A Parallel Compact Hash Table

Steven van der Vegt, Alfons Laarman
s.vandervegt@student.utwente.nl a.w.laarman@ewi.utwente.nl
Formal Methods and Tools, University of Twente, The Netherlands

Abstract. We present the first parallel compact hash table algorithm. It delivers high performance and scalability due to its dynamic region-based locking scheme with only a fraction of the memory requirements of a regular hash table.

1 Introduction

During the last decade or so, we are witnessing a shift from ever faster sequential microprocessors towards multi-core processors. This shift is caused by physical limitations on the nanostructures inside the processor chip and is therefore irreversible. Most software systems, however, are still not equipped fully to benefit from the newly available parallelism.

Data structures, like *hash tables*, are crucial building blocks for these systems and many have been parallelized [4,6]. A hash table stores a subset of a large *universe* U of keys and provides the means to lookup individual keys in constant time. It uses a *hash function* to calculate an address h from the unique *key*. The entire key is then stored at its hash or home location in a table (an array of *buckets*): $T[h] \leftarrow key$. Because often $|U| \gg |T|$, multiple keys may have the same hash location. We can handle these so-called *collisions* by calculating alternate hash locations and searching for a key in the list of alternate locations, a process known as *probing*.

In the case that $|U| \leq |T|$, a hash table can be replaced with a *perfect hash function* and a bit array, saving considerable memory. The former ensures that no collisions can occur, hence we can simply turn "on" the bit at the home location of a key, to add it to the set. *Compact hashing* [3] generalizes this concept for the case $|U| > |T|$ by storing only the part of the key that was not used for addressing in T: the *remainder*. The complete key can now be reconstructed from the value in T and the home location of the key. If, due to collisions, the key is not stored at its home location, additional information is needed. Cleary [3] solved this problem with very little overhead by imposing an order on the keys in T and introducing three administration bits per bucket.

The bucket size b of Cleary compact hash tables is thus dependent on U and T as follows: $b = w - m + 3$, with the key size $w = \lceil log_2(|U|) \rceil$ and $m = \lceil log_2(|T|) \rceil$. Assuming that all the buckets in the table can be utilized, the compression ratio obtained is thus close to the information theoretical lower bound of storing a subset of U in a list T, where $b_{optimal} = w - m + 1$ [5]. Note that good compression ratios ($\frac{b}{w}$) are only obtained when m is significant with respect to w.

Problem description. Compact hashing has never been parallelized, even though it is ideally suited to be used inside more complex data structures, like *tree tables* [8] and *binary decision diagrams* (BDDs) [2]. Such structures maintain large tables with small pieces of constant-sized data, like pointers, yielding an ideal m and w for compact hashing. But even more interesting than obtaining some (constant-factor) memory reductions, is the ability to store more information in machine-sized words, for efficient parallelization depends crucially on memory alignment and low-level operations on word-sized memory locations [4,7].

Contributions. We present an efficient scheme to parallelize both the Cleary table and the order-preserving bidirectional linear probing (BLP) algorithm that it depends upon. The method is *lockless*, meaning that it does not use operating system locks, thereby providing the performance required for use in high-throughput environments, like in BDDs, and avoiding memory overhead.

Our algorithm guarantees read/write exclusion, but not on the lowest level of buckets, as in [4,7], nor on fixed-size regions in the table as in *region-based/striped* locking, but instead on the logical level of a *cluster*: a maximal subarray $T[i \dots j]$ such that $\forall x : i \leq x \leq j \implies T[x].occ$,where $T[x].occ$ denotes a filled bucket. We call this novel method: *dynamic region-based locking* (DRL).

2 Background

In the current section, we explain the Cleary table and the BLP algorithm it uses. Finally, we discuss some parallelization approaches that have been used before for hash tables and the issues that arise when applying them to the Cleary table.

For this discussion, the distinction between *open-addressing* and *chained* hash tables is an important one. With open addressing, the probing for alternate locations is done inside the existing table as is done in BLP and hence also in Cleary tables. While *chained* or *closed-addressing* hash tables resolve collisions by maintaining (concurrent) linked lists at each location in the table.

2.1 Bidirectional Linear Probing

Linear probing (LP) is the simplest form of open addressing: alternate hash locations in the table are calculated by adding one to the current location. While this probing technique provides good spatial locality, it is known for producing larger clusters, i.e., increasing the average probing distance [4].

BLP [1,9] mitigates the downside of LP, by enforcing a global order on the keys in the buckets using a *monotonic hash function*: if $k_1 < k_2$ then $hash(k_1) \leq hash(k_2)$. Therefore, the look-up of a key k boils down to: compare the k to the bucket at the home location h, if $T[h] > k$, probe left linearly ($h' \leftarrow h - 1$), until $T[h'] = k$. If k is not present in the table, the probe sequence stops at either an *empty* bucket, denoted by $\neg T[h'].occ$, or when $T[h'] < k$. If $T[h] < k$, do the reverse.

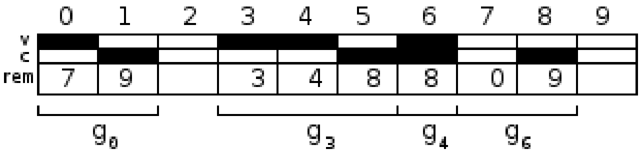
To maintain order during an insert of a key, the BLP algorithm needs to move part of a cluster to the left or the right in the table, thereby making space for the new key at the correct (in-order) location. This move is usually done with pair-wise swaps, starting from the empty bucket at one end of the cluster. Therefore, this is referred to as the *swapping* operation. For algorithms and a more detailed explanation, please refer to [9].

