This is an implementation of Hopscotch Hashing as presented by Maurice Herlihy, Nir Shavit and Moran Tzafrir in their paper titled “Hopscotch Hashing”. We have adapted few of the strategies outlined in this paper and implemented the concurrent Hopscotch hashing algorithm as part of our ‘Distributed Computing’ course work.

The paper claims that Hopscotch Hashing algorithm delivers better performance as the table density increases compared to traditional hashing algorithms. To understand this behaviour we have benchmarked the Hopscotch Hashing algorithm with traditional Linear Probing algorithm in a concurrent testing environment.

Contact Authors
mp2737@nyu.edu, roystonpm@gmail.com, shp322@nyu.edu
1. Hopscotch Hashing

Hopscotch hashing is one of the open addressing techniques to resolve the collision in hash tables. Hopscotch hashing includes the features of linear probing, cuckoo hashing. This algorithm is designed to avoid the limitations and overheads in linear probing and cuckoo hashing.

Linear Probing is one of the traditional open addressing technique in which the items are kept in an array. When a new element needs to be added the array is searched from its hash location till it finds an empty location. This method helps in terms of caching as the structure is maintained in the cache due to multiple visits on the array structure. Unfortunately, contains() has to search through the array structure linearly and the performance degrades as the load factor increases. After a period of time, the unsuccessful contains has to go through the entire array to find that the element doesn’t exist.

Cuckoo Hashing is also one of the open addressing techniques but unlike linear probing cuckoo hashing requires deterministic number of steps to locate an element. Cuckoo hashing generally uses two hashing function. It tries to hash to first location if it is full it tries the next hash function and if both are full, it tries to push the element there to add the new element. The element pushed outside hash to other function and the cycle continues till no item has to be displaced or if the fixed number of displacement is reached in which case the table is resized. The disadvantage of cuckoo hashing is as it displaces through different sequence of unrelated locations in different cached lines. Also as the table load increases (about 50%), the number of displacement increases and causing it to resize often, so the performance reduces as the table density increases.

We will now show the implementation of Hopscotch algorithm which uses the advantages of the above mentioned algorithms.
2. Implementation of Hopscotch Hashing

Hopscotch uses a single hash function. The contains( ) will find the elements hashed into the entry or one of the next H-1 entries, H is constant. (Generally it is 32, standard machine word size). Each location in the array is called as a bucket, whose structure is

![Bucket Structure Diagram]

Key and Data is the usual (key, data) pair as in any hash algorithm. Hop-Info is an H-bit bitmap that indicates which of the next H-1 entries contains the elements that are hashed to this location initially. In this way an element hashed to a particular location can be found by scanning through the hop-Info and looking through the buckets that has elements hashed to this bucket. As we determine H based on the machine word size, the complete lookup can be done within two loads of cache lines.

For example assume H = 4

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>F</td>
<td>G</td>
<td>h</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig.2 The hop-info of bucket 2 is 1010. Most significant bit indicates 1, which means ‘c’ at bucket 2 is hashed to the same bucket. Then the next ‘1’ 1010, indicates the element at bucket 4, ‘e’ is actually hashed to bucket ‘2’
The lock is bucket structure is used to maintain a lock in a concurrent environment when different threads try to add or remove from the same location. It is sufficient to control the hop-info with the lock structure as the hop-info takes care of the access to key and data. We will see more about the lock structure in the later part of this article.

The timestamp in the bucket structure is used to synchronize the add method and contains method as contain method don’t need a lock to read.

3. Add method

We will see the implementation of add method through an example

Fig 3-a, b Different steps being followed for a adding an element. Assume H=4.

An element ‘x’ hash to bucket 6. But bucket 6 has an element ‘a’ already. So the algorithm looks through the structure to find an empty location. Bucket 13 is empty, but we cannot move the element ‘a’ at bucket 6 to bucket 13 as the constant H =4. As we compare if the element is empty we do a compare and swap for a dummy value to make sure no concurrent threads uses the location. We can only displace the element to the next H-1 buckets. So we start looking
to displace an element to move to free space at Bucket 13 and to find an empty slot for ‘6’. We first look at location 10, which is H-1 distance from 13, we realize that element at location 11 can be displaced from the hop-info of location 10. 0100 means the element hashed to location 10 has been displaced to location 11. So we move element at location 11 and change the hop-info of location 10 to 0001 (as the element earlier hashed to location 10 is moved to 13).

