Phase-Concurrent Hash Tables for Determinism

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

ABSTRACT

We present a deterministic *phase-concurrent* hash table in which operations of the same type are allowed to proceed concurrently, but operations of different types are not. Phase-concurrency guarantees that all concurrent operations commute, giving a deterministic hash table state, guaranteeing that the state of the table at any quiescent point is independent of the ordering of operations. Furthermore, by restricting our hash table to be phase-concurrent, we show that we can support operations more efficiently than previous concurrent hash tables. Our hash table is based on linear probing, and relies on history-independence for determinism.

We experimentally compare our hash table on a modern 40-core machine to the best existing concurrent hash tables that we are aware of (hopscotch hashing and chained hashing) and show that we are 1.3–4.1 times faster on random integer keys when operations are restricted to be phase-concurrent. We also show that the cost of insertions and deletions for our deterministic hash table is only slightly more expensive than for a non-deterministic version that we implemented. Compared to standard sequential linear probing, we get up to 52 times speedup on 40 cores with dual hyper-threading. Furthermore, on 40 cores insertions are only about $1.3\times$ slower than random writes (scatter). We describe several applications which have deterministic solutions using our phase-concurrent hash table, and present experiments showing that using our phase-concurrent deterministic hash table is only slightly slower than using our non-deterministic one and faster than using previous concurrent hash tables, so the cost of determinism is small.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel Programming*

Keywords: Hash Table, Determinism, Applications

1. INTRODUCTION

Many researchers have argued the importance of deterministic results in developing and debugging parallel programs (see e.g. [32, 6, 8, 25, 3]). Two types of determinism are distinguished—external determinism and internal determinism. External determinism requires that the program produce the same output when given the same input, while internal determinism also requires certain intermediate steps of the program to be deterministic (up to some level of abstraction) [3]. In the context of concurrent access, a data structure is internally deterministic if even when operations are applied concurrently the final observable state depends uniquely on the set of operations applied, but not on their order. This property is equivalent to saying the operations commute with respect to the final observable state of the structure [36, 32]. Deterministic concurrent data structures are important for developing internally deterministic parallel algorithms—they allow for the structure to be updated concurrently while generating a deterministic result independent of the timing or scheduling of threads. Blelloch et al. show that internally deterministic algorithms using nested parallelism and commutative operations on shared state can be efficient, and their algorithms make heavy use concurrent data structures [3].

However, for certain data structures, the operations naturally do not commute. For example, in a hash table mixing insertions and deletions in time would inherently depend on ordering since inserting and deleting the same element do not commute, but insertions commute with each other and deletions commute with each other, independently of value. The same is true for searching mixed with either insertion or deletion. For a data structure in which certain operations commute but others do not, it is useful to group the operations into phases such that the concurrent operations within a phase commute. We define a data structure to be ***phase-concurrent*** if subsets of operations can proceed (safely) concurrently. If the operations within a phase also commute, then the data structure is deterministic. Note that phase-concurrency can have other uses besides determinism, such as giving more efficient data structures. It is the programmer's responsibility to separate concurrent operations into phases, with synchronization in between, which for most nested parallel programs is easy and natural to do.

In this paper, we focus on the hash table data structure. We describe a deterministic phase-concurrent hash table and prove its correctness. Our data structure builds upon a sequential history-independent hash table [4] and allows concurrent insertions, concurrent deletions, concurrent searches, and reporting the contents. It does not allow different types of operations to be mixed in time, because commutativity (and hence determinism) would be violated in general. We show that using one type of operation at a time is still very useful for many applications. The hash table uses open addressing with a prioritized variant of linear probing and guarantees that in a quiescent state (when there are no operations ongoing) the exact content of the array is independent of the ordering of previous updates. This allows, for example, quickly returning the contents of the hash table in a deterministic order simply by packing out the empty cells, which is useful in many applications. Returning the contents could be done deterministically by sorting,

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
Copyright 2014 ACM 978-1-4503-2821-0/14/06 ...$15.00
http://dx.doi.org/10.1145/2612669.2612687 .

but this is more expensive. Our hash table can store key-value pairs either directly or via a pointer.

We present timings for insertions, deletions, finds, and returning the contents into an array on a 40-core machine. We compare these timings with the timings of several other implementations of concurrent and phase-concurrent hash tables, including the fastest concurrent open addressing [15] and closed addressing [20] hash tables that we could find, and two of our non-deterministic phase-concurrent implementations (based on linear probing and cuckoo hashing). We also compare the implementations to standard sequential linear probing, and to the sequential history-independent hash table. Our experiments show that our deterministic hash table significantly outperforms the existing concurrent (non-deterministic) versions on updates by a factor of 1.3–4.1. Furthermore, it gets up to a $52\times$ speedup over the (standard) non-deterministic sequential version on 40 cores with two-way hyper-threading. We compare insertions to simply writing into an array at random locations (a scatter). On 40 cores, and for a load factor of $1/3$, insertions into our table is only about $1.3\times$ the cost of random writes. This is because most insertions only involve a single cache miss, as does a random write, and that is the dominant cost.

Such a deterministic hash table is useful in many applications. For example, Delaunay refinement iteratively adds triangles to a triangulation until all triangles satisfy some criteria (see Section 5). "Bad triangles" which do not satisfy the criteria are broken up into smaller triangles, possibly creating new bad triangles. The result of Delaunay refinement depends on the order in which bad triangles are added. Blelloch et al. show that using a technique called *deterministic reservations* [3], triangles can be added in parallel in a deterministic order on each iteration. However, for the algorithm to be deterministic, the list of new bad triangles returned in each iteration must also be deterministic. Since each bad triangle does not know how many new bad triangles will be created, the most natural and efficient way to accomplish this is to add the bad triangles to a deterministic hash table and return the contents of the table at the end of each iteration. Without a hash table, one would either have to first mark the bad triangles and then look through all the triangles identifying the bad ones, which is inefficient, or use a fetch-and-add to a vector storing bad triangles (non-deterministic), leading to high contention, or possibly use a lock-free queue (non-deterministic), again leading to high contention. By using a deterministic hash table in conjunction with deterministic reservations, the order of the bad triangles is deterministic, giving a deterministic implementation of parallel Delaunay refinement.

We present six applications which use hash tables in a phase-concurrent manner, and show that our deterministic phase-concurrent hash table can be used both for efficiency and for determinism. For four of these applications—remove duplicates, Delaunay refinement, suffix trees and edge contraction—we believe the most natural and/or efficient way to write an implementation is to use a hash table. We show that for these applications, using our deterministic hash table is only slightly slower than using a non-deterministic one based on linear probing, and is faster than using cuckoo hashing or chained hashing (which are also non-deterministic). For two other applications—breadth-first search and spanning tree—we implement simpler implementations using hash tables, compared to array-based versions directly addressing memory. We show that the implementations using hash tables are not much slower than the array-based implementations, and again using our deterministic hash table is only slightly slower than using our non-deterministic linear probing hash table and faster than using the other hash tables.

Contributions. The contributions of this paper are as follows. First, we define the notion of phase-concurrency. Second, we show that phase-concurrency can be applied to hash tables to obtain both determinism and efficiency. We prove correctness and termination of our deterministic phase-concurrent hash table. Third, we present a comprehensive experimental evaluation of our hash tables with the fastest existing parallel hash tables. We compare our deterministic and non-deterministic phase-concurrent linear probing hash tables, our phase-concurrent implementation of cuckoo hashing, hopscotch hashing, which is the fastest existing concurrent open addressing hash table as far as we know, and an optimized implementation of concurrent chained hashing. Finally, we describe several applications of our deterministic hash table, and present experimental results comparing the running times of using different hash tables in these applications.

2. RELATED WORK

A data structure is defined to be ***history-independent*** if its layout depends only on its current contents, and not the ordering of the operations that created it [13, 24]. For sequential data structures, history-independence is motivated by security concerns, and in particular ensures that examining a structure after its creation does not reveal anything about its history. In this paper, we extend a sequential history-independent hash table based on open addressing [4] to work phase-concurrently. Our motivation is to design a data structure which is deterministic independent of the order of updates. Although we are not concerned with exact memory layout, we want to be able to return the contents of the hash table very quickly and in an order that is independent of when the updates arrived. For a history-independent open addressing table, this can be done easily by packing the non-empty elements into a contiguous array, which just involves a parallel prefix sum and cache-block friendly writes.

Several concurrent hash tables have been developed over the years. There has been significant work on concurrent closed addressing hash tables using separate chaining [16, 9, 19, 23, 28, 12, 33, 20, 14]. It would not be hard to make one of these deterministic when reporting the contents of the buckets since each list could be sorted by a priority at that time. However, such hash tables are expensive relative to open address hashing because they involve more cache misses, and also because they need memory management to allocate and de-allocate the cells for the links. The fastest closed addressing hash we know of is Lea's `ConcurrentHashMap` from the Java Concurrency Package [20], and we compare with a `C++` implementation of it, obtained from Herlihy et al. [15], in Section 6.