2.2 A Compact Hash Table Using the Cleary Algorithm

As explained in Sec. 1, Cleary's compact hash table [3] stores only the remainder of a key in T. With the use of the sorting property of the BLP algorithm and 3 additional *administration* bits per bucket, the home location h of the remainder can be reconstructed, even for colliding entries that are not stored at their home location. The *rem* function is the complement of the monotonic hashing function and calculates the remainder, e.g., $\mathit{rem}(x) = x\%10$ and $\mathit{hash}(x) = x/10$.[1] A *group* h is a sequence of successive remainders in T with the same home location h. All adjacent groups in T form a cluster, as defined in Sec. 1, which by definition is enclosed by empty buckets.

The first administration bit *occ* is used to indicate occupied buckets. The *virgin* bit is set on a bucket h to indicate the existence of the related group h in T. And finally, the *change* bit marks the last (right-most) remainder of a group, such that the next bucket is empty or the start of another group.

Fig. 1 shows the Cleary table with $|T| = 10$ that uses the example *hash* and *rem* functions from above. A group h is indicated with g_h. Statically, keys can be reconstructed by multiplying the group number by 10, and adding the remainder: $\mathit{key}(j) = \mathit{group}(T[j]) \times 10 + T[j] = \mathit{hash}^{-1}(\mathit{group}(T[j])) + T[j]$. For example, bucket 6 stores remainder 8 and $\mathit{group}(6) = 4$, therefore $\mathit{key}(6) = 4 \times 10 + 8 = 48$.



ph!]

Fig. 1. Example Cleary table with 10 buckets containing 8 remainders, 2 clusters and 4 groups, representing the keys: 7,9,33,34,38,48,60,69.

The algorithms maintain the following invariants [3]: the amount of *change* and *virgin* bits within a cluster is always equal, and, when a virgin bit is set on a bucket, this bucket is always occupied.

[1] To increase the performance of the hash function, it is common practice to apply an invertible randomization function to the key before hashing it [1,3,5]. Throughout this paper, we assume keys to be randomized.

```
1:  procedure VCOUNT-LEFT($j$)
2:      $c \leftarrow 0$                          ▷ count variable
3:      while $T[j].occ$ do
4:          $c \leftarrow c + T[j].virgin$
5:          $j \leftarrow j - 1$
6:      return $j, c$
7:  procedure FIND($k$)
8:      $j \leftarrow hash(k)$
9:      if $\neg T[j].virgin$ then
10:         return $NOT\_FOUND$                   ▷ false
11:     $(j, c) \leftarrow$ VCOUNT-LEFT($j$)
12:     $j \leftarrow j + 1$
13:     while $c \neq 0 \wedge T[j].occ$ do
14:         if $c = 1 \wedge T[j] = rem(k)$ then
15:             return $FOUND$                    ▷ true
16:         $c \leftarrow c - T[j].change$
17:         $j \leftarrow j + 1$
18:     return $NOT\_FOUND$                       ▷ false
```

```
Require: $(\exists i : \neg T[i].occ) \wedge \neg$FIND($k$)
1:  procedure PUT($k$)
2:      $h \leftarrow hash(k)$
3:      $(j, c) \leftarrow$ VCOUNT-LEFT($h$)
4:      $T[j] \leftarrow rem(k)$
5:      $T[j].occ \leftarrow 1$
6:      $T[j].change \leftarrow 0$
7:      while $c \neq 0$ do
8:          if $T[h].virgin \wedge c = 1 \wedge$
9:            $T[j+1] > rem(k)$ then
10:             return
11:         $c \leftarrow c - T[j+1].change$
12:         SWAP($T[j+1], T[j]$)
13:         $j \leftarrow j + 1$
14:     if $T[h].virgin$ then
15:         $T[j-1].change \leftarrow 0$
16:     $T[j].change \leftarrow 1$
17:     $T[h].virgin \leftarrow 1$
```

Alg. 1. Functions for finding (**a**) and inserting (**b**) a key in a Cleary table.

The FIND function in Alg. 1a makes use of these invariants as follows: it counts the number of *virgin* bits between the home location h and the left end of the cluster in c (see VCOUNT-LEFT). Since the last encountered *virgin* bit corresponds to the left-most group, the group h can now be located by counting c *change* bits to the right (l.13-17). The first iteration where $c = 1$ marks that start of group h. Hence, the algorithm starts comparing the remainders in $T[j]$ with $rem(k)$ at l.14, and returns *FOUND* when they are equal. Once c becomes 0 again, the group h did not contain the key, and *NOT_FOUND* is returned at l.18.

The PUT function in Alg. 1b inserts the remainder of k in the empty bucket left of the cluster around h at l.4-6 and swaps it in place at l.7-13 (SWAP only swaps the remainder and the *change* bit). In this case, *in place* means two things: within group h as guaranteed by l.7 and l.8, and sorted by remainder value as guaranteed by l.9. Furthermore, PUT guarantees the correct setting of the administration bits. First, the *occ* bit is always set for every inserted element at l.5. Also, before return, the *virgin* bit is always set for $T[h]$ (see l.8 and l.17).

