A Concurrent Bidirectional Linear Probing Algorithm
Towards a Concurrent Compact Hash Table

Steven van der Vegt
University of Twente
The Netherlands
s.vandervegt@student.utwente.nl

ABSTRACT

Hash tables are commonly used data structures for representing subsets of a large key universe. In these tables, the complete key is stored in order to resolve the item in case of a collision. Since the 1980's, there are known techniques to reduce the used memory by only storing a part of the original key. However, there are no known concurrent implementations of these so-called compact hash tables. Nowadays, the way to obtain the most performance gain, is to utilize the multiple CPU cores many machines are equipped with. This paper describes the parallelization of the bidirectional linear probing algorithm developed by Amble & Knuth, with a minimal memory overhead. We proved correct and implemented the algorithm and show, with results from experiments, that it scales well.

1. INTRODUCTION

To store sets of data in memory, many data structures have been developed in the last decades. An array is one of the simplest of those structures and built up from a list of adjacent items in memory. A fixed-size item can be located in memory by multiplying its index by the size of an item and adding this to the base address of the array. This results in a look-up time of *O(1)* and a total memory usage of $N * sizeof(item)$ for an array size of N items. Although this data structure is fast and memory efficient it may not suit all needs. In situations where not for every index an item exists, memory will be wasted. Another problem is when the universe of possible indexes exceeds the maximum possible numbers of items in memory. For example, a list of 7 characters (8-bits per character) is used as a index on a 32-bits computer system. The amount of possible keys $(2^8)^7 > 2^{32}$ is much larger than the possible addressable items. Both problems can be solved by the use of a hash function which maps the large universe of the item index (from now on called a *key*) to the array index: $hash(K) \rightarrow i$, where i is a index such that $0 \leq i < N$. An example to map a 32 bits key to an hash table with the size of 2^8 elements is dividing the key by 2^{24}. This data structure is called a hash table.

Because the universe of the key is larger than that of the array index, many different keys are mapped onto the same array index. The occasion that during an insert an index is already occupied is called a collision, and can be solved using chaining or open addressing. Chaining involves creating a linked list for every bucket, where a collisions has occurred. Every time an item collides, an extra bucket is added to the linked list. Open addressing uses the unused buckets in the array to solve the collisions. The search for the correct item is called probing. To verify whether a collision has occurred, the original key is stored alongside the item. Thus, a standard hash table is formed of so called *buckets* which contain the a key and a value.

The amount of memory needed to store n buckets of size w (in bits) in an open addressing hash table is nw. This way of storing is called *linear storage.* According to the information theory, the minimum size of a data structure which holds n items of size w is $n(w - \log_2 n + 1)$ [4]. This is considerably less than the linear method for a certain n and w. Based on this theory, compact hash tables are developed which try to approach this theoretical limit. A data structure developed by John G. Cleary in 1984 [2] is such a compact hash table. Instead of storing the whole key, this compact table only stores $w - \log_2 n + 3$ bits. One bit is used to verify if a bucket is occupied, the other two are used to reconstruct the home location of the element. This can save a lot of memory. If, for example, we have a w of 10 bits, and a hash table with $n = 32$ buckets there are $2^{10-5} = 32$ keys mapped to one bucket. So, every bucket should only store the last $5 + 3$ bits of the key to uniquely distinguish itself, instead of the 10 bits which is stored in conventional hash tables. For large hash tables, the memory reduction can be significant.

We will limit the scope of this research is to a special type of hash table designed for sets that only grow monotonically. These type of tables can be used in, for example, planning problems and model checkers. Because the table only grows it has several properties: only inserts, and lookups are performed on the table. The table is searched for an existing key, if this key is not found it is inserted. We will call this action find-or-put. Only the key is needed, thus no values are stored in the buckets. No resizing is required because the table takes up all available memory. We use open addressing, which means the buckets are stored in an array and we do not make use of any pointers or linked lists. This saves memory, especially on 64-bit systems.

The solutions these special tables are used in are often CPU intensive. Since the enormous increase of CPU clock speed is coming to a halt because of physical limits, CPU manufacturers compensate this by adding more cores to a CPU. For an algorithm to benefit from this increasing CPU power is must be parallelized. Many research is going into the parallelization of existing data structures like hash tables, and recently Cliff Click [3] parallelized an open ad-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
15th Twente Student Conference on IT June 20st, 2011, Enschede, The Netherlands.
Copyright 2011, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

dressing hash table using the compare and swap instruction (CAS). The use of the CAS instruction resulted in a highly scalable and elegant solution which is an inspiration for this research.