Martin and Davis [22], Purcell and Harris [27] and Gao et al. [11] describe lock-free hash tables with open addressing. For deletions, Gao et al.'s version marks the locations with a special "deleted" value, commonly known as tombstones, and insertions and finds simply skip over the tombstones (an insertion is not allowed to fill a tombstone). This means that the only way to remove deleted elements is to copy the whole hash table. All of these hash tables are non-deterministic and quite complex. In our experiments, we use an implementation of non-deterministic linear probing similar to that of Gao et al. (see Section 6).

Herlihy et al. [15] describe and implement an open addressing concurrent hash table called hopscotch hashing, which is based on cuckoo hashing [26] and linear probing. Their hash table guarantees that an element is within K locations of the location it hashed to (where K could be set to the machine word size), so that finds will touch few cache lines. To maintain this property, insertions which find an empty location more than K locations away from the location h that it hashed to will repeatedly displace elements closer to h until it finds an empty slot within K locations of h (or resizes if no empty slot is found). A deletion will recursively bring in elements later in the probe sequence to the empty slot created. Their hash table requires locks and its layout is non-deterministic even if only one type of operation is performed concurrently. Hopscotch hashing is the fastest concurrent hash table available as far as we know, and we use it for comparison in Section 6.

Kim and Kim [18] recently present several implementations of parallel hash tables, though we found our code and the hopscotch hashing code of [15] to be much faster. Van der Vegt and Laarman describe a concurrent hash table using a variant of linear probing called bidirectional linear probing [34, 35], however it requires a monotonic hash function, which may be too restrictive for many applications. Their hash table is non-deterministic and requires locks. Alcantara et al. describe a parallel hashing algorithm using GPUs [1], which involves a synchronized form of cuckoo hashing, and is non-deterministic because collisions are resolved non-deterministically. Concurrent cuckoo hashing has also been discussed by Fan et al. [10], and very recently by Li et al. [21]. The hash table of Fan et al. supports concurrent access by multiple readers and a single writer, but do not support concurrent writers. Li et al. extends this work by supporting concurrent writers as well.

Phase-concurrency has been previously explored in the work on room synchronizations by Blelloch et al. [2]. They describe phase-concurrent implementations of stacks and queues. However, they were concerned only about efficiency, and their data structures are not deterministic even within a single phase. We believe our hash table is the first deterministic phase-concurrent hash table.

3. PRELIMINARIES

We review the sequential history-independent hash table of Blelloch and Golovin [4]. The algorithm is similar to that of standard linear probing. It assumes a total order on the keys used as priorities. For insertion, the only difference is that if during the probe sequence a key currently in the location has lower priority than the key being inserted, then the two keys are swapped. An insertion probes the exact same number of elements as in standard linear probing. For finds, the only difference is that since the keys are ordered by priority, it means that a find for a key k can stop once it finds a location i with a lower priority key. This means that searching for keys not in the table can actually be faster than in standard linear probing. One common method for handling deletions in linear probing is to simply mark the location as "deleted" (a tombstone), and modify the insert and search accordingly. However, this would not be history-independent. Instead, for deletions in the history-independent hash table, the location where the key is deleted is filled with the next lower priority element in the probe sequence that hashed to or after that location (or the empty element if it is at the end of the probe sequence). This process is done recursively until the element that gets swapped in is the empty element.

Our code uses the atomic ***compare-and-swap*** (CAS) instruction. The instruction takes three arguments—a memory location (*loc*), an old value (*oldV*) and a new value (*newV*); if the value stored at *loc* is equal to *oldV* it atomically stores *newV* at *loc* and returns *true*, and otherwise it does not modify *loc* and returns *false*. We use $\&x$ to refer to the memory location of variable x.

We define phase-concurrency as follows:

DEFINITION 1 (PHASE-CONCURRENCY). *A data structure with operations O and operation subsets S is phase-concurrent if $\forall s \in S$, $s \subseteq O$ and all operations in s can proceed concurrently and are linearizable.*

4. DETERMINISTIC PHASE-CONCURRENT HASH TABLE

Our deterministic phase-concurrent hash table extends the sequential history-independent hash table to allow for concurrent inserts, concurrent deletes, and concurrent finds. The contents can also be extracted (referred to as the *elements* operation) easily by simply packing the non-empty cells. Using the notation of Definition 1, our hash table is phase-concurrent with:

- $O = \{$insert, delete, find, elements$\}$, and
- $S = \big\{\{$insert$\}, \{$delete$\}, \{$find, elements$\}\big\}$

The code for insertion, deletion and find is shown in Figure 1, and assumes that the table is not full and that different keys have different priorities (total ordering). For simplicity, the code assumes there is no data associated with the key, although it could easily be modified for key-value pairs. Note that the code works for arbitrary key-value sizes as for structure sizes larger that what a compare-and-swap can operate on, a pointer (which fits in a word) to the structure can be stored in the hash table instead. The code assumes a hash function h that maps keys into the range $[0, \ldots, |M|-1]$, and that the keys have a total priority ordering that can be compared with the function $<_p$. By convention, we assume that the empty element ($\perp$) has lower priority than all other elements. The code uses NEXTINDEX(i) and PREVINDEX(i) to increment and decrement the index modulo the table size. Note that both INSERT and DELETE do not have return values, so we only need to ensure that a set of inserts (or deletes) are commutative with respect to the resulting configuration of the table.

For a given element v, INSERT loops until it finds a location with $\perp$ (Line 3) or it finds that v is already in the hash table (Line 5), at which point it terminates. If during the insert, it finds a location that stores a lower priority value (Line 8), it attempts to replace the value there with v with a CAS, and if successful the lower priority key is temporarily removed from the table and INSERT is now responsible for inserting the replaced element later in the probe sequence, i.e. the replaced element is set to v (Line 9).

For a given element v, DELETE first finds v or an element after v in the probe sequence at location k (Lines 27–29) since v may either not be in the table or its position has been shifted back due to concurrent deletions. If v is not at location k, then DELETE decrements the location (Lines 30–32) until either v is found (Line 33) or the location becomes less than $h(v)$ (Line 30), in which case v is not in the table. After finding v, DELETE finds the replacement element for v by calling FINDREPLACEMENT (Line 34). FINDREPLACEMENT first increments the location until finding a replacement element that is either $\perp$ or a lower priority element that hashes after v (Lines 13–16). The resulting location will be one past the replacement element, so it is decremented on Line 17. Then because the replacement element could have shifted, it decrements the the location until finding the replacement element (Lines 18–23). DELETE then attempts to swap in the replacement element v' on Line 35, and if successful, and $v' \neq \perp$ (Line 36), there is now an additional copy of v' in the table so DELETE is responsible for deleting v' (Lines 37–39). Otherwise, if the CAS was unsuccessful, either v has already been deleted or used as a replacement element so possibly appears at some earlier location. DELETE decrements the location and continues looping (Line 41).

To FIND an element v, the algorithm starts at $h(v)$ and loops upward until finding either an empty location or a location with a key with equal or lower priority (Lines 43–45). Then it returns the result of the comparison of v with that key (Line 46). Since there is a total priority ordering on the keys, $M[i]$ will contain v if and only if v is in the table.

```
1  procedure INSERT($v$)
2     $i = h(v)$
3     while $(v \neq \perp)$
4        $c = M[i]$
5        if $(c = v)$ return
6        elseif $(c >_p v)$ then
7           $i = \text{NEXTINDEX}(i)$
8        elseif $(\text{CAS}(\&M[i], c, v))$ then
9           $v = c$
10          $i = \text{NEXTINDEX}(i)$
11 procedure FINDREPLACEMENT($i$)
12    $j = i$
13    do
14       $j = \text{NEXTINDEX}(j)$
15       $v = M[j]$
16    while $(v \neq \perp$ and $h(v) > i)$
17    $k = \text{PREVINDEX}(j)$
18    while $(k > i)$
19       $v' = M[k]$
20       if $(v' = \perp$ or $h(v') \leq i)$ then
21          $v = v'$
22          $j = k$
23       $k = \text{PREVINDEX}(k)$
24    return $(j, v)$
25 procedure DELETE($v$)
26    $i = h(v)$
27    $k = i$
28    while $(M[k] \neq \perp$ and $v <_p M[k])$
29       $k = \text{NEXTINDEX}(k)$
30    while $(k \geq i)$
31       if $(v = \perp$ or $v \neq_p M[k])$
32          $k = \text{PREVINDEX}(k)$
33       else
34          $(j, v') = \text{FINDREPLACEMENT}(k)$
35          if $(\text{CAS}(\&M[k], v, v'))$ then
36             if $(v' \neq \perp)$ then
37                $v = v'$
38                $k = j$
39                $i = h(v)$
40             else return
41          else $k = \text{PREVINDEX}(k)$
42 procedure FIND($v$)
43    $i = h(v)$
44    while $(M[i] \neq \perp$ and $v <_p M[i])$
45       $i = \text{NEXTINDEX}(i)$
46    return $(M[i] = v)$
```

Figure 1. Phase-concurrent deterministic hashing with linear probing.

