (as is the case for the approximate data structures in this paper) and preserve the integrity of the reconciliation process can therefore be completely appropriate in some banking operations. As this example illustrates, approximate computing may be applicable in a broader range of application domains than many researchers and developers currently envision.

2. Data Structures

Approximate Tree: 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 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. Figure 1 presents a C++ template that implements the insertion algorithm. Our Barnes-Hut implementation uses instances of the `internal` template to represent internal tree nodes and instances of the `list` template to represent external nodes (each of which may reference up to `M` bodies).

There are many ways for parallel 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).

Approximate Array List: Figure 2 presents a C++ template that implements an array list append algorithm. 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` in Figure 2 may overwrite references to concurrently appended elements.

Final Check: We 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. We call the time between the check and the commit the *window of vulnerability* — during this time, 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. Redoing the null check at line 11 of Figure 1 immediately before linking in the new node at line 13 (and retrying the insert if another thread has already linked another node) can reduce the number of dropped nodes [18].

```
 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:};
```

Figure 1. Internal Tree Template and Insert Algorithm

```
 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

Key Concepts: A strength of our approximate data structures is that they are essentially standard sequential data structures that 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. Two key structuring principles underly this engineering:

- **Link At End:** Some data structure updates link a new leaf or subtree into the tree. Because our algorithms link the new leaf or subtree into place only after they fully initialize and construct it, 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) reads `next` (which references the next available array element) three times: first to check if the array is full (line 6, Figure 2), next 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 can 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 (line 5, Figure 2), 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.

Applying the above structuring principles delivers acceptable parallel implementations of other standard building block data structures such as extensible arrays [18]. The resulting templates are designed to work together as components of hybrid data structures (such as hash tables [18] and space subdivision trees [18]).

2.1 Acceptability Argument

Even though our approximate data structures may drop inserted elements, they never crash and correctly preserve key data structure consistency properties required for clients to correctly traverse and retrieve elements from the data structure. It is possible to formalize these consistency properties and show that the data structures satisfy them in all parallel executions [18]. We consider each property in turn and provide an argument that each update that may affect the property preserves the property. Conceptually, the argument proceeds by induction — we assume that the tree constructed to date satisfies all the consistency properties, then argue that each tree update preserves these properties.

In addition to these hard logical consistency properties, the tree must also contain enough body objects so that the force computation produces a sufficiently accurate result. We do not attempt to establish 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 by observing executions of the program. Specifically, we compare the results that the program produces when it uses our unsynchronized data structures with results that we know to be accurate, then use this comparison to evaluate the end-to-end acceptability of the data structures in the context of the Barnes-Hut computation.

2.2 Memory Consistency Models

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 conflicting memory accesses are not ordered by synchronization operations or explicitly identified atomic read or write instructions, 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 and line `5`, Figure 2 (as well as any reads to fields in shared objects perfomed in other methods executed during the parallel tree construction, the `index` method, for example) as atomic reads. 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.

3. Experimental Results

We present results from a parallel Barnes-Hut computation that uses the algorithms described in Section 2 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 and T is the number of parallel threads.

3.1 Barnes-Hut Space-Subdivision Tree

Barnes-Hut 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).

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. We have instantiated the data structures in Figures 1 and 2 to obtain an approximate Barnes-Hut space-subdivision tree [18]. For comparison purposes, we implemented several versions:

- **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 and acquires the lock on the child. Tree locking is a standard way to synchronize tree updates.
- **CAS (Compare And Swap)**: This version uses compare and swap (CAS) instructions to make individual data structure updates execute atomically. In Figure 1, the CAS at line 13 checks that element `children[i]` is still `NULL`. The CAS at line 21 checks that `children[i]` is still equal to `t`. In Figure 2 the CAS at line 7 checks that `elements[i]` is still `NULL`. If a CAS fails, the version retries the insertion.
  Although this version still contains data races, it does not drop bodies and produces the same result as the TL version (the analysis required to verify that this is the case is nontrivial).
- **UL (Update Locking:)** This version uses fine-grained locks to make updates execute atomically. The granularity and conditions checked are the same as for the CAS version. Like the CAS version, this version contains data races but produces the same result as the TL version.