This paper introduces a bidirectional linear probing find-or-put algorithm using a locking mechanism with little memory overhead. This type of hash table is the base of the compact Cleary table [2]. We provide a proof of its correctness, and show that it scales well. With this algorithm we believe to become one step closer to finding a concurrent Cleary table.

Next in Section 2, where we describe the sequential search and insert algorithms of a bidirectional linear probing table. Then in Section 2.3 we combine these to a sequential find-or-put. In Section 3, we introduce our parallel version of this find-or-put algorithm. We describe its logic and in Section 4, we prove its correctness. In Section 5, we describe the experiments with the table and their results. Finally, in Section 5 we give some recommendations how, with the result of this paper, the Cleary algorithm can be parallelized.

2. BIDIRECTIONAL LINEAR PROBING

Linear probing is a scheme for finding a location by sequentially searching the table. In case of a collision, the new probing location is calculated by adding an *interval* value to the current location. When the interval is set to 1 this probing technique provides good memory caching through good locality of reference. Bidirectional linear probing lets the algorithm probe up or down, depending on the value of the key in the initial bucket. The index of this initial bucket is found by the hash function. Bidirectional linear probing requires a hash function which guarantees that for keys k_1 and k_2, if $k_1 < k_2$ then $hash(k_1) < hash(k_2)$. Thus, all the keys in the table are ordered. At the start of the probing algorithm the key from the initial bucket is compared to the key under search. If the key in the initial bucket is greater than the searched key, the algorithm starts probing down the table, otherwise it goes up.

2.1 The Data Structure

We will use an array of M elements numbered 0 to $M-1$. We call this array our hash table. Every position can contain a key K_i for a certain i so that $0 \leq i < M$. The search problem consists of taking a given argument K and determine if there exists an i such that $K = K_i$. If U is the universe of all possible keys of n-bits we will define a function for each $K \in U : hash(K) \Rightarrow \textit{the hash address of } k$. The function is constrained so that $0 \leq h < M$ and will be used as an initial index for the probing process. Some elements in the table will be filled while others may be empty. To indicate that status, every element has an *occupied* property.

2.2 Algorithms

The first algorithm, as shown in Algorithm 1, is a search algorithm using a bidirectional linear probing technique, and developed by Amble & Knuth[1]. The elements in the table are always in order, so an element can be found by either probing left or right depending on the value of the initial hash-location. The amount of probes depends on how far the element is away from its original hashing location. The closer these elements are to this location, the more balanced the table is, and the faster a search can be performed. The second algorithm, as shown in Algorithm 2, is an insert algorithm based on the insert algorithm of Amble & Knuth, which ensures the conditions needed for

Algorithm 1 Search for an element, given key K, using the method of bidirectional linear probing

```
procedure BOOL SEARCH(K)
    j ← hash(K)
    if table[j] = K then
        return true
    if table[j] < K then
        while table[j + 1].occupied ∧ table[j + 1] < K do
            j ← j + 1
    else if table[j] > K then
        while table[j − 1].occupied ∧ table[j − 1] > K do
            j ← j − 1
    return table[j] = K
```

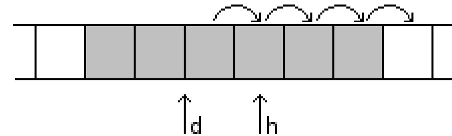


Figure 1. An example of the swap procedure during an insert when *table*[h] > K. All occupied elements right of d are swapped one position to the right. Occupied elements are marked with gray. The home location of a key K is marked with h. Its destination is marked with d.

the bidirectional linear probing. The algorithm searches either up or down for a empty element. This direction depends on the the value of the initial location of j (see Figure 1). If this element is greater than the searched key, the whole cluster is too much to the left, if the value is less than the key, the cluster is too much to the right. To correct this, the algorithm will shift the elements one position into the correct direction by swapping an empty element into the correct location (such that the all elements stay in order). The new K will then be inserted into this the empty position.

Our algorithm is a variation of the original algorithm. The original algorithm always probes right for a empty element. To prevent an unbalanced table, it checks if the values are not too far away from their home location. If this is the case an extra swap step takes place which shifts the whole cluster one position to the left. Our algorithm is simpler and results from experiments shows that it is comparable efficient.