To understand the correct setting of the *change* bits, we introduce an invariant: at l.8, $group(T[j+1]) \leq h$. Consequently, a return at l.10, means that the remainder is not swapped to the end of group h, therefore the *change* bits do not require updating. On the other hand, if the **while** loop terminates normally, the remainer is swapped to the end of group h, therefore the *change* bit needs to be set (l.16). If group h already existed ($T[h].virgin$ = true), the previous last remainder of the group needs to have its *change* bit unset (l.15).

We illustrate PUT with an example. Inserting the key 43 into the table of Fig. 1 gives a $h = hash(43) = 4$ and $rem(43) = 3$. Searching for the empty bucket left of the cluster at l.3, results in $j = 2$ and $c = 2$, since there are two *virgin* bits in buckets 3 and 4. The remainder is initially inserted in $T[2]$ (l.4-6). At l.12 the remainder in bucket 2 is swapped with bucket 3 (the *virgin* bit remains unchanged). These steps are repeated until j points to bucket 5. Then, at l.11 c becomes 1, indicating $group(T[j+1]) = h$. In the next iteration ($j' = j - 1$), the condition at l.8-9 holds, meaning that the remainder is at its correct location: at the start of g_4.

If instead, we were inserting the key 49, c would have become 0, ending the **while** loop with $j = 6$ (l.7), after swapping the remainder 9 to bucket 6. Because g_4 already existed, the previous *change* bit (now on $T[5]$) is unset by l.14-15. Finally, the *change* bit at bucket 6 is set by l.16.

To make groups grow symmetrically around their home locations and keep probing sequence shorter, it is important that the PUT function periodically also starts inserting remainders from the right of the cluster (not shown in the algorithm). Our experimental results confirm that a random choice between the two insert directions yields the same probe distances as reportedly obtained by the optimal replacement algorithms in [1].

2.3 Related Work on Parallel Hash Tables

In this subsection, we recapitulate some relevant, existing approaches to parallelize hash tables. With relevant, we mean parallel hash tables that can efficiently store smaller pieces of data (remember, from the introduction, that the key size w should be significant with respect to m for compact hashing to be effective). Furthermore, the scalability should be good for high-throughput systems like inside BDDs.

Many parallel hash table implementations are based on chaining. More advanced approaches even introduce more pointers per bucket, for example: *split-ordered lists* [6, Sec. 13.3], which: “move[s] the buckets among the [keys], instead of moving the [keys] among the buckets”. While these kind of hash tables lend themselves well for maintaining small sets in parallel settings like graphical user interfaces, they are less suited for our goals for two reasons: (1) the pointers require relatively much additional memory compared to the small bucket sizes that are so typical for compact hashing and (2) the pointers increase the *memory working set*, which is disastrous for scalability on modern computer systems with steep memory hierarchies [7,4].

Slightly more relevant to our cause is the use of operating system locks to make access to a hash table (chained or open addressing) concurrent. One lock can be used for the entire table, but this is hardly scalable. Alternatively, one lock can be used per bucket, but this uses too much memory (we measured 56 bytes for POSIX locking structures, this excludes any memory allocated by the constructor). A decent middle way is to use one lock for a group of buckets. The well-known *striped* hash table [6, Sec. 13.2.2], does this for chained tables. To employ the same idea for an open-addressing table, it does not make sense to ‘stripe’ the locks over the table buckets. Preferably, we group subsequent buckets into one region, so that only one lock needs to be taken for multiple probes. We dub this method “region-based locking” (RBL).

Lockless hash tables avoid the use of operating system locks entirely. Instead, atomic instructions are used to change the status of buckets ("locking" in parentheses). A lockless hash table (LHT) is presented in [7], based on ideas from [4]. It uses open addressing with LP and even modifies the probe sequence to loop over cache lines ("walking the line") to lower the memory working set and achieve higher scalability. For maximum scalability, only individual buckets are "locked" using one additional bit; the only memory overhead that is required.

None of the above-mentioned methods are suitable for *ordered hash tables*, like BLP and Cleary tables. First the regions in RBL are fixed, while the clusters in ordered tables can be at the boundary of a region. While this could be solved with more complicated locking mechanism, it would negatively affect the performance of RBL, which is already meager compared to the lockless approaches (see Sec. 4). The lockless approach, in turn, also fails for ordered hash tables since it is much harder to "lock" pairs of buckets that are swapped atomically. And even if it would be technically possible to efficiently perform an atomic pairwise swap, it would severely increase the amount of (expensive) atomic operations per insert (Sec. 3.2 discusses the complexity of the swapping operations).

In [9], we introduced a lockless algorithm for BLP that "locks" only the cluster during swapping operation. FIND operations do not require this exclusive access, for an ongoing PUT operation can only cause false negatives that can be mitigated by another *e*xclusive FIND operation. However, this method is not suitable for the Cleary table, since its FIND function is *probe-sensitive*, because it counts the *virgin* and *change* bits during probing. Therefore, it can cause false positives in case of ongoing swapping operations. The current paper is an answer to the future work of [9].

3 Dynamic Region-Based Locking

In the current section, we first present *dynamic region-based locking* (DRL): a locking strategy that is compatible with the access patterns of both the BLP algorithm with its swapping property and the Cleary table with its probe-sensitive lookup strategy. We limit our scope to a procedure that combines the FIND and PUT functions, described in the previous section, into the FIND-OR-PUT function, which searches the table for a key k and inserts k if not found. The reason for this choice is twofold: first, it covers all issues of parallelizing the individual operations, and second, the FIND-OR-PUT operation is sufficient to implement advanced tasks like *model checking* [7,8].