Note that for INSERT, DELETE and FIND, it is crucial that the hash table is not full, otherwise the operations may not terminate. Throughout our discussion, we assume wraparound with modulo arithmetic. Since the table is not full, every cluster has a beginning, and when comparing the positions of two elements within a cluster, the "higher" position is the one further from the beginning of the cluster in the forward direction with wraparound. We want to show that when starting with an empty hash table, our phase-concurrent hash table maintains the following invariant:

DEFINITION 2 (ORDERING INVARIANT). *If a key v hashes to location i and is stored in location j in the hash table, then for all $k, i \leq k < j$ it must be that $M[k] \geq_p v$.*

As long as the keys are totally ordered by their priorities, the ordering invariant guarantees a unique representation for a given set of keys [4]. This invariant was shown to hold in the sequential history-independent hash table [4].

The concurrent versions of insert and delete work similarly to the sequential versions, but need to be careful about concurrent modifications. What we show is that the union of the keys being inserted and the current content always equals the union of all initial keys and all insertions that started. A key property to make it work is that since only insertions are occurring, the priority of the keys at a given location can only increase. We note that this implementation should make clear that is not safe to run inserts concurrently with finds, since an unrelated key can be temporarily removed and invisible to a find.

The deletion routine is somewhat trickier. It allows for multiple copies of a key to appear in the table during deletions. In fact, with p concurrent threads it is possible that up to $p+1$ copies of a single key appear in the table at a given time. This might seem counter-intuitive since we are deleting keys. Recall, however, that when a key v is deleted, a replacement v' needs to be found to fill its slot. When v' is copied into the slot occupied by v, there will temporarily be two copies of v', but the delete operation is now responsible for deleting one of them. The sequential code deletes the second copy, but in the concurrent version since there might be concurrent deletes aimed at the same key, the delete might end up deleting the version it copied into, another thread's copy, or it might end up not finding a copy and quitting. The important invariant is that for a value v the number of copies minus the number of outstanding deletes does not change (when a copy is made, the number of copies is increased but so is the number of outstanding deletes). A key property that makes deletions work is that since only deletions are occurring, the priority of the keys at a given location can only decrease, and hence a key can only move to locations with a lower index.

We now prove important properties of our hash table. We use M_v to indicate the set of (non-empty) values contained in the hash table, I_v to indicate the set of values in a collection of insertion operations I, and $|M|$ to indicate the size of the table.

THEOREM 1. *Starting with a table M that satisfies the ordering invariant and with no operations in progress, after any collection of concurrent insertions I complete (and none are in progress) with $|M_v \cup I_v| < |M|$, M will satisfy the following properties:*

- *M contains the union of the keys initially in the table and all values in I, and*
- *M satisfies the ordering invariant.*

Furthermore, all insertion operations are non-blocking and terminate in a finite number of steps.

PROOF. We assume all instructions are linearizable and consider the linearized sequential ordering of operations. We use *step* to refer to a position in this sequential ordering. At a given step, we use I_v to indicate the set of values for which an INSERT has started. Between when an INSERT starts and finishes, we say that it is *active* with some value. At its start, an INSERT(v) is active with the value v, but whenever it performs a successful $\text{CAS}(\&M[i], v, c)$ on Line 8, the INSERT becomes active with the value c on the next step (Line 9)—it is now responsible for inserting c instead of v. When it does a successful $\text{CAS}(\&M[i], v, \perp)$ an INSERT is no longer active—it will terminate as soon as it gets to the next start of the while loop and do nothing to the shared state in the meantime. We also say that an INSERT is no longer active when it reads a value c on Line 4 that is equal to v—it will terminate on Line 5.

We use A_v to indicate the union of values of all INSERT's that are active. We use M_v to indicate the values contained in M on a given step, and M_s to be the initial values contained in M. We will prove that the following invariants are maintained on every step:

1. $M_v \cup A_v = M_s \cup I_v$, and
2. the table M satisfies the ordering invariant.

Since at the end $A_v = \emptyset$, these invariants imply properties 1 and 2 of the theorem.

Invariant 1 is true at the start since A_v and I_v are both empty and $M_s = M_v$ by definition. The invariant is maintained since (1) when an INSERT starts, its value is added to both A_v and I_v and therefore the invariant is unchanged, (2) when an INSERT terminates it reads a $M[i] = v$, so a v is removed from A_v but it exists in M_v so the union is unaffected, (3) every CAS with $c = \perp$ removes a v from A_v but inserts it into M_v, maintaining the union, and (4) every CAS with $c \neq \perp$ swaps an element in M_v with an element in A_v, again maintaining the union. In the code, whenever a CAS succeeds, c is placed in the location where v was (by the definition of CAS) and immediately afterward v is set to c (Line 9).

Invariant 2 is true at the start by assumption. The invariant is maintained since whenever a CAS$(\&M[i], v, c)$ succeeds it must be the case after the CAS that (1) all locations from $h(v)$ up to i have equal or higher priority than v, and (2) all keys that hash to or before i but appear after i have lower priority than v. These properties imply that the ordering invariant is maintained. The first case is true since the only time i is incremented for v is when $c = M[i]$ has a equal or higher priority (Lines 6–7) and since we only swap higher priority values with lower priority ones ($v >_p c$ for all CAS's), once a cell has an equal or larger priority than v, it always will. Also when we have a successful CAS, swap v and c, and increment i, it must be the case that all locations in the probe sequence for the new v and before the new i have priority higher than the new v. This is because it was true before the swap and the only thing changed by the swap was putting the old v into the table, which we know has a higher priority than the new v. The second case of invariant 2 is true since whenever we perform a CAS we are only increasing the priority of the value at that location.

The termination condition is true since when the hash table of size $|M|$ is not full, an INSERT can call NEXTINDEX at most $|M|$ times before finding an empty location. Therefore for p parallel INSERT's, there can be at most $p|M|$ calls to NEXTINDEX. Furthermore, any CAS failure of an INSERT is associated with a CAS success of another INSERT. A CAS success corresponds to either a call to NEXTINDEX (Line 7) or termination of the insertion. Therefore, for a set of p parallel INSERT's, there can be at most $p - 1$ CAS failures for any one CAS success and call to NEXTINDEX. So after $p^2|M|$ CAS attempts, all INSERT's have terminated. It is non-blocking because an INSERT can only fail on a CAS attempt if another INSERT succeeds and thus makes progress. □

THEOREM 2. *Starting with a table M with $|M_v| < |M|$ that satisfies the ordering invariant and with no operations in progress, after any collection of concurrent deletes D complete (and none are in progress), the table will satisfy the following properties:*

- *M contains the difference of the keys initially in the table and all values in D, and*
- *M satisfies the ordering invariant.*

Furthermore, all delete operations are non-blocking and terminate in a finite number of steps.

PROOF. Similar to insertions, from when a DELETE starts until it ends, it is active with some value: initially it is active with the v it was called with and after a successful CAS$(\&M[k], v, v')$ for $v' \neq \perp$ it becomes active with v' (Lines 35–37). A DELETE finishes on CAS$(\&M[k], v, \perp)$ or when the condition of the while loop on Line 30 no longer holds (in this case, it finishes because v is not in the table).

During deletions, the table M can contain multiple copies of a key. The definition of the ordering invariant is still valid with multiple copies of a key, and for a fixed multiplicity the layout remains unique. Unlike insertions, to analyze deletions we need to keep track of multiplicities.

We use D_v to indicate the set of values in D, and M_s the initial contents of M. We use $A(v)$ to indicate the number of active DELETE's with value v, and $M(v)$ to indicate the number of copies of v in M. We will prove that the following invariants are maintained at every step:

1. $\forall v \in M_s$, if $v \in M_s \setminus D_v$ then $M(v) - A(v) = 1$, and otherwise $M(v) - A(v) < 1$,
2. the table M satisfies the ordering invariant allowing for repeated keys, and
3. on Line 30, the index k of a DELETE of v must point to or past the last copy of v ("rightmost" copy with respect to the cluster).

Since at the end $A(v) = 0$ for all v, these invariants prove the properties of the theorem.

Invariant 1 is true at the start since D_v is empty and $\forall v \in M_s$, $A(v) = 0$. To show that the invariant is maintained we consider all events that can change $M(v), A(v)$ or D_v. These are: (1) when a DELETE on v starts, then $A(v)$ is incremented making $M(v) - A(v)$ less than 1 (since it can be at most 1 before the start) and v is added to D_v so v is not in $M_s \setminus D_v$, (2) when a CAS$(\&M[k], v, \perp)$ succeeds, $A(v)$ and $M(v)$ are both decremented, therefore canceling out, (3) when a CAS$(\&M[k], v, v')$ for $v' \neq \perp$ succeeds, then by Lines 35–37, $A(v)$ and $M(v)$ are both decremented, canceling out, and $A(v')$ and $M(v')$ are both incremented, again canceling out, and (4) when a DELETE finishes due to the condition not holding on Line 30, the value v cannot be in the table because of invariant 3, so $A(v)$ is decremented, but $M(v) - A(v)$ is less than 1 both before and after since $M(v) = 0$.