We will now describe Algorithm 2. The algorithm inserts an item into the table. It requires that the table is not full. First create the hash of the key on line 2, Then check if that location is empty at line 3. If this is the case insert the key on that location. Otherwise, depending on the value of the location probe left or write for an empty location. If the empty location is found, check if it is in the correct position. If not, swap it with it's appropriate neighbor until it is at the correct position(lines 9 to 11). The final step is inserting the new key into the empty location on line 18. This algorithm will create a table with the property: $\forall i : K_i < K_{i+1}$. Since the elements can be swapped beyond the array boundaries some extra breathing space is needed at $j < 0$ and $j > M-1$.

Suppose $M = 10$ and there are $N = 8$ keys:
145 293 297 458 553 526 841 931
The function *hash()* returns the first digit: $hash(245) = 2$.

Algorithm 2 Insertion of key K which assures the condition for bidirectional linear probing

Require: table is not full
1: **procedure** INSERT(K) ▷ insert key K
2: $j \leftarrow hash(K)$
3: **if** $\neg table[j].occupied$ **then**
4: $table[j] \leftarrow K$
5: **return**
6: **if** $table[j] > K$ **then**
7: **while** $table[j].occupied$ **do**
8: $j \leftarrow j + 1$
9: **while** $table[j-1] > K \wedge \neg table[j-1].occupied$ **do**
10: $table[j] \leftarrow table[j-1]$
11: $j \leftarrow j - 1$
12: **else if** $table[j] < K$ **then**
13: **while** $table[j].occupied$ **do**
14: $j \leftarrow j - 1$
15: **while** $table[j+1] < K \wedge \neg table[j+1].occupied$ **do**
16: $table[j] \leftarrow table[j+1]$
17: $j \leftarrow j + 1$
18: $table[j] \leftarrow K$

Thus with a 3 digits key the hash function maps 1000 keys onto a table of size 10. The first element 145 has a hash value of 1. The corresponding position T_1 is free and the value can be inserted there. The second value to be inserted has a hash value of 2 and its corresponding position T_2 is also free in the table. The third value gets a hash value of 2 and collides with the previously inserted element. The algorithm compares the value of the home location, which is 293 with the current key. The new key is bigger so it probes to the left for an empty location. This location is found at T_0 and swapped into place at T_2. Now the final step is to insert 297 at location T_2. The rest of the values are inserted in a similar way.

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
145	293	297	0	458	526	553	0	841	931

2.3 A find-or-put Algorithm

Separate search and insert methods in a parallel environment are not easily composable. No duplicate items are allowed in our hash table, however, with a separate search and insert method this cannot easily be ensured. For example: worker a searches for a key, does not find it and inserts it. But between the return of the search function en before the call to the insert function, another worker b inserts the key. Now the table contains the element twice. Therefore, we distilled a find-or-put algorithm from the above algorithms which is shown in Algorithm 3.

3. PARALLEL FIND-OR-PUT ALGORITHM

In this section, we present the parallel find-or-put algorithm which uses a locking strategy. The reason for locking is ensuring specific right for the data at a time. This is important because it is very hard to make assumptions about the state of the data if many workers can read and write to the table at once. The easiest way of locking the data is locking the whole table. However, this is not an efficient method since only one worker at a time can access the data. Therefore we use the method of region-based locking. This fine grained method involves locking a region of the table which is kept as small as possible. Normal region-based locking systems use static boundaries, but because of the variable clusters, we introduce a method of dynamic-region locking. For the boundaries of a region we choose the closest empty elements left and right to the home location. Once a worker has obtained these locks, no other worker can write to this region. For the locks we use a bit field per bucket. This method has limited memory overhead and does not require a context switch like operating system locks such as mutexes.

Algorithm 3 Search for a key K. If the element is not in the table, insert it.