Additionally, in Sec. 3.2, we show that DRL only slightly increases the number of memory accesses for both BLP and PCT. From this and the limited number of atomic operations that it requires, we conclude that its scalability is likely as good as LHT's. We end with a correctness proof of DRL in Sec. 3.3.

3.1 Parallel FIND-OR-PUT Algorithm

We generalize the lockless BLP algorithm from [9] to accommodate Cleary compact hashing with its *probe-sensitive* FIND operation. It uses one extra bit field per bucket (*lock*) to provide light-weight mutual exclusion. This method has limited memory overhead and does not require a context switch and additional synchronization points like operating system locks.

The atomic functions TRY-LOCK and UNLOCK control this bit field and have the following specifications: TRY-LOCK requires an empty and unlocked bucket and guarantees an empty, locked bucket or otherwise fails. UNLOCK accepts multiple buckets and ensures all are unlocked upon return (each atomically, the multiple arguments are merely syntactic sugar). These functions can be implemented using the processor's CAS(a, b, c) operation, which updates a word-sized memory location at a with c atomically, if and only if the condition b holds for location a [6, Ch. 5.8]. CAS returns the initial value at location a, used to evaluate the condition.

Alg. 2 shows the dynamic locking scheme for the FIND-OR-PUT algorithm. First, at l.3, the algorithm tries a non-exclusive write using CAS, which succeeds if the home location h is empty and unlocked ($\neg lock \wedge \neg occ$). The success of the operation can be determined from the return value *old* of CAS (see l.4). If a lock or full bucket was detected, the algorithm is restarted at l.7.

From l.10 onwards, the algorithm tries to acquire exclusive access to the cluster around $T[h]$. Note that $T[h]$ is occupied. At l.10 and l.11, the first empty location left of and right of h are found in T. If both can be locked, the algorithm enters a local *critical section* (*CS*) after l.16, else it restarts at l.13 or l.16 (after releasing all taken locks). In the *CS*, the algorithm can now safely perform exclusive reads and exclusive writes on the cluster (l.17 and l.20).

DRL is suitable in combination with the FIND and PUT operations of both BLP and the Cleary table. If we are implementing the BLP algorithm using this locking scheme, then FIND at l.8 can perform a non-exclusive read (concurrent to any ongoing write operations). The possibility of a false negative is mitigated by an upcoming exclusive read at l.17. For the Cleary algorithm, however, the non-exclusive read needs to be dropped because the probe-sensitive lookup mechanism might yield a false positive due to ongoing swapping operations.

```
 1: procedure FIND-OR-PUT($k$)
 2:     $h \leftarrow hash(k)$      ▷ non-excl. write:
 3:     $old \leftarrow$ CAS($T[h], \neg lock \wedge \neg occ, k$)
 4:     if $\neg old.occ \wedge \neg old.lock$ then
 5:         return INSERTED
 6:     else if $old.lock$ then
 7:         return FIND-OR-PUT($k$)
 8:     if FIND($k$) then  ▷ non-excl. read
 9:         return FOUND
10:     $left \leftarrow$ CL-LEFT($h$)
11:     $right \leftarrow$ CL-RIGHT($h$)
12:     if $\neg$TRY-LOCK($T[left]$) then
13:         return FIND-OR-PUT($k$)
14:     if $\neg$TRY-LOCK($T[right]$) then
15:         UNLOCK($T[left]$)
16:         return FIND-OR-PUT($k$)
17:     if FIND($k$) then  ▷ exclusive read
18:         UNLOCK($T[left]$, $T[right]$)
19:         return FOUND
20:     PUT($k$)              ▷ exclusive write
21:     UNLOCK($T[left]$, $T[right]$)
22:     return INSERTED
```

Alg. 2. Concurrent bidirectional linear find-or-put algorithm

3.2 Complexity and Scalability

Two questions come to mind when studying the DRL: (1) What is the added complexity compared to the sequential BLP or Cleary algorithm? (2) What scalability can we expect from such an algorithm. Below, we discuss these matters.

For *ordered hash tables*, like BLP and Cleary tables, the cluster size L depends on the load factor α, as follows: $L = (\alpha - 1)^{-2} - 1$ [1], where $\alpha = n/|T|$ and n the number of inserted keys. Since DRL probes to the empty buckets at both ends of the cluster, it requires $(\alpha - 1)^{-2} + 1$ bucket accesses. When implementing the Cleary table using DRL, this is the complexity for the FIND-OR-PUT operation independent whether an insert occurred or not, because in both cases it "locks" the entire cluster. Note that we do not count the bucket accesses of the called FIND and the PUT operations, since, in theory, these could be done simultaneously by the CL-LEFT and CL-RIGHT operations. In practice, this seems unnecessary, because the cluster will be cache hot after locking it.

The sequential Cleary FIND and PUT algorithm have to probe to one end of the cluster to count the virgin and change bits, hence require $(\alpha - 1)^{-2}$ bucket accesses (again assuming that we can count both in one pass or that the second pass is cached and therefore insignificant). We conclude that Cleary+DRL (with one worker thread) is only twice as slow as the original Cleary algorithm.