- **HA (Hyperaccurate):** The Update Locking version running with a smaller `tol` parameter (the original `tol` parameter divided by 1.25). The `tol` parameter controls the center-of-mass approximation in the force computation phase — the smaller the `tol` parameter, the deeper the phase goes 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.
- **FP (First Parallel)**: The synchronization-free approximate versions in Figures 1 and 2.
- **FC (Final Check)**: The synchronization-free approximate versions augmented to use final checks.

We run all versions on a 16 core 2.4 GHz Intel Xeon E7340 with 16 GB of RAM and Debian version 2.6.27. We compile all versions with g++ -O4. We simulate 100 steps of a system with 256K bodies. We initialize the positions and velocities of the bodies to psuedorandom numbers and the masses of the bodies uniformly to 1.

3.2 Accuracy

We define the distance Δ_Y^X between two versions X and Y as:

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

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.

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 approximate data structure construction algorithms — in comparison with the Hyperaccurate version, all other versions (including the approximate versions) compute results with essentially identical accuracy.

Final Check: 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

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:

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

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

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

The difference between the Hyperaccurate and Update Locking versions grows substantially faster than the difference between the Update Locking and First Parallel/Final Check versions. The First Parallel/Final Check versions will therefore remain as acceptably accurate as the Update Locking version even for long simulations with many steps.

3.3 Performance

Table 3 presents speedup numbers for the different versions. All of the numbers are calculated relative to the sequential version, which executes without parallelization overhead. The TL version exhibits poor parallel performance (the performance decreases as the number of cores increases), which we attribute to a combination of synchronization overhead and bottlenecks associated with locking the top cells in the tree. The remaining versions scale — the FP and FC versions run between 5% to 10% faster than the synchronized versions. We attribute the performance difference to the synchronization overhead. Because the FP and FC versions have no synchronization, their base sequential performance is essentially identical to the sequential version.

4. Related Work

Parallel data structures relax the order in which operations such as queue insertions and removals can complete [10, 23]. 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 drop elements).

Wait-free data structures has been an active research area for some years [7]. Our research differs in that 1) our synchronization-free algorithms 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]); 2) we do not aspire to provide a data structure that satisfies all of the standard correctness properties; and 3) our algorithms are essentially clean, easily-understandable sequential algorithms that have been engineered to execute in parallel without crashing. Wait-free implementations are typically significantly more complex than their sequential counterparts.

The QuickStep parallelizing compiler generates approximate code with acceptable data races [13, 14]; the Dubstep compiler removes synchronization in parallelized code (creating acceptable data races) [17]. [20] presents a simple unsynchronized accumulator and extensible array with a final check; [19] presents unsynchronized parallel space-subdivision tree algorithms for the Barnes-Hut simulation [19].

Chaotic relaxation runs iterative solvers without barrier synchronization after each solver iteration [2, 5, 26]. Convergence theorems prove that the computation will still converge even in the presence of data races. The performance impact depends on the specific problem at hand — some converge faster with chaotic relaxation, others more slowly. Chaotic solvers typically operate on arrays instead of linked data structures. Because chaotic solvers come with a convergence test, they (in principle) impact only the performance and not the accuracy. As used in the Barnes-Hut computation, approximate data structures do not come with a convergence test or any other run-time mechanism that checks the accuracy. Approximate data structures may therefore impact both the performance and the accuracy. For this reason we introduce an evaluation metric that evaluates the accuracy impact of approximate data structures in comparison with the accuracy impact of other approximation mechanisms, in this case the center of mass approximation that is at the heart of the Barnes-Hut algorithm.

The race-and-repair project has developed an unsynchronized parallel hash table insertion algorithm [27]. 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 [19]) 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.

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.

5. Conclusion

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 present general approximate tree and array building blocks (with associated approximate data structure construction algorithms) that can be composed to obtain approximate data structures. We use these building blocks to obtain an approximate synchronization-free parallel space-subdivision tree construction algorithm that 1) contains data races but 2) nevertheless 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. Parallel synchronization-free approximate data structure construction (full version). `http://people.csail.mit.edu/rinard/paper/hotpar13.full.pdf`.

[19] 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.

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

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

[24] 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.

[25] 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.

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

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

[28] 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.