Require: table is not full
1: **procedure** FIND-OR-PUT(K)
2: $j \leftarrow hash(K)$
3: **if** $\neg table[j].occupied$ **then**
4: $table[j] \leftarrow K$
5: **return** *INSERTED*
6: **if** $table[j] = K$ **then**
7: **return** *FOUND*
8: **if** $table[j] < K$ **then**
9: **while** $table[j].occupied \wedge table[j] < K$ **do**
10: $j \leftarrow j - 1$
11: **if** $table[j] = K$ **then return** *FOUND*
12: $j \leftarrow hash(K)$
13: **while** $table[j].occupied$ **do**
14: $j \leftarrow j + 1$
15: **while** $table[j-1].occupied \wedge table[j-1] > K$ **do**
16: $table[j] = table[j-1]$
17: $j \leftarrow j - 1$
18: $table[j] \leftarrow K$
19: **return** *INSERTED*
20: **else**
21: Same, only in the other direction

To parallelize the algorithm we make use of several helper functions. The function CAS as specified in Algorithm 4 updates an element to *newval* atomically if the *testval* is equal to *memory_location*. On most platforms the *CAS* operation is implemented in hardware.

Algorithm 4 Specification of the atomic function CAS.

procedure CAS(*memory_location*, *testval*, *newval*)
oldval ← *memory_location*
if *memory_location* = *testval* **then**
memory_location ← *newval*
return *oldval*

The atomic functions *try_lock* and *unlock* have the following specifications: *try_lock* requires an empty and unlocked bucket and guarantees a locked element or otherwise fails. *try_lock* needs to be implemented with an atomic operation such as CAS. Unlock does not require anything but ensures the element is unlocked after it returns.

The parallel find-or-put algorithm is given in Algorithm 5. A worker starts with determining the home location h of K. With the atomic *CAS* operation the worker tests if the this location is still free, and if so, it inserts K. If the *CAS* fails we can conclude that either the element is locked or it is occupied. In the first case, the worker starts over to try again later. In the second case, the worker continues a search for K. Depending on the contents of the home location, the search for K goes searching left or right in the table. The function FIND-FROM determines the correct location for key K. If, at this location K is

not found, the worker searches for the elements *t_left* and *t_right* which are empty and as close as possible to h such that *t_left* $< h <$ *t_right*. When these elements are found the worker tries to lock them. If these locks succeed the algorithm is in its critical section CS at line 26. In its critical section, this worker is the only one which has write rights on the cluster between its *t_left* and *t_right* elements. So, there is no other worker which can insert elements in this cluster. Between the last check if K was in the table, and the moment the algorithm entered its CS, another worker could have inserted K. So there is a last check for the existence of K in the table. If K is found, the worker returns with a FOUND. Otherwise K has to be inserted. It does this in the same way as the sequential find-or-put in Algorithm 3. After the empty element is swapped in place, K is inserted and the algorithm leaves its CS by unlocking the boundaries at line 41 and returns with INSERTED.

4. CORRECTNESS PROOFS

In this section, we will prove the algorithm correct by researching the safety properties: consistency and atomicity. Using a proof with contradiction, we show that there can be never two workers in their critical section working on the same cluster in the hash table. With the prove we show that the algorithm is free from deadlocks and starvation.

4.1 Consistency and Atomicity

The use of this type of hash table can be compared to a database. Its most important safety properties are properties which are also important for databases: consistency and atomicity. When multiple workers perform actions on the hash table, we need the guarantee that the table is always is a consistent state, like there cannot be two of the same elements in the table. Also, after a insert we need the guarantee that the data is inserted correct. There cannot be a partially inserted key. Therefore an insert needs to be atomic.

Proof. If the home-location of a key K is empty, these properties will follow from the atomic CAS operation on line 11. For every other case we will proof that there cannot be two workers in their CS working on the same area at the same time.
The '$\rightarrow$' operator is used to show a happens-before relation between to actions[5]. For example, $find(x) \rightarrow lock(table[x])$ shows that first x has to be found before its element can be locked. Furthermore, we distinguish two workers W_i and W_j and their t_left and t_right locations which are l_i, r_i, l_j and r_j. For the home-location $hash(K)$ we will use h_i en h_j.
We now explain the local ordering of a worker during an insert. A worker x searches for its *t_left* and *t_right* boundaries. After these are found, the worker tries to lock them both, starting with *t_left*. These locks take place on lines 21 and 23. If both locks are acquired, the worker is in it CS. Here a worker will occupy one of its empty boundaries. This happens on one of the following lines 33, 39 of 41. We describe these local orderings as follows: $find(l_x) \rightarrow find(r_x) \rightarrow lock(l_x) \rightarrow lock(r_x) \rightarrow (occupy(l_x) \oplus occupy(r_x))$.

THEOREM 4.1. *There cannot be two workers being in their CS at the same time and work on the same cluster such that* $l_i \leq l_j \leq r_i \vee l_i \leq r_j \leq r_i \vee (l_j \leq l_i \wedge r_j \geq r_i)$.