For BLP+DRL the story changes, but the outcome is the same. The sequential BLP algorithm does not have to probe to the end of the cluster and is empirically shown to be much faster than LP [1]. However, DRL+BLP is correct with non-exclusive reads as long as an unsuccessful FIND operation is followed by an exclusive FIND to mitigate false negatives, as is done in Alg. 2. But false negatives are rare, so again the parallel FIND operation is not much slower than the sequential one. The same holds for the PUT operation, since the sequential version on average needs to swap half of an entire cluster and the parallel version "locks" the whole cluster.

Scalability of DRL can be argued to come from three causes: first, the complexity (in memory access) of the parallel algorithm is the same the sequential versions, as shown above, second, the number of (expensive) atomic operations used is low, DRL uses two at most, and third, the memory accesses are all consecutive. We analyze the third cause in some more detail.

To mitigate the effect of slow memories, caching is important for modern multi-core systems. Each memory access causes a fixed region of memory, known as a cache line, to be loaded into the CPU's cache. If it is written to, the entire line is invalidated and has to be reload on all cores that use it; an operation which is several orders of magnitude more expensive than other operations using in-cache data. We have shown before that highly scalable hashing algorithms can be obtained by lowering the number of cache lines that are accessed: the *memory working set* [7].

The open-addressing tables discussed in this paper exhibit only consecutive memory accesses. And while it seems that the amount of buckets probed in the Cleary algorithm is high, typically few cache lines are accessed. For example, there are 26 bucket accesses on average for $\alpha = 0.8$, while on average only

$\lceil 26/64 \rceil + 26/64 = 1.41$ cache lines are accessed, assuming a bucket size of 1 byte and a cache line size of 64 byte. When α grows to 0.85, we get 1.71 cache line accesses on average, and when $\alpha = .9$, 3.59 accesses. Note finally that with buckets of 1 byte, the cleary algorithm can store keys of more than 32 bit for large tables, e.g, if $m = 28$, then $w = b + m - 3 = 8 + 28 - 3 = 33$, while non-compacting hash table requires five bytes per bucket to store as many data. In conclusion, we can expect Cleary+DRL to perform and scale good until load factors of 0.8 and competitive performance to that of [7].

3.3 Proof of Correctness

To prove correctness, we show that Alg. 2 is *linearizable*, i.e., its effects appear instantaneously to the rest of the system [6, Ch. 3.6]. Here, we do this in a constructive way: first, we construct all possible local schedules that Alg. 2 allows, then we show by contradiction that any interleaving of the schedules of two workers always respects a certain critical section (CS) of the algorithm, and finally, we generalize this for more workers. From the fact that CS is the only place where writes occur, we can conclude linearizability.[2] We assume that all lines in the code can be executed as atomic steps.

If the home location of a key k is empty, correctness follows from the properties of the atomic CAS operation at l.3. For every other table accesses (l.17 and l.20), we prove that never two workers can be in their CS for the same cluster.

The '$\rightarrow$' operator is used to denote the *happens-before relation* between those steps [6]. For example, '$\text{CL-RIGHT}_i(\overline{x}) \rightarrow \text{TRY-LOCK}_i(x)$' means that a Worker i always first executes CL-RIGHT writing to the variable x (l.11), and subsequently calls TRY-LOCK using (reading) the variable x. We omit the subscript i, if it is clear from the context which worker we are talking about. We concern ourselves with the following local happens-before order: $\text{CAS}(h) \rightsquigarrow \text{CL-LEFT}(\overline{l}) \rightarrow \text{CL-RIGHT}(\overline{r}) \rightarrow \text{TRY-LOCK}(l) \rightsquigarrow \text{TRY-LOCK}(r) \rightsquigarrow (occ(l) \oplus occ(r))$, where $occ(x)$ signifies a fill of a bucket ($T[x].occ \leftarrow 1$) and $\rightsquigarrow$ indicates a happens-before relation dependent on a condition. Depending on the replacement end (left or right), PUT fills one of the buckets at the end of the cluster, hence the exclusive-or: $\oplus$. Furthermore, we write l_i, r_i and h_i for: the *left* variable, the *right* variable and the home-location $h_i = hash(k)$, all local to a Worker i.

Lemma 1. *Alg. 2 ensures that when two workers try to enter their CS for the same cluster, then:* $l_i = l_j \vee r_i = l_j \vee l_i = r_j \vee r_i = r_j$.

Proof. Assume Worker W_i is in its CS, and Worker W_j is about to enter the CS for the same cluster. Since W_i is in its CS, we have $T[l_i].lock$ and $T[r_i].lock$. W_i is going to perform the step $occ(l_i)$ or $occ(r_i)$. Note that these operations might influence the clusters, as two clusters separated by only one empty bucket, may become one upon filling the bucket.

[2] For completeness sake, we should also mention that we only allow for false positives to occur in non-exclusive reads and that unsuccessful non-exclusive reads are always followed by a read operation in the CS, i.e., an exclusive read.

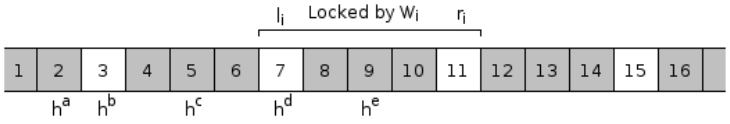


Fig. 2. Several clusters and empty positions. The cluster 8-10 is locked by worker W_i. Location marked with h^a to h^e potential home locations for worker W_j.