Invariant 2 is true at the start by assumption. The only way it could become violated is if as a result of a CAS$(\&M[k], v, v')$, the value v' falls out of order with respect to values after location j (i.e. there is some key that hashes at or before j, is located after j, and has a higher priority than v'). This cannot happen since the replacement element found is the closest key to j that hashes after j and has lower priority than v. The loop in Lines 13–16 scans upward to find an element that hashes after v in the probe sequence, and the while loop at Lines 18–23 scans downward in case the desired replacement element was shifted down in the meantime by another thread. It is important that this loop runs backwards and is the reason that there are two redundant looking loops, one going up and one going back down.

Invariant 3 is true is since the initial find (Lines 27–29) locates an index of an element with priority lower that v, which must be past v, and FINDREPLACEMENT returns an index at or past the replacement v'. k is only decremented on a failed CAS, which in this case means that v can only be at an index lower than k.

To prove termination, we bound the number of index increments and decrements a single DELETE operation can perform while executing in parallel with other deletes. For a hash table of size $|M|$, the while loop on Lines 30–41 can execute at most $|M|$ times before i changes, and i will only increase since the replacement element must have a higher index than the deleted element. i can increase at most $|M|$ times before $v' = \perp$, so the number of calls to FINDREPLACEMENT is at most $|M|^2$. The number of decrements and assignments to k in the while loop on Lines 30–41 is at most $|M|$ per iteration of the while loop (for a total of $|M|^2$).

FINDREPLACEMENT contains a loop incrementing j, which eventually finishes because the condition on Line 16 will be true for a location containing $\perp$, and a loop decrementing j, which eventually finishes due to the condition on Line 18. So the total number of increments and decrements is at most $2|M|$ per call to FINDREPLACEMENT. The initial find on Lines 27–29 involves at most $|M|$ increments. Therefore, a DELETE operation terminates after at most $|M| + |M|^2 + 2|M|^3$ increments/decrements, independent of the result of the CAS on Line 35. A collection of p DELETE's terminates in at most $p(|M| + |M|^2 + 2|M|^3)$ increments/decrements. Increments, decrements and all instructions in between are non-blocking and thus finish in a finite amount of time. Therefore, concurrent deletions are non-blocking. □

Combining. For a deterministic hash table that stores key-value pairs, if there are duplicate keys, we must decide how to combine the values of these keys deterministically. This can be done by passing a commutative combining function that is applied to the values of pairs with equal keys and updating the location (using a double-word CAS) with a pair containing the key with the combined values. Our experiments in Section 6 use $\min$ or $+$ as the combining function.

Resizing. Using well-known techniques it is relatively easy to extend our hash table with resizing [14]. Here we outline an approach for growing a table based on incrementally copying the old contents to a new table when the load factor in the table is too high. An INSERT can detect that a table is overfull when a probe sequence is too long. In particular, theoretically a probe sequence should not be longer than $k \log n$ with high probability for some constant k that depends on the allowable load factor. Once a process detects that the table is overfull, it allocates a new table of twice the size and (atomically) places a link to the new table accessible to all users. A lock can be used to avoid multiple processes allocating simultaneously. This would mean that an insertion will have to wait between when the lock is taken and the new table is available, but this should be a short time, and only on rare occasions.

Once the link is set, new INSERTs are placed in the new table. Furthermore, as long as the old table is not empty, every INSERT is responsible for copying at least two elements from the old table to the new one. The thread responsible for creating the new table allocates the elements to copy to other threads, and thereafter some form of work-stealing [5] is used to guarantee that a thread has elements to copy when there are still uncopied elements. As long as a constant number of keys are copied for every one that is inserted, the old table will be emptied before the new one is filled. This way only two tables are active at any time. There is an extra cost of indirection on every INSERT since the table has to be checked to find if it has been relocated. However, most of the time this pointer will be in a local cache in shared mode (loaded by any previous table access) and therefore the cost is very cheap. When there are two active tables, finds and deletes would look in both tables.

5. APPLICATIONS

In this section, we describe applications which use our deterministic hash table. For these applications, using a hash table is either the most natural and/or efficient way to implement an algorithm, or it simplifies the implementation compared to directly addressing the memory locations. Our hash table implementation contains a function ELEMENTS() which packs the contents of the table into an array and returns it. It is important that ELEMENTS() is deterministic to guarantee determinism for the algorithms that use it.

Delaunay refinement and breadth-first search use the WRITEMIN function for determinism, which takes two arguments–a memory location *loc* and a value *val* and stores *val* at *loc* if and only if *val* is less than the value at *loc*. It returns *true* if it updates the value at *loc* and *false* otherwise. WRITEMIN is implemented with a compare-and-swap [29].

Remove Duplicates. The ***remove duplicates*** problem takes as input a sequence of elements, a hash function on the elements and a comparison function, and returns a sequence in which duplicate elements are removed. This is a simple application which can be implemented using a hash table by simply inserting all of the elements into the table and returning the result of ELEMENTS(). For determinism, the sequence returned by ELEMENTS() should contain the elements in the same order every time, which is guaranteed by a deterministic hash table. This is an example of an application where the most natural and efficient implementation uses hashing (one could remove duplicates by sorting and removing consecutive equal-valued elements, but it would be less efficient).

Delaunay Refinement. A Delaunay triangulation of n points is a triangulation such that no point is contained in the circumcircle of any triangle in the triangulation. The ***Delaunay refinement*** problem takes as input a Delaunay triangulation and an angle α, and adds new points to the triangulation such that no triangle has an angle less than α. We refer to a triangle with an angle less than α as a ***bad triangle***.

Initially all of the bad triangles of the input triangulation are computed and stored into a hash table. On each iteration of Delaunay refinement, the contents of the hash table are obtained via a call to ELEMENTS() and the bad triangles mark (using a WRITEMIN with their index in the sequence) all of the triangles that would be affected if they were to be inserted. Bad triangles whose affected triangles all contain their mark are "active" and can proceed to modify the triangulation by adding their center point. This method guarantees there are no conflicts, as any triangle in the triangulation is affected by at most one active bad triangle. During each iteration of the refinement, new triangles with angles less than α are generated and they are inserted into the hash table as they are discovered. This process is repeated until either a specified number of new points are added or the triangulation contains no more bad triangles. For determinism, it is important that the call to ELEMENTS() is deterministic, as this makes the indices/priorities of the bad triangles, and hence the resulting triangulation deterministic.

This is an example of an application where using a hash table significantly simplifies the implementation. Prior to inserting a point, it is hard to efficiently determine how many new bad triangles it will create, and pre-allocate an array of the correct size to allow for storing the new bad triangles in parallel.

Suffix Tree. A ***suffix tree*** stores all suffixes of a string S in a trie where internal nodes with a single child are contracted. A suffix tree allows for efficient searches for patterns in S, and also has many other applications in string analysis and computational biology. To allow for expected constant time look-ups, a hash table is used to store the children of each internal node. Our phase-concurrent hash table allows for parallel insertions of nodes into a suffix tree and parallel searches on the suffix tree. This is an example of an application where hash tables are used for efficiency, and where the inserts and finds are naturally split into two phases.

Edge Contraction. The ***edge contraction*** problem takes as input a sequence of edges (possibly with weights) and a label array R, which specifies that vertex v should be relabeled with the value

$R[v]$. It returns a sequence of unique edges relabeled according to R. Edge contraction is used in recursive graph algorithms where certain vertices are merged into "supervertices" and the endpoints of edges need to be relabeled to the IDs of these supervertices. Duplicate edges are processed differently depending on the algorithm.

To implement edge contraction, we can insert the edges into a hash table using the two new vertex IDs as the key, and any data on the edge as the value. A commutative combining function can be supplied for combining data on duplicate edges. For example, we might keep the edge with minimum weight for a minimum spanning tree algorithm, or add the edge weights together for a graph partitioning algorithm [17]. To obtain the relabeled edges for the next iteration, we make a call to ELEMENTS(). To guarantee determinism in the algorithm, the hash table must be deterministic. This idea is used to remove duplicate edges on contraction in a recent connected components implementation [31].

Breadth-First Search. The ***breadth-first search*** (BFS) problem takes a graph G and a starting vertex r, and returns a breadth-first search tree rooted at r containing all nodes reachable from r. Vertices in each level of the BFS can be visited in parallel. A parallel algorithm stores the current level of the search in a ***Frontier*** array, and finds all unvisited neighbors of the Frontier array that belong to the next level in parallel. During this process, it also chooses a parent vertex for each unvisited neighbor in the next level. However, if multiple vertices on the frontier share an unvisited neighbor, one must decide which vertex becomes the parent of the neighbor. This can be done deterministically with the WRITEMIN function.