We will prove theorem 4.1 using a contradiction. Without loss of generality because of symmetry, we assume one worker W_i to have entered the critical section first and derive the following contradiction with Theorem 4.1:

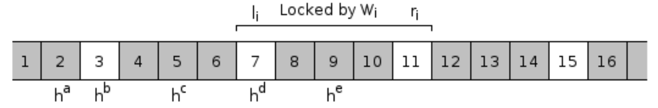


Figure 2. Several clusters and empty positions. The cluster with elements 8 to 10 is locked by worker W_i. Location marked with h^a to h^e potential home locations for worker W_j.

ASSUMPTION 4.1. *When worker W_i is already in its CS and another worker W_j arrives in his CS then the following holds:* $l_i \leq l_j \leq r_i \vee l_i \leq r_j \leq r_i \vee (l_j \leq l_i \wedge r_j \geq r_i)$.

The steps for a worker to arrive in its CS are:
$cas(h_j) \rightarrow find(l_j) \rightarrow find(r_j) \rightarrow lock(l_j) \rightarrow lock(r_j)$.
The remaining step for worker i which is already in its CS is: $(occupy(l_i) \oplus occupy(r_i))$

To prove theorem 4.1 we will use the following lemmas:

LEMMA 4.2. *Between a t_left and a t_right there are no empty elements.*

PROOF. The locations *t_left* and *t_right* are defined as follows: *t_left* is always the most closest empty element to h such that *t_left* $< h$. *t_right* is always the most closest empty element to h such that *t_right* $> h$. Furthermore, in algorithm 5 there does not exist an action for element x which results in $x.occupied \Rightarrow \neg x.occupied$. Thus, we can conclude: $\forall i : t_left > i > t_right \Rightarrow table[i].occupied)$ $\square$

LEMMA 4.3. *The possible values stated in Assumption 4.1 in combination with Algorithm 5 can be restricted to the following cases:* $l_i = l_j \vee r_i = l_j \vee l_i = r_j \vee r_i = r_j$.

PROOF. We will illustrate the cases of this proof with Figure 2. It shows an array of elements which represents the hash table. A cluster is an adjacent list of elements marked with the color gray. Every cluster is separated by an empty location marked with the color white. There are several representative home-locations marked with h^a to h^e (e.g., choosing a different location within the same region leaves the results of the $find()$ operations unaffected for one schedule).
Worker W_i is in its CS and has locked the cluster with boundaries l_i and r_i. W_i is going to perform the step $(occupy(B) \oplus occupy(C))$. Worker W_j has yet to enter its CS and is going to perform the the steps: $cas(h_j) \rightarrow find(l_j) \rightarrow find(r_j)$. Because of lemma 4.2 we know that there are no empty buckets between l_j and r_j. We will now show with some situations S_x for different home locations for worker W_j and interleavings with W_i that lemma 4.3 holds. Other possible home locations which are not covered here can be derived in a similar way, and will hold due to the symmetry of the table.
S_1: Worker W_j has $h_j = h^a$. This location is occupied so the $cas(h_j)$ fails. Then W_j performs the steps $find(l_j) \rightarrow find(r_j)$. $l_j < 1$ and $r_j = 3$. Since always $l_j < r_j < l_i$ Assumption 4.1 does not hold, making lemma 4.3 trivially true.
S_2: Worker W_j has $h_j = h^b$. This location is unoccupied and not locked, so the $cas(h_j)$ succeeds and the algorithm returns never reaching the critical section, making Lemma 4.3 vacuously true.
S_3: Worker W_j has $h_j = h^c$. This location is occupied so the $cas(h_j)$ fails. Then the step $find(l_j)$ results in $l_j = 3$,