Please note here we need to get lock only on location 10 to displace element at ‘11’ to ‘13’ as the element is initially hashed to location ‘10’. As other operation (remove) also gets lock on the hashed bucket to remove an element, lock on the hashed bucket is sufficient.

Now the bucket ‘11’ is empty as the element is displaced to 13, but we cannot move the element at bucket ‘6’ as ‘11’ is still far away (H-1, 3 here). So we again look at H-1 previous location from ‘11’, at bucket ‘8’. We realize from the hop-info that element at ‘9’ can be displaced to bucket ‘11’ for the same reason as above. Once ‘9’ is displaced, we can move the element at bucket ‘6’ to add element ‘x’ to bucket ‘6’.

We change the value of timestamp before we release the lock. The timestamp as described before is used to synchronize add method and contains method. Contains method uses a wait-free approach and does not need a lock for its operation.

4. Remove method

The remove method works very similar to traditional hashing algorithm. Remove method locates the bucket to which the key is hashed to and then locks the starting bucket. It checks the starting bucket for the key, if it finds the key it sets to NULL and return after unlocking the bucket. If it could not find, we scan through the hop-info of the starting bucket and checks all the elements within its hop range H-1 to which the element hashed to the starting bucket has been displaced. If it finds the key, it removes the key by setting the key to NULL and returns after unlocking the bucket.

5. Contains method

Contains method locates the starting bucket by hashing the input key. It checks if the key of the starting bucket matches the key, if it matches it returns true. Else it makes use of the hop info of the starting bucket to find if the element is present within its Hop range H-1. When an element hashes to the particular location, the set bits in the hop info is extracted and verified in the hop
neighbourhood locations to find the element. If the element is not found in the
neighbourhood, a false is returned.

The timestamp as explained in add method helps to synchronize add and contain
method. If the timestamp is changed after the contain method starts its
operation, it signifies some other thread has changed the value. It checks again
through the structure else returns false.

Unlike linear probing, Contain method in Hopscotch hashing has to check only
H locations to find if the key exists or not.

6. Benchmarking of Hopscotch Hashing

The original paper based on which we have implemented Hopscotch hashing
claims that Hopscotch hashing continues to perform better as the table density
increases compared to most of the traditional hashing algorithm. To understand
the reasoning behind we ran a benchmarking on Hopscotch Hashing with
traditional linear probing in a concurrent testing environment.

We did our benchmarking in NYU Energon server, whose configuration is,
8 Core 1.8GHz Intel Xeon, 4 GB Memory. The maximum capacity of the hash
table is 1048576. Hop size H =32. We have executed our benchmarking on 2
different sets of operations. 90 % contains, 5% add and 5% removes –
commonly founded distribution in Hash tables. Also with an uncommon
distribution of operations 60% contains, 20% add and 20 remove.

For our both our cases we have executed 64 threads and 64000 operations
where each thread executes 1000 operations.
Fig. 4. 64 threads; 64000 operations; 60% contain, 20% add and 20% remove

Fig. 5. 64 threads; 64000 operations; 90% contain, 5% add and 5% remove

On both of the above benchmarking cases we could see that Hopscotch performs better than linear probing. Also as the table density increases, Hopscotch continues to perform better which is considered one of the important features of this algorithm. We believe the reason behind the better performance of Hopscotch hashing is even as the table density increases, hopscotch algorithm
has to scan through only 32 elements (H = 32) to find an element. But in linear probing due to contamination the number of search increases as the table density increases. Also an unsuccessful search sometimes causes the linear probing to scan through the whole array list whereas the hopscotch algorithm still has to look only in 32 buckets.

7. Conclusion
We have implemented Hopscotch hashing algorithm and compared its advantages over different algorithm in a concurrent testing environment. We believe with careful tuning of Hopscotch Bucket structure to fit a H buckets in a single cache line, scanning will need at most two cache lines.

For the source code please visit, http://github.com/harieshsathya/Hopscotch-Hashing

8. References