Another issue is how to generate the new Frontier array for the next level. One option is to have all parents copy all of its unvisited neighbors of the current Frontier array into a temporary array. To do this in parallel, one must first create an array large enough to contain all unvisited neighbors of all vertices in the Frontier array (since at this point we have not assigned parents yet), assign segments of the array to each vertex in Frontier, and have each vertex in Frontier copy unvisited neighbors that it is a parent of into the array. This array is then packed down with a prefix sums and assigned to Frontier.

An alternative solution is to use a concurrent hash table and insert unvisited neighbors into the table. Obtaining the next Frontier array simply involves a call to ELEMENTS(). With this method, duplicates are removed automatically, and the packing is hidden from the user. This leads to a much cleaner solution. If one wants to look at or store the frontiers or simply generate a level ordering of the vertices, then it is important that ELEMENTS() is deterministic. The pseudo-code for this algorithm is shown in Figure 2. This method gives a deterministic BFS tree. In Section 6 we show that using our deterministic phase-concurrent hash table does not slow down the code by much compared to the best previous deterministic BFS code [30], which uses memory directly as described in the first method above.

Spanning Forest. A ***spanning forest*** for an undirected graph G, is a subset of the edges that forms a forest (collection of trees) and spans all vertices in G. One way to implement a deterministic parallel spanning forest algorithm is to use the technique of *deterministic reservations* described by Blelloch et al. [3]. In this technique, the edges are assigned unique priorities at the beginning. Each iteration contains a reservation phase and a commit phase. The reservation phase involves the edges finding the components they connect (using a union-find data structure) and then reserving their components if they are different. The commit phase involves the edges checking if they made successful reservations on at least one component, and if so linking the components together. The unsuccessful edges connecting different components are kept for the next iteration and the process is repeated until no edges remain.

```
procedure BFS($G$, $r$)                          ▷ $r$ is the root
    Parents = $\{\infty, \dots, \infty\}$        ▷ initialized to all $\infty$ (unvisited)
    Parents[$r$] = $r$
    Frontier = $\{r\}$
    while (Frontier $\neq \{\}$) do
        Create hash table $T$
        parfor $v \in$ Frontier do               ▷ loop over frontier vertices
            parfor ngh $\in N(v)$ do             ▷ loop over neighbors
                if (WRITEMIN(&Parents[ngh], $v$)) then
                    $T$.INSERT(ngh)
        Frontier = $T$.ELEMENTS()                ▷ get contents of $T$
        parfor $v \in$ Frontier do
            Parents[v] = −Parents[v]             ▷ negative indicates visited
    return Parents
```

Figure 2. Hash Table-Based Breadth-First Search

If the vertex IDs are integers from the range $[1, \dots, n]$, then an array of size n can be used to store the reservations. However, if the IDs are much larger integers or strings, it may be more convenient to use a hash table to perform the reservations to avoid vertex relabeling. Determinism is maintained if the hash table is deterministic. For the reservation phase, edges insert into a hash table each of its vertices (as the key), with value equal to the edge priority. For a deterministic hash table, if duplicate vertices are inserted, the one with the value with the highest priority remains in the hash table. In the commit phase, each edge performs a hash table find on the vertex it inserted and if it contain the edge's priority value, then it proceeds with linking its two components together. We show in Section 6 that the implementation of spanning forest using a hash table is only slightly slower than the array-based version.

6. EXPERIMENTS

In this section, we analyze the performance of our concurrent deterministic history-independent hash table (***linearHash-D***) on its own, and also when used in the applications described in Section 5.

We compare it with two non-deterministic phase-concurrent hash tables that we implement, and with the best existing concurrent hash tables that we know of (hopscotchHash and chainedHash). ***linearHash-ND*** is a concurrent version of linear probing that we implement, which places values in the first empty location and hence depends on history (non-deterministic). It is based on the implementation of Gao et al. [11], except that for deletions it shifts elements back instead of using tombstones, and does not support resizing. We note that in linearHash-ND, insertions and finds can proceed concurrently (although we still separate them in our experiments), since inserted elements are not displaced. ***cuckooHash*** is a concurrent version of cuckoo hashing that we implement, which locks two locations for an element insertion, places the element in one of the locations, and recursively inserts any evicted elements. To prevent deadlocks, it acquires the locks in increasing order of location. It is non-deterministic because an element can be placed in either of its two locations based on the order of insertions. For key-value pairs, on encountering duplicate keys linearHash-D uses a priority function [29] on the values to deterministically decide which pair to keep, while the non-deterministic hash tables do not replace on duplicate keys.

hopscotchHash is a fully-concurrent open-addressing hash table by Herlihy et al. [15], which is based on a combination of linear probing and cuckoo hashing. It uses locks on segments of the hash table during insertions and deletions. We noticed that there is a time-stamp field in the code which is not needed if operations of different types are not performed concurrently. We modified the code accordingly and call this phase-concurrent version ***hopscotchHash-PC***. ***chainedHash*** is a widely-used fully-concurrent closed-addressing hash table by Lea [20] which places elements in linked lists. It was originally implemented in Java, but we were able to obtain a C++ version from the authors of [15]. We also tried the chained hash map (`concurrent_hash_map`) implemented as part of Intel Threading Building Blocks, but found it to be slower than chainedHash. We implement the ELEMENTS() routine for both hopscotch hashing and chained hashing, as the implementations did not come with this routine. For hopscotch hashing, we simply pack out the empty locations. For chained hashing, we first count the number of elements per bucket by traversing the linked lists, compute each bucket's offset into an array using a parallel prefix sum, and then traverse the linked lists per bucket copying elements into the array (each bucket can proceed in parallel). The original implementation of chainedHash acquires a lock at the beginning of an insertion and deletion. This leads to high lock contention for distributions with many repeated keys. We optimized the chained hash table such that insertion only acquires a lock after an initial find operation does not find the key, and deletion only acquires a lock after an initial find operation successfully finds the key. This contention-reducing version is referred to as ***chainedHash-CR***.

We also include timings for a serial implementation of the history-independent hash table using linear probing (***serialHash-HI***) and a serial implementation using standard linear probing (***serialHash-HD***).

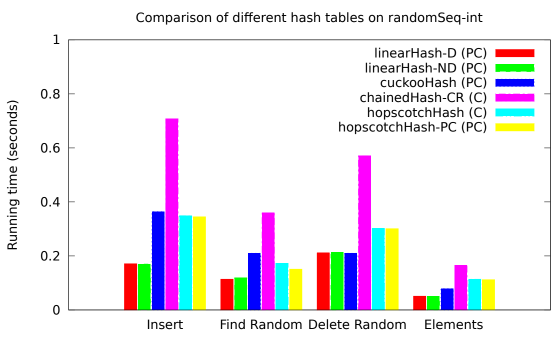
For the applications, we compare their performance using the phase-concurrent hash tables that we implement and the chained hash table[1]. For breadth-first search and spanning tree we also compare with implementations that directly address memory and show that the additional cost of using hash tables is small.

We run our experiments on a 40-core (with hyper-threading) machine with 4×2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache) and 256GB of main memory. We run all parallel experiments with two-way hyper-threading enabled, for a total of 80 threads. We compiled all of our code with g++ version 4.8.0 with the `-O2` flag. The parallel codes were compiled with Cilk Plus, which is included in g++.

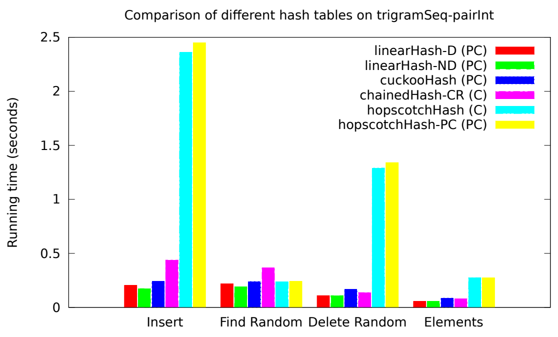
For our experiments, we use six input distributions from the Problem Based Benchmark Suite (PBBS) [30]. ***randomSeq-int*** is a sequence of n random integer keys in the range $[1, \dots, n]$ drawn from a uniform distribution. ***randomSeq-pairInt*** is a sequence of n key-value pairs of random integers in the range $[1, \dots, n]$ drawn from a uniform distribution. ***trigramSeq*** is a sequence of n string keys generated from trigram probabilities of English text (there are many duplicate keys in this input). ***trigramSeq-pairInt*** has the same keys as trigramSeq, but each key maintains a corresponding random integer value. For this input, the key-value pairs are stored as a pointer to a structure with a pointer to a string, and therefore involves an extra level of indirection. ***exptSeq-int*** is a sequence of n random integer keys drawn from an exponential distribution—this input is also used to test high collision rates in the hash table. ***exptSeq-pairInt*** contains keys from the same distribution, but with an additional integer value per key. For all distributions, we used $n = 10^8$. For the open addressing hash tables, we initialized a table of size 2^{28}.