Algorithm 5 Concurrent bidirectional linear find-or-put algorithm

```
 1: procedure FIND-FROM(K, j)
 2:     if table[j] > K then
 3:         while table[j].occupied ∧ table[j] > K do
 4:             j ← j − 1
 5:     else
 6:         while table[j].occupied ∧ table[j] < K do
 7:             j ← j + 1
 8:     return j
 9: procedure FIND-OR-PUT(K)
10:     h ← hash(K)
11:     oldval ← CAS(table[h], ¬table[h].locked ∧ ¬table[h].occupied, K)
12:     if ¬oldval.occupied ∧ ¬oldval.locked then
13:         return INSERTED
14:     else if oldval.locked then
15:         return FIND-OR-PUT(K)
16:     d ← FIND-FROM(K, h)
17:     if table[d].occupied ∧ table[d] = K then
18:         return FOUND
19:     t_left ← largest index < h such that ¬table[t_left].occupied
20:     t_right ← smallest index > h such that ¬table[t_right].occupied
21:     if ¬try_lock(table[t_left]) then
22:         return FIND-OR-PUT(K)
23:     if ¬try_lock(table[t_right]) then
24:         unlock(table[t_left])
25:         return FIND-OR-PUT(K)
26:     d ← FIND-FROM(K, h)
27:     if table[d] = K then
28:         unlock(table[t_left])
29:         unlock(table[t_right])
30:         return FOUND
31:     if DIRECTION() = DOWN then
32:         j ← t_right
33:         while table[j − 1].occupied ∧ table[j − 1] > K do
34:             table[j] ← table[j − 1]
35:             j ← j − 1
36:     else
37:         j ← t_left
38:         while table[j + 1].occupied ∧ table[j + 1] < K do
39:             table[j] ← table[j + 1]
40:             j ← j + 1
41:     table[j] = K
42:     unlock(table[t_left])
43:     unlock(table[t_right])
44:     return INSERTED
```

the result of $find(r_j)$ is dependent on the state of W_i. If W_i has not already performed $(occupy(7) \oplus occupy(11))$ than $find(r_j) = 7$, otherwise it depends if W_i has executed $occupy(7)$ or has executed $occupy(11)$. In the former case, $find(r_j) = 11$ and in the later, $find(r_j) = 7$. So, $r_j = 7 = l_i \vee r_j = 11 = r_i$.
S_4: Worker W_j has $h_j = h^d$. The result of the $cas(h_j)$ now depends on the state of W_i. If W_i has not performed any steps, then $cas(h_j)$ restarts the algorithm. If W_i has performed $occupy(7)$ than W_j continues with the algorithm and $find(l_j) \rightarrow find(r_j)$ results in $l_j = 3, r_j = 11 = r_i$. If W_i has performed step $occupy(11)$ then $l_j = 7 = l_i, r_j = 15$.
S_5: Worker W_j has $h_j = h^e$. Since h^e is occupied $cas(h_j)$ fails and W_j continues with the $find(l_j)$ and $find(r_j)$. The result depends on if W_i has executed $(occupy(7) \oplus occupy(11))$. For the following interleavings, we get the following results:
1: $W_j : find(l_j) \rightarrow W_j : find(r_j) \rightarrow W_i : (occupy(7) \oplus occupy(11)) \Rightarrow l_j = 7, r_j = 11 = r_i$
2: $W_j : find(l_j) \rightarrow W_i : occupy(7) \rightarrow W_j : find(r_j) \Rightarrow l_j = 7 = l_i, r_j = 11 = r_i$
3: $W_j : find(l_j) \rightarrow W_i : occupy(11) \rightarrow W_j : find(r_j) \Rightarrow l_j = 7 = l_i, r_j = 15$
4: $W_i : occupy(7) \rightarrow W_j : find(l_j) \rightarrow W_j : find(r_j) \Rightarrow l_j = 3, r_j = 11 = r_i$
5: $W_i : occupy(11) \rightarrow W_j : find(l_j) \rightarrow W_j : find(r_j) \Rightarrow l_j = 7 = l_i, r_j = 15$

Thus, for every interleaving, we have shown that in case Assumption 4.1 holds, the following holds: $l_i = l_j \vee r_i = l_j \vee l_i = r_j \vee r_i = r_j$. □

Using lemma 4.3 and knowing that worker W_i hash performed $lock(l_i) \rightarrow lock(r_i)$, we can conclude that for worker W_j always either line 21 or line 23 must fail. This is in contradiction with our Assumption 4.1, so we have proved mutual exclusion. □
Because of the mutual exclusion we can state that, writes are always executed exclusively within one region. Reads may fail at line 17 because a write is in progress by another worker, but in this case the algorithm will enter a read/write-exclusive state for that region (lines 26 to 30).

4.2 Deadlock Freedom

When two workers start waiting on each other, a deadlock situation occurs. There is no explicit existence of a busy wait section in the algorithm, there are however points where the algorithm may restart indefinitely, lines 15, 22 and 25. In all of the cases the algorithm restarts, it has encountered a locked element. But since every worker always unlocks its own locked elements, the restart will never result in a deadlock.