Worker W_j has yet to enter its *CS*, executing the steps: CAS(h_j) → CL-LEFT($\bar{l}_j$) → CL-RIGHT($\bar{r}_j$). With a generalizable example, Fig. 2 illustrates five non-trivial cases that we consider, where W_j starts with a h_j respective to the cluster l_i, r_i. Clusters in T are colored gray and we assume that they are separated by one empty bucket (white), because more empty buckets makes the resulting cases only more trivial. There are several representative home-locations marked with h^a to h^e (e.g., choosing a different location within the same cluster leaves the results of the CL-LEFT and CL-RIGHT operations unaffected). Locations on the right of r_i follow from symmetry. Below, we consider the outcome of all the cases for h_j. We use the fact that there are no empty buckets between l_j and r_j.

$h_j = h^a$: Because $T[h_j].occ$, CAS(h_j) fails. W_j performs the steps CL-LEFT($\bar{l}_j$) → CL-RIGHT($\bar{r}_j$). Since $l_j = 1 < r_j = 3 < l_i$, Lemma 1 is vacuously true.

$h_j = h^b$: This location is unoccupied and not locked, so the CAS(h_j) succeeds and the algorithm returns never reaching *CS*, making Lemma 1 vacuously true.

$h_j = h^c$: This location is occupied so CAS(h_j) fails. Next, the step CL-LEFT($\bar{l}_j$) results in $l_j = 3$. The result r_j of CL-RIGHT is dependent on the state of W_i. If W_i has not already performed any *occ* or did perform *occ*(11), then $r_j = 7$. If W_i has executed *occ*(7), then $r_j = 11$. So, $r_j = 7 = l_i \vee r_j = 11 = r_i$.

$h_j = h^d$: The success of the CAS(h_j) depends on the state of W_i. If W_i has not performed any steps, then CAS(h_j) restarts the algorithm at l.7. If W_i has performed *occ*(7), then W_j continues with CL-LEFT($\bar{l}_j$) and CL-RIGHT($\bar{r}_j$), resulting in $l_j = 3, r_j = 11 = r_i$. If W_i has performed step *occ*(11), then $l_j = 7 = l_i, r_j = 15$.

$h_j = h^e$: Since h^e is occupied, CAS(h_j) fails again. W_j continues with the CL-LEFT($\bar{l}_j$) and CL-RIGHT($\bar{r}_j$). The result depends on if W_i has executed *occ*(7) or *occ*(11). We distinguish five interleavings:
1: CL-LEFT($\bar{l}_j$) → CL-RIGHT($\bar{r}_j$) → $(occ_i(7) \oplus occ_i(11)) \Rightarrow l_j = 7, r_j = 11 = r_i$
2: CL-LEFT(l_j) → $occ_i(7)$ → CL-RIGHT(r_j) $\Rightarrow l_j = 7 = l_i, r_j = 11 = r_i$
3: CL-LEFT(l_j) → $occ_i(11)$ → CL-RIGHT(r_j) $\Rightarrow l_j = 7 = l_i, r_j = 15$
4: $occ_i(7)$ → CL-LEFT(l_j) → CL-RIGHT(r_j) $\Rightarrow l_j = 3, r_j = 11 = r_i$
5: $occ_i(11)$ → CL-LEFT(l_j) → CL-RIGHT(r_j) $\Rightarrow l_j = 7 = l_i, r_j = 15$

Thus, under the above assumption: $l_i = l_j \vee r_i = l_j \vee l_i = r_j \vee r_i = r_j$. □

Theorem 1. *No two workers can be in their CS at the same time and work on the same cluster such that* $l_i \leq l_j \leq r_i \vee l_i \leq r_j \leq r_i \vee (l_j \leq l_i \wedge r_j \geq r_i)$.

Proof. By contradiction, assume the opposite: both W_i and W_j reach their CS and $l_i \leq l_j \leq r_i \vee l_i \leq r_j \leq r_i \vee (l_j \leq l_i \wedge r_j \geq r_i)$. Without loss of generality because of symmetry, we assume again W_i to have entered its CS first. The steps for W_j to arrive in its CS are:
$\textsc{cas}(h_j) \rightarrow \textsc{cl-left}(l_j) \rightarrow \textsc{cl-right}(r_j) \rightarrow \textsc{try-lock}(l_j) \rightarrow \textsc{try-lock}(r_j)$.
The remaining step for W_i is: $occ(l_i) \oplus occ(r_i)$
W_i hash performed $\textsc{try-lock}(l_i) \rightarrow \textsc{try-lock}(r_i)$, thus we have $T[l_i].lock \wedge T[r_i].lock$. According to Lemma 1 that at least one of the locations l_j and r_j equals either l_i or r_i. Therefore, W_j will always fail with either $\textsc{try-lock}(l_j)$ or $\textsc{try-lock}(r_i)$. This conclusively proves mutual exclusion for two workers. Since additional workers cannot influence W_j in such a way that Lemma 1 is invalidated, Theorem 1 also holds for $N > 2$ workers. □

Absence of deadlocks (infinite restarts at l.7, l.13 and l.16), follows from the fact that all "locks" are always released before a restart or a return. Furthermore, we have absence of livelocks, because workers first "lock" the left side of a cluster. The one which locks the right side first, wins. With a fair scheduler the algorithm is also starvation-free, because each worker eventually finished its CS in a finite number of steps. From this, we conclude that Alg. 2 is linearizable.