Figures 3(a) and 3(b) compare the hash tables for several operations on randomSeq-int and trigramSeq-pairInt, respectively. For

[1]The source code for hopscotch hashing that we obtained online sometimes does not work correctly on our Intel machine (it was originally designed for a Sun UltraSPARC machine), so we do not use it in our applications.



(a) Times (seconds) for 10^8 operations on randomSeq-int.



(b) Times (seconds) for 10^8 operations on trigramSeq-pairInt.

Figure 3. Times (seconds) for 10^8 operations for the hash tables on 40 cores (with hyper-threading). (PC) indicates a phase-concurrent implementation and (C) indicates a concurrent implementation.

Insert, we insert a random set of keys from the distribution starting from an empty table. For **Find Random** and **Delete Random** we first insert n elements (not included in the time) and then perform the operations for a random set of keys from the distribution. **Elements** is the time for returning the contents of the hash table in a packed array. Table 1 lists the parallel and serial running times (seconds) for insertions, finds, deletions and returning the elements for the various hash tables on different input sequences. For **Find** and **Delete** we first insert n elements (not included in the time) and then perform the operations either on the same keys (**Inserted**) or for a random set of keys from the distribution (**Random**).

As Figure 3 and Table 1 indicate, insertion, finds and deletions into the deterministic (history-independent) hash table are slightly more expensive than into the history-dependent linear probing version. This is due to the overhead of swapping and checking priorities. Elements just involves packing the contents of the hash table into a contiguous array, and since for a given input, the locations occupied in the hash table are the same in the linear probing tables, the times are roughly the same (within noise) between the two serial versions and the two parallel version. On a single thread, the serial versions are cheaper since they do not use a prefix sum.

Overall, linearHash-D and linearHash-ND are faster than cuckooHash, since cuckooHash involves more cache misses on average (it has to check two random locations). Elements is also slower for cuckooHash because each hash table entry includes a lock, which increases the memory footprint. For random integer keys, our linear probing hash tables are 2.3–4.1$\times$ faster than chainedHash and chainedHash-CR, as chained hashing incurs more cache misses. As expected, in parallel chainedHash performs very poorly under the sequences with many duplicates (trigramSeq, trigramSeq-pairInt, exptSeq and exptSeq-pairInt) due to high lock contention, while chainedHash-CR performs better.

(a) **Insert**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	3.94	–	4.76	–	5.42	–	8.58	–	3.01	–	3.58	–
serialHash-HD	3.89	–	4.43	–	4.99	–	7.71	–	2.91	–	3.04	–
linearHash-D	4.53	0.171	5.45	0.216	5.53	0.115	8.66	0.204	3.08	0.119	3.71	0.141
linearHash-ND	4.52	0.17	4.77	0.213	5.02	0.108	8.2	0.174	2.96	0.109	3.12	0.119
cuckooHash	7.91	0.364	14.0	0.43	8.3	0.177	12.0	0.242	4.7	0.184	7.23	0.208
chainedHash	13.3	0.774	15.3	0.784	9.54	9.78	14.0	18.4	7.9	2.57	8.48	5.25
chainedHash-CR	14.4	0.708	16.8	0.71	9.1	0.324	13.7	0.438	7.19	0.35	7.56	0.401
hopscotchHash	9.19	0.349	9.21	0.363	7.04	1.54	9.63	2.36	6.15	1.97	6.0	2.02
hopscotchHash-PC	9.18	0.345	9.21	0.365	7.03	1.55	9.59	2.45	6.16	1.94	5.99	2.09

(b) **Find Random**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	3.97	–	4.17	–	6.11	–	10.9	–	3.38	–	3.12	–
serialHash-HD	4.03	–	4.36	–	5.95	–	9.42	–	2.77	–	2.91	–
linearHash-D	4.23	0.114	4.19	0.149	6.17	0.12	10.6	0.219	3.16	0.069	3.11	0.07
linearHash-ND	4.02	0.119	4.35	0.144	5.89	0.117	10.1	0.19	2.79	0.067	2.91	0.078
cuckooHash	6.64	0.21	8.13	0.255	7.7	0.174	12.4	0.24	5.1	0.127	6.1	0.14
chainedHash	9.04	0.356	9.06	0.3	9.84	0.247	15.0	0.364	5.0	0.189	6.01	0.17
chainedHash-CR	9.06	0.359	9.05	0.301	9.74	0.245	15.0	0.365	5.9	0.188	5.99	0.168
hopscotchHash	5.2	0.173	5.02	0.169	6.8	0.167	10.2	0.236	3.51	0.094	3.49	0.091
hopscotchHash-PC	4.76	0.151	4.72	0.15	6.84	0.167	9.7	0.241	3.42	0.088	3.43	0.088

(c) **Find Inserted**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	3.36	–	3.59	–	5.78	–	10.3	–	2.8	–	2.78	–
serialHash-HD	3.22	–	3.45	–	5.6	–	8.66	–	2.48	–	2.62	–
linearHash-D	3.36	0.109	3.6	0.142	5.73	0.114	9.94	0.204	2.6	0.067	2.6	0.068
linearHash-ND	3.22	0.106	3.44	0.125	5.5	0.11	9.55	0.195	2.48	0.064	2.61	0.073
cuckooHash	6.03	0.205	7.34	0.228	7.88	0.165	11.6	0.222	4.66	0.12	5.59	0.13
chainedHash	7.83	0.403	7.91	0.327	9.47	0.253	14.5	0.367	5.68	0.214	5.73	0.191
chainedHash-CR	7.87	0.406	7.89	0.327	9.36	0.249	14.5	0.366	5.69	0.213	5.7	0.188
hopscotchHash	4.67	0.168	4.67	0.166	6.44	0.157	9.31	0.22	3.22	0.09	3.22	0.09
hopscotchHash-PC	4.45	0.154	4.46	0.15	6.48	0.157	9.25	0.24	3.14	0.083	3.16	0.084

(d) **Delete Random**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	4.89	–	5.8	–	3.69	–	4.17	–	2.82	–	3.13	–
serialHash-HD	4.87	–	5.85	–	3.09	–	3.77	–	2.83	–	3.14	–
linearHash-D	5.84	0.211	7.27	0.229	3.79	0.071	4.6	0.109	2.95	0.0968	3.7	0.099
linearHash-ND	5.9	0.213	7.43	0.235	3.85	0.071	4.64	0.109	3.02	0.0936	3.76	0.107
cuckooHash	6.16	0.21	7.16	0.266	5.57	0.15	8.01	0.166	4.25	0.109	4.69	0.142
chainedHash	16.2	0.63	16.4	0.597	4.79	2.38	6.02	2.7	7.16	2.79	7.28	7.01
chainedHash-CR	15.0	0.571	14.9	0.512	4.33	0.11	5.19	0.137	6.04	0.204	6.03	0.358
hopscotchHash	7.19	0.302	7.1	0.316	4.16	1.32	4.89	1.29	4.36	1.32	4.31	1.25
hopscotchHash-PC	7.07	0.301	7.06	0.32	4.15	1.33	4.95	1.34	4.36	1.31	4.28	1.24

(e) **Delete Inserted**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	5.05	–	6.1	–	3.51	–	4.36	–	3.11	–	3.5	–
serialHash-HD	5.15	–	6.37	–	3.48	–	4.01	–	3.13	–	3.5	–
linearHash-D	6.13	0.24	7.98	0.264	3.73	0.068	4.59	0.102	3.33	0.115	4.18	0.126
linearHash-ND	6.36	0.242	8.38	0.269	3.8	0.07	4.34	0.102	3.35	0.11	4.23	0.119
cuckooHash	6.16	0.217	7.41	0.272	5.74	0.143	7.72	0.16	4.41	0.114	4.99	0.147
chainedHash	15.7	0.737	16.6	0.69	4.22	2.2	5.15	2.65	6.8	2.59	6.92	4.58
chainedHash-CR	14.9	0.714	14.9	0.624	3.77	0.126	4.62	0.153	5.64	0.372	5.65	0.45
hopscotchHash	7.2	0.33	7.8	0.343	3.96	1.32	4.89	1.28	4.69	1.38	4.54	1.29
hopscotchHash-PC	7.06	0.319	7.75	0.347	3.93	1.31	4.85	1.36	4.68	1.38	4.52	1.27