4.3 Starvation Freedom

When a worker always encounters a lock, and never can get to work, we can say the worker suffers from starvation. We conjecture that the algorithm always exhibits progress given that a fair scheduler is used. Therefore, we can say that this algorithm is safe from starvation.

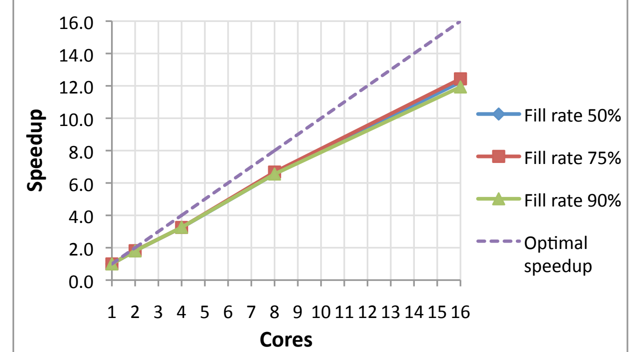
5. EXPERIMENTS

To determine the scalability of the parallel algorithm as showed in Algorithm 5, we implemented the algorithm on the Linux operating system in the C language using pthreads. The computer system is a 4 CPU AMD Opteron(tm) Processor model 8356 (thus, 16 cores in total) with 64GB of main memory. With the implementation we performed

Table 1. The amount of seconds for a certain benchmark to complete.

(Sec)	Fillrate		
workers	50%	75%	90%
1	23.13	38.05	52.62
2	12.77	20.95	29.17
4	7.12	11.70	16.14
8	3.53	5.70	8.03
16	1.89	3.06	4.41

Figure 3. The speedup of the parallel algorithm per fill rate, compared to base case with 1 worker.



several experiments which consists of creating a table with randomly generated values. The benchmarks, which the results can be viewed in Table 5, involved inserting the values by calling FIND-OR-PUT and reading them several times, by calling FIND-OR-PUT. For the benchmarks we used a hash table of size 2^{26} and a read/write ratio of 1:2. The experiments consisted of measuring the write and read time for several fill rates and workers. For the base case we take the amount of time, the benchmark takes for 1 worker to complete. Because of a lack of time, we did not measure the sequential algorithm, but we hypothesize that our parallel algorithm is not much slower. The speedup for a benchmark with multiple workers is determined by dividing the time of a benchmark by the amount of time for the base case.

As is shown in Figure 3, the parallel algorithm scales very well. We have achieved a maximum speedup of 12.4 with 16 workers with a fill rate of 75%. Due to this results we suspect the hash table to show similar results when using a 32 core machine. We did a quick comparison to the hash table developed by Laarman et al. [6] which showed that our table, in its current unoptimized form, is at least as fast as their table.

6. FUTURE WORK

Since we only made a quick implementation of the algorithm, it would be interesting to research if we can achieve more speedup by optimizing the table. Furthermore, it would be interesting to compare the performance of the hash table to other concurrent hash tables like the one from Laarman et al. [6] and Cliff Click [3].

The next phase would be researching which changes the algorithm needs to fit in the Cleary algorithm. Reading and writing at the same time is not possible with the Cleary algorithm. This is because the information during a read has to stay consistent, otherwise the algorithm does not know where an element can be found. Thus, all the read

operations need to be performed exclusively from write operations, hence we need read/write exclusion.

7. CONCLUSION

First, we have introduced a simplified version of the bidirectional linear probing insert algorithm of Amble & Knuth. In experiments this simplification, is shown to work as efficient as the original algorithm. Then we introduced a sequential find-or-put algorithm which we parallelized with almost no memory overhead. We proved the algorithm correct and made an implementation, which we benchmarked. Experiments show that table scales well, and we achieved a speedup of 12.4 with 16 cores. The development of this algorithm is one step closer to a concurrent compact Cleary table [2].

8. REFERENCES

[1] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.

[2] J. Cleary. Compact hash tables using bidirectional linear probing. *Computers, IEEE Transactions on*, C-33(9):828 –834, september 1984.

[3] C. Click. A lock-free hash table. Talk at JavaOne 2007, `http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf`, 2007.

[4] J. Geldenhuys and A. 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.

[5] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[6] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In N. Sharygina and R. Bloem, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*, USA, October 2010. IEEE Computer Society.