4 Experiments

In the current section, we show an empirical evaluation of the Parallel Cleary Table (PCT), i.e. Cleary+DRL, by comparing its absolute performance and scalability with that of BLP+DRL, LHT and RBL. In our experiments, several parameters have been fixed as follows: $m = 28$, $b = 16$ for PCT, while for the non-compacting tables $b = 64$, and finally $\alpha = 0.9$. These parameters reflect best the goals we had in mind for this work, since all tables can store pointers larger than 32 bits. Furthermore, the load factor and bucket size for PCT is higher than the values discussed in Sec. 3.2, creating a healthy bias against this algorithm. Additionally, we investigated the influence of different load factors on all tables.

We used the following benchmark setup. All tables were implemented in the C language using pthreads.[3] For RBL, we determined the optimal size of the regions by finding the size that yielded the lowest parallel runtime. For table of 2^{28} buckets, this turned out to be 2^{13}. The benchmarks were run on Linux servers with 4 AMD Opteron(tm) 8356 CPUs (16 cores total) and 64GB memory. The maximum key size w that all tables can store in our configuration is 40: for PCT we have $w = b + m - 4 = 16 + 28 - 4 = 40$, and for BLP, LHT and RBL we have $w = 64 - 2 = 62$ (2 for the *lock* and *occ* bit). Therefore, we fed the tables with 40 bit keys, generated with a pseudo random number generator.

Table 4 gives the runtimes of all hash tables for different read/write ratios and load factor of 90%. Beside the runtimes with 1, 2, 4, 8, and 16 cores (T_N for $N \in \{1, 2, 4, 8, 16\}$), we included the runtimes of the sequential versions of the algorithms T_{seq}, i.e., the algorithm run without any locks and atomic

[3] Available at: http://fmt.cs.utwente.nl/tools/ltsmin/memics-2011

Table 1. Runtimes of BLP, RBL, LHT and PCT with r/w ratios 0:1, 3:1 and 9:1.

Alg.	LHT			RBL			BLP			PCT		
r/w ratio	0:1	0:3	0:9	0:1	0:3	0:9	0:1	0:3	0:9	0:1	0:3	0:9
T_{seq}	77.5	242.4	569.2	76.7	239.9	563.2	71.8	279.1	676.0	54.5	368.9	1050.
T_1	81.6	255.2	599.2	145.9	565.4	1404.	97.5	302.0	726.3	77.3	565.9	1543.
T_2	51.6	157.6	371.0	85.0	327.6	813.4	60.8	188.8	443.9	44.4	317.7	863.9
T_4	26.5	77.9	184.0	46.2	170.2	424.9	31.3	94.0	219.1	23.4	159.7	431.9
T_8	13.9	39.6	92.9	24.0	89.4	219.2	16.5	47.8	110.3	11.5	79.7	216.0
T_{16}	7.7	21.1	48.8	13.5	48.6	120.5	9.4	25.5	57.2	6.0	41.6	112.9

instructions. From this, we can deduce the overhead from the parallelization. Comparing the runs with a r/w ratio of 0:1, we see that the sequential variants have more or less the same runtime (PCT is slightly faster, due to its compacter table). Only the lockless algorithms show little overhead when we compare T_{seq} to T_1, while DRL shows that the POSIX mutexes slow the algorithm down by a factor of two. The same trend is reflected in the values for T_N with $N > 1$.

If we now focus our attention to the higher r/w ratios, we see that reads are much more expensive for PCT. This was expected, since non-exclusive reads in DRL are not allowed for PCT as explained in the previous section. To investigate the influence of the r/w ratio, we plotted the absolute speedups ($S_N = T_{seq}/T_N$) of the presented runs in Fig. 3. The lightweight locking mechanism of DRL delivers good scalability for PCT and BLP, almost matching those of LHT. While PCT speedups are insensitive to the r/w ratio, since the algorithm always performs the same locking steps for both read and write operations, BLP shows much better speedups for higher r/w ratios. Finally, we see that RBL is no competition to the lockless algorithms.

To investigate the effects of the load factor, we measured the 16-core runtimes of all algorithms for different load factors. To obtain different load factors we modified the number of keys inserted and not the hash table size, therefore we plotted the normalized runtimes T^{norm} in Fig. 4 ($T^{norm} = T/\alpha$, where $\alpha = n/|T|$ is the load factor and n the number of keys inserted). Due to the open-addressing nature of the hash tables presented here, the asymptotic behavior is expected for α close to 100% (the probe sequences grow larger as the table fills up). However, this effect is more pronounced for PCT, again because of the read-write exclusion, and for RBL, because more locks have to be taken once the probe distance grows.

5 Discussion and Conclusions

We have introduced DRL: an efficient lockless mechanism to parallelize BLP and Cleary compact hash tables efficiently. We have shown, analytically and empirically, that these Parallel Cleary Tables (PCT) scale well up to load factors of at least 80%. This is acceptable, since the compression ratio, obtained by compact hashing, can be far below this value.