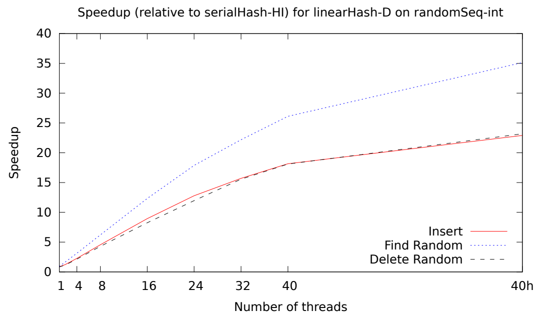
(f) **Elements**	randomSeq-int		randomSeq-pairInt		trigramSeq		trigramSeq-pairInt		exptSeq-int		exptSeq-pairInt	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serialHash-HI	0.974	–	1.1	–	0.758	–	0.753	–	0.603	–	0.821	–
serialHash-HD	0.986	–	1.08	–	0.759	–	0.761	–	0.554	–	0.814	–
linearHash-D	1.55	0.0511	2.25	0.0875	1.41	0.0575	1.43	0.056	1.05	0.0468	1.7	0.0514
linearHash-ND	1.55	0.0504	2.21	0.0857	1.42	0.0576	1.46	0.0554	1.06	0.0477	1.69	0.0794
cuckooHash	1.91	0.0791	2.54	0.115	2.45	0.0856	2.4	0.0866	1.64	0.0733	2.23	0.101
chainedHash	6.3	0.159	6.47	0.132	1.96	0.0782	1.97	0.0789	3.36	0.0934	3.38	0.0963
chainedHash-CR	6.33	0.165	6.44	0.131	1.97	0.0784	1.96	0.0785	3.38	0.091	3.37	0.0938
hopscotchHash	2.25	0.114	2.7	0.15	2.1	0.228	2.16	0.275	2.14	0.103	2.6	0.127
hopscotchHash-PC	2.26	0.112	2.73	0.147	2.09	0.229	2.16	0.274	2.14	0.1	2.61	0.128

Table 1. Times (seconds) for hash table operations with $n = 10^8$. (40h) indicates 40 cores with hyper-threading, and (1) indicates one thread.

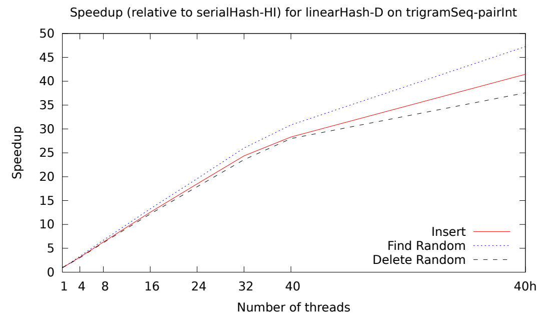
Compared to hopscotch hashing, which is the fastest concurrent open addressing hash table that we are aware of, both of our phase-concurrent versions of linear probing are faster. For random integer keys, the deterministic version is about $2\times$ faster than hopscotch hashing for inserts, and $1.3\times$ faster for finds and deletes. For elements, we are also faster because we store less information per hash table entry. Hopscotch hashing does not get good speedup for insertions and deletions for the sequences with many repeats (i.e. the trigram and exponential sequences) due to lock contention. Compared to cuckooHash, on the lower-contention random integer sequence, hopscotch hashing is faster for finds and inserts but slower for deletes and elements (it stores more data).

Figures 4(a) and 4(b) show the speedup of linearHash-D relative to serialHash-HI on varying number of threads on randomSeq-int and trigramSeq-pairInt, respectively. We use a hash table of size

2^{28} and applied 10^8 operations of each type. We see that all of the operations get good speedup as we increase the number of threads.



(a) Speedup on randomSeq-int.



(b) Speedup on trigramSeq-pairInt.

Figure 4. Speedup relative to serialHash-HI for linearHash-D versus number of threads. (40h) indicates 80 hyper-threads.

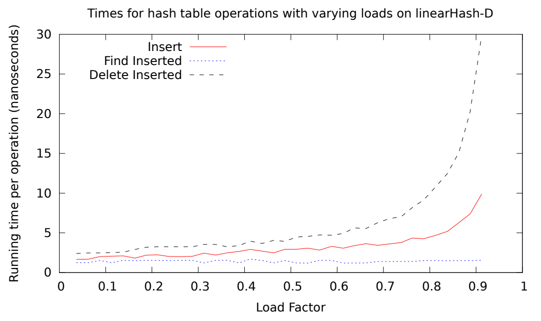


Figure 5. Times (nanoseconds) per operation with varying loads for linearHash-D on 40 cores (with hyper-threading). Values on the x-axis indicate the load factor (fraction of the table that is full).

Figure 5 shows the per operation running times on linearHash-D with varying loads. For this experiment, we used a hash table of size 2^{27}, and the table is first filled to the specified load before timing the operations. We note that inserts and deletes become more expensive as load increases, with a rapid increase as the load approaches 1.

We also compare the performance of hash table inserts to doing random writes (times for 10^8 writes are shown in Table 2). For a uniformly random sequence (randomSeq-int), parallel insertion into the deterministic hash table with a load of $1/3$ is $1.3\times$ slower than parallel random writes. We also compare with a conditional random write, which only writes to the location if it is empty, and the parallel running time is about the same as for random writes.

Very recently, Li et al. [21] describe a concurrent cuckoo hash table that achieves up to 40 million inserts per second for filling up a hash table to 95% load using 16 cores and integer key-value pairs, where the integers are 8 bytes each. On 16 cores, our linearHash-ND performs 75 million inserts per second and linearHash-D performs 65 million inserts per second filling the table up to 95% load and using integer key-value pairs with 8-byte integers. As the performance of linear probing degrades significantly at high loads, for smaller loads we are faster than their hash table by a larger factor. However, the hash table of Li et al. is fully-concurrent, and optimizations can probably be made for a phase-concurrent setting.

Memory Operation	(1)	(40h)
Random write	1.62	0.129
Conditional random write	1.82	0.131
Hash table insertion	4.53	0.171

Table 2. Times (seconds) for 10^8 random writes (scatter)

Applications. For the applications, we compare implementations using different versions of the hash tables. For the open addressing hash tables, a larger table size decreases the load and usually leads to faster insertions, deletions and finds, but our algorithms require either returning the elements of the hash table or mapping over the elements, which takes time proportional to the size of the hash table. Due to this trade-off, we chose table sizes which gave the best overall performance per application. For chained hashing, we only present the times for chainedHash-CR, as we tried both chainedHash and chainedHash-CR and found that the timings were within 5% of each other as the inputs do not exhibit high contention. We did not use hopscotch hashing in our applications as the implementation we obtained did not always work correctly (see Footnote 2).

For remove duplicates, we use the same input distributions as in the previous experiments ($n = 10^8$), though we only report a subset of them due to space constraints. Removing duplicates involves a phase of insertions, which is more efficient with a larger table in open addressing, and a call to ELEMENTS(), which is more efficient with a smaller table in open addressing. We found that setting the table size to 2^{27} for the open addressing hash tables gave the best overall performance. The times for using linearHash-D, linearHash-ND, cuckooHash and chainedHash to remove duplicates are shown in Table 3. We see that our deterministic version of linear probing is 7–23% slower than our non-deterministic version on the key-value inputs with many duplicates because the deterministic version may perform a swap on duplicate keys. Both linear probing tables outperform the cuckoo and chained hash tables.

Remove Duplicates	randomSeq-int		trigramSeq-pairInt		exptSeq-int	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	6.36	0.212	10.4	0.242	3.72	0.139
linearHash-ND	6.33	0.212	9.64	0.213	3.63	0.116
cuckooHash	11.0	0.417	12.9	0.3	5.76	0.185
chainedHash-CR	19.9	1.32	15.6	0.586	9.67	0.541

Table 3. Times (seconds) for Remove Duplicates

For Delaunay refinement, we use as input the Delaunay triangulation of the 2D-cube and 2D-kuzmin geometry data from the PBBS [30], each of which contain 5 million points. The times for the hash table portion of one iteration of Delaunay refinement, which involves a call to ELEMENTS() and hash table insertions, are shown in Table 4. For the open addressing hash tables, we use a table size of twice the number of bad triangles rounded up to the nearest power of 2. LinearHash-D performs slightly slower than linearHash-ND, but allows for a deterministic implementation of Delaunay refinement. Both of our linear probing hash tables outperform the cuckoo hash table and chained hash tables for this application.

For suffix trees, we use three real-world texts (from `http://people.unipmn.it/manzini/lightweight/corpus/`): ***etext99*** (105 MB) and ***rctail96*** (115 MB) are taken from real English texts, and ***sprot34.dat*** (110 MB) is taken from a protein sequence. We measure the times for the portion of the code which

Delaunay Refinement	2DinCube		2Dkuzmin	
	(1)	(40h)	(1)	(40h)
linearHash-D	1.01	0.033	0.986	0.033
linearHash-ND	0.95	0.031	0.956	0.032
cuckooHash	1.62	0.051	1.56	0.054
chainedHash-CR	1.89	0.079	1.95	0.099

Table 4. Times (seconds) for Delaunay Refinement

inserts the nodes into the suffix tree (represented with a hash table), and also the times for searching one million random strings in the suffix tree (which uses hash table finds). For the searches, we use strings with lengths distributed uniformly between 1 and 50. Half of the search strings are random sub-strings of the text, which should all be found, and the other half are random strings, most of which will not be found. For the open addressing hash tables, we use a size of twice the number of nodes in the suffix tree rounded up to the nearest power of 2. The times are shown in Table 5. Again our deterministic linear probing hash table is only slightly slower than our non-deterministic one, and both of them outperform the cuckoo hash table and chained hash tables.

(a) **Suffix Tree Insert (Size)**	etext99 (105 MB)		rctial96 (115 MB)		sprot34.dat (110 MB)	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	4.84	0.12	4.96	0.117	4.77	0.115
linearHash-ND	4.6	0.114	4.74	0.112	4.57	0.109
cuckooHash	9.11	0.184	8.85	0.177	8.6	0.172
chainedHash-CR	7.72	0.256	7.65	0.238	7.39	0.235

(b) **Suffix Tree Search**	etext99		rctial96		sprot34.dat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	1.08	0.023	0.728	0.015	0.803	0.017
linearHash-ND	1.07	0.023	0.713	0.015	0.787	0.017
cuckooHash	1.22	0.026	0.826	0.017	0.911	0.019
chainedHash-CR	1.35	0.03	0.91	0.02	1.01	0.023

Table 5. Times (seconds) for Suffix Tree operations

For edge contraction, breadth-first search and spanning forest, we use use three undirected graphs from PBBS. ***3D-grid*** is a grid graph in 3-dimensional space where every vertex has six edges, each connecting it to its 2 neighbors in each dimension. It has a total of 10^7 vertices and 3×10^7 edges. ***random*** is a random graph where every vertex has five edges to neighbors chosen randomly. It has a total of 10^7 vertices and 5×10^7 edges. The ***rMat*** graph [7] has a power-law degree distribution. It has a total of 2^{24} vertices and 5×10^7 edges.

We time one round of edge contraction when used as a part of a graph separator program. A maximal matching is first computed on the input graph to generate the vertex relabelings (not timed) and then edges with their relabeled endpoints are inserted into a hash table if the endpoints are different (timed). Duplicate edges between the same vertices after relabeling have their weights added together using a fetch-and-add. Since in linearHash-D, the edges may shift around during insertions, it requires using compare-and-swap on the entire edge. On the other hand, in linearHash-ND, once an element is inserted it no longer moves, so when encountering duplicate edges, it only needs to add the weight of the duplicate edge to the inserted edge and can use the faster `xadd` atomic hardware primitive to do this. The linear probing hash table sizes are set to $4/3$ times the number of edges, rounded up to the nearest power of 2. The times are shown in Table 6. Our deterministic version of linear probing is about 15% slower than the non-deterministic version, but guarantees a a deterministic ordering of the edges and hence a deterministic graph partition when used in a graph partitioning algorithm. Again, both of our linear probing hash tables outperform cuckoo hashing and chained hashing.

For each iteration of BFS, we use a hash table with size equal to the sum of the degrees of the frontier vertices rounded up to the nearest power of 2 for linear probing and twice that size for

Edge Contraction	3D-grid		random		rMat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
linearHash-D	6.03	0.154	10.9	0.265	10.8	0.272
linearHash-ND	5.4	0.136	9.09	0.229	9.18	0.235
cuckooHash	9.31	0.269	16.8	0.447	16.7	0.455
chainedHash-CR	11.6	0.55	20.1	0.907	20.0	0.917

Table 6. Times (seconds) for Edge Contraction

cuckoo hashing. Table 7 gives the running times for various BFS implementations where ***serial*** is the serial implementation, ***array*** is the implementation which uses a temporary array to compute new frontiers as described in Section 5. LinearHash-D is slightly slower than linearHash-ND, and both linear probing tables outperform cuckooHash and chainedHash-CR. In parallel, the deterministic hash table-based BFS is 16–35% slower than the array-based BFS. On a single thread, linearHash-D is faster on two of the inputs, however it does not get as good speedup. We observed that in parallel, the linear probing hash table-based BFS implementations spend 70-80% of the time performing hash table insertions, and sequentially they spend 80-90% of the time on insertions.

Breadth-First Search	3D-grid		random		rMat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial	2.3	–	2.89	–	3.33	–
array	3.57	0.271	4.89	0.169	6.81	0.225
linearHash-D	3.2	0.367	5.44	0.211	6.25	0.262
linearHash-ND	3.21	0.362	5.43	0.204	6.24	0.256
cuckooHash	4.56	0.454	7.3	0.292	9.1	0.373
chainedHash-CR	5.08	1.14	8.11	0.343	9.78	0.439

Table 7. Times (seconds) for Breadth-First Search

For spanning forest, we compare versions using hash tables with a serial version and an array-based version. For the versions using open addressing tables, we use a table of size twice the number of vertices rounded up to the nearest power of 2. The timings are shown in Table 8. LinearHash-D and linearHash-ND perform similarly, and they both outperform the cuckoo and chained hash tables. The deterministic hash table-based version is 14–26% slower than the array-based version, but avoids vertex relabeling when the vertex IDs are integers from a large range or are not integers.

Spanning Forest	3D-grid		random		rMat	
	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial	1.42	–	1.87	–	2.35	–
array	3.54	0.186	4.68	0.226	6.13	0.289
linearHash-D	4.73	0.212	5.87	0.286	7.31	0.346
linearHash-ND	4.8	0.215	5.86	0.282	7.36	0.344
cuckooHash	5.86	0.251	7.08	0.341	9.08	0.387
chainedHash-CR	6.04	0.408	7.46	0.544	9.73	0.662

Table 8. Times (seconds) for Spanning Forest

For BFS and spanning forest, our experiments show that hash tables can replace directly addressing memory, while incurring only a small performance penalty.

7. CONCLUSION

We have described a phase-concurrent deterministic hash table based on linear probing and proved its correctness. We have shown experimentally that the performance of various operations on the deterministic hash table is competitive with those of a phase-concurrent history-dependent one based on linear probing, and achieves good speedup. Our deterministic hash table outperforms the best existing concurrent hash tables. We have described six applications which use phase-concurrent hash tables that we are aware of. We show that using our deterministic hash table within these applications gives performance that is competitive with or only slightly slower than using our non-deterministic linear probing hash table, and is faster than using the existing concurrent tables. Future work includes implementing automatic resizing in our hash table and exploring ways to automatically separate operations into phases efficiently, e.g. by using room synchronizations [2].

Acknowledgments

This work is supported by the National Science Foundation under grant number CCF-1314590, the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and a Facebook Graduate Fellowship. We thank Kristy Gardner and Phillip Gibbons for useful discussions.

References

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the GPU. *ACM Trans. Graph.*, 28(5), 2009.

[2] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Scalable room synchronizations. *Theory Comput. Syst.*, 36(5):397–430, 2003.

[3] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *Proceedings of Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[4] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *IEEE Symposium on Foundations of Computer Science*, pages 272–282, 2007.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[6] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, pages 442–446, 2004.

[8] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2009.

[9] C. S. Ellis. Concurrency in linear hashing. *ACM Trans. Database Syst.*, 12(2):195–217, 1987.

[10] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 371–384, 2013.

[11] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005.

[12] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Principles of Distributed Computing*, pages 260–269, 2002.

[13] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. *Algorithmica*, pages 57–74, 2005.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[15] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364, 2008.

[16] M. Hsu and W.-P. Yang. Concurrent operations in extendible hashing. In *Proceedings of Very Large Data Bases*, pages 241–247, 1986.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[18] E. Kim and M.-S. Kim. Performance analysis of cache-conscious hashing techniques for multi-core CPUs. *International Journal of Control and Automation*, 6(2):121–134, Apr. 2013.

[19] V. Kumar. Concurrent operations on extendible hashing and its performance. *Commun. ACM*, 33(6):681–694, 1990.

[20] D. Lea. Hash table util.concurrent.concurrenthashmap in java.util.concurrent the Java Concurrency Package.

[21] X. Li, D. G. Anderson, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*, 2014.

[22] D. R. Martin and R. C. Davis. A scalable non-blocking concurrent hash table implementation with incremental rehashing. Unpublished manuscript, 1997.

[23] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Symposium on Parallelism in Algorithms and Architectures*, pages 73–82, 2002.

[24] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *Symposium on Theory of Computing*, pages 492–501, 2001.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Architectural Support for Programming Languages and Operating Systems*, pages 97–108, 2009.

[26] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[27] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. In *International Symposium on Distributed Computing*, pages 108–121, 2005.

[28] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

[29] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *Symposium on Parallelism in Algorithms and Architectures*, pages 152–163, 2013.

[30] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *Symposium on Parallelism in Algorithms and Architectures*, pages 68–70, 2012.

[31] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Symposium on Parallelism in Algorithms and Architectures*, 2014.

[32] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *ACM POPL*, pages 218–231, 1990.

[33] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[34] S. van der Vegt. A concurrent bidirectional linear probing algorithm. In *15th Twente Student Conference on Information Technology*, 2011.

[35] S. van der Vegt and A. Laarman. A parallel compact hash table. In *Proceedings of the 7th International Conference on Mathematical and Engineering Methods in Computer Science*, 2011.

[36] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 37(12):1488–1505, 1988.