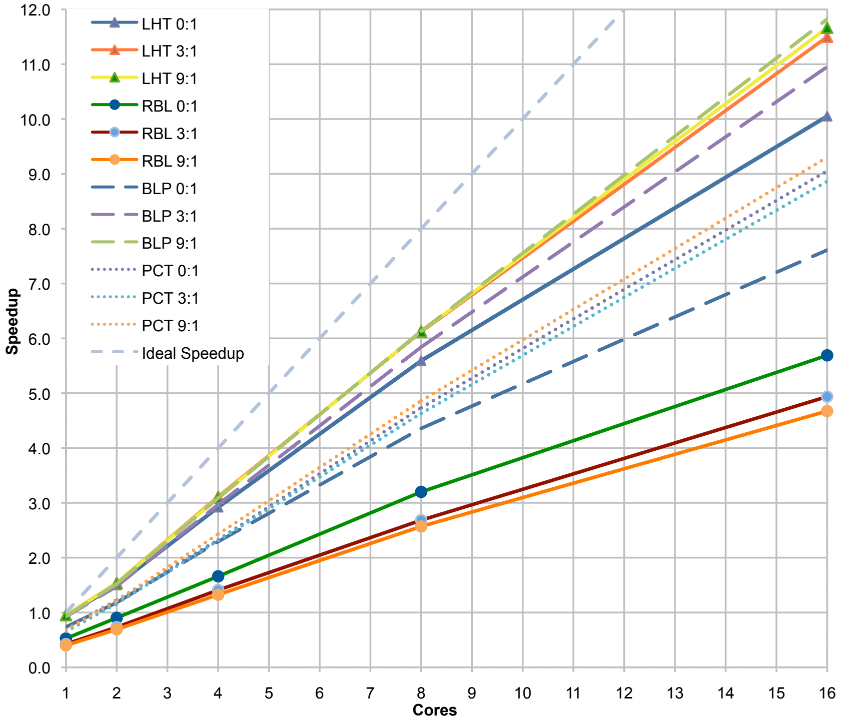


Fig. 3. Speedups of BLP, RBL, LHT and PCT with r/w ratios 0:1, 3:1 and 9:1.

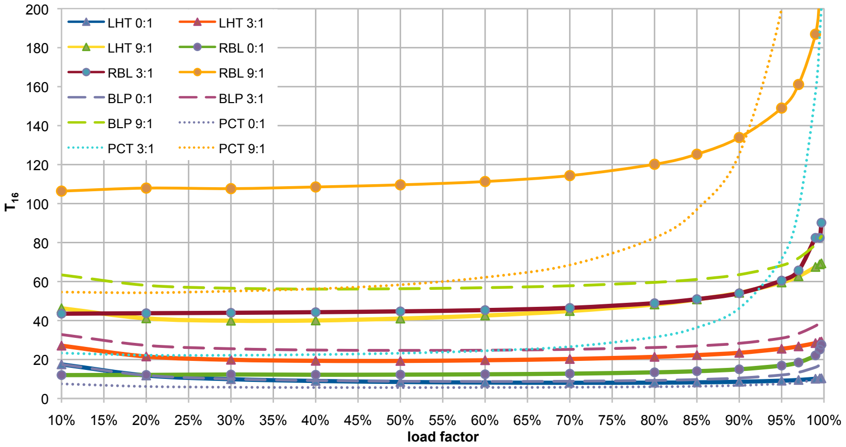


Fig. 4. 16-core runtimes of BLP, RBL, LHT and PCT.

With experiments, we also compared both parallel ordered hash tables (PCT and BLP) with a state-of-art lockless hash table (LHT) and a region-based locking table that uses operating system locks (RBL). We found that PCT and BLP can compete with LHT. On the other hand, RBL scales worse than the other lockless tables. We finally showed that PCT comes with higher costs for FIND operations. However, this also holds for the sequential algorithm because it has to probe to the end of the cluster as the analysis showed and as is reflected by the good speedups that PCT still exhibits with high r/w ratios.

While we concentrated in this work on a parallel FIND-OR-PUT algorithm, we think that other operations, like individual FIND, PUT and DELETE operation, can be implemented with minor modifications.

In future work, we would like to answer the following questions: Could DRL be implemented with locking only one side of the cluster and the home location? Could PCT be implemented with non-exclusive reads? The former could further improve the scalability of DRL, while the latter could transfer the performance figures of parallel BLP to those of PCT.

Acknowledgements

We thank Jaco van de Pol for providing useful comments on this work.

References

1. O. Amble and D. E. Knuth. Ordered Hash Tables. *The Computer Journal*, 17(2):135–142, 1974.
2. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
3. J.G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computers*, C-33(9):828 –834, September 1984.
4. C. Click. A Lock-Free Hash Table. Talk at JavaOne 2007, http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf, 2007.
5. Jaco Geldenhuys and Antti Valmari. A nearly memory-optimal data structure for sets and mappings. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 136–150, Berlin, Heidelberg, 2003. Springer-Verlag.
6. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. M. Kaufmann, March 2008.
7. A.W. Laarman, J.C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In N. Sharygina and R. Bloem, editors, *FMCAD 2010*, pages 247–255, USA, October 2010. IEEE Computer Society.
8. A.W. Laarman, J.C. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In A. Groce and M. Musuvathi, editors, *SPIN 2011*, LNCS, pages 38–56, London, July 2011. Springer.
9. S. van der Vegt. A Concurrent Bidirectional Linear Probing Algorithm. In C. Heijnen and H. Koppelman, editors, *15th Twente Student Conference on Information Technology, Enschede, The Netherlands*, volume 15 of *TSConIT*, pages 269–276, Enschede, June 2011. Twente University Press. http://referaat.cs.utwente.nl/TSConIT/download.php?id=981.