PARALLEL DATA STRUCTURES & ALGORITHMS

Bloom filters

CS F367

PROJECT REPORT

By

Bhuvnesh Jain

2014A7PS028P

Under the supervision of

Shan Sundar Balasubramaniam, Professor,

Department of Computer Science & Information Systems

BITS PILANI, PILANI CAMPUS

ACKNOWLEDGEMENT

This Project was done under the guidance of Professor Shan Sundar Balasubramaniam. I thank him for providing insight and expertise that greatly assisted the research and implementation of the data structures and algorithms.

I would also like to thank Chirag Agarwal, my companion in the project, for providing assistance during implementations and the debugging.

CERTIFICATE

This is to certify that the Project entitled, **Parallel Data Structures and Algorithms – Bloom filters**, submitted by **Bhuvnesh Jain**, ID No. **2014A7PS028P**, in partial fulfilment of the requirement of **CS F367** embodies the work done by him under my supervision.

Signature of Supervisor

Date:

S Balasubramaniam

Professor, Department of CSIS

PROJECT ABSTRACT

The search for algorithms which can concurrently runs queries like insertion of elements in set, querying whether an element is present in the set or not etc. is still going on. Data Structures like binary search trees, hash tables, tries, link lists, bloom filters etc. exist which support above operations.

This project aims at designing an efficient algorithm for concurrent bloom filter. The algorithm designed should be able to efficiently carry out operations like insertion of object and querying whether an element is present in the set or not in parallel. The design of the algorithm should be portable across systems.

CONTENTS

BLOOM FILTERS

Bloom filter [1] is a probabilistic data structure which is used to efficiently query for the presence of an element in a set. The advantages lie in the fact that it never returns false negatives i.e. it never returns false when the object is present in the set. Although false positives are possible, their probability can be reduced by carefully choosing the Bloom filter parameters.

Sequential Algorithm Description

A Bloom filter is an array of “m” bits, where each bit can either take the value 0 or 1. Initially all the bits of the Bloom filter are set to 0, indicating the absence of any object in the set. We choose “k” independent hash functions that yield an integer in the range $[1, m]$ (inclusive). For each element to be inserted, “k” hash values are computed using these hash functions. The element to be inserted in the Bloom filter could be anything, varying from simple number to strings to complex structures. The only requirement is that it should be possible to calculate a hash value for the object being inserted. The pseudo code for the algorithm for *insertion* and for a *find* query in Bloom filter is given below:

```
void insert (object obj)
{
   for X in 1 to k
   {
      H = get_hash_value(obj)
      //1 <= H <= m
      set position H in bloom_filter to 1
   }
}

bool find (object obj)
{
   for X in 1 to k
   {
      H = get_hash_value(obj)
      //1 <= H <= m
      if (position H is not set in bloom_filter)
      {
         return false
      }
   }
   return true
}
```

The algorithm clearly indicates that the Bloom filter will never return false negatives as if the object is present in the set, the $\boldsymbol{k}$ bits corresponding to its hash function must have seen set to 1 by the insertion algorithm and found set, while querying for the object.

But false positives can exist as one or more hash values of an object may be same as hash values of one or more other objects. So, even if the object is not present in the set, its $\boldsymbol{k}$ bits could have been set by insertion of other objects.

Time Complexity Analysis

The insertion and query in the Bloom filter both take $\boldsymbol{O(h * k)}$ operations in the worst case, where $\boldsymbol{O(h)}$ is the complexity to find the hash value of the object being inserted. In hardware implementations, this can be reduced easily by computing all the hash values for the object in parallel as the computation of one hash function is totally independent of other. This can be viewed as an example of "data parallel" execution.

Number of False Positives

We assume that all the hash functions are independent of each other. Also, each hash function maps to each array index equiprobably.

Let **m** denote the number of bits in the Bloom filter, so the probability of a certain bit being set to 1 is $\frac{1}{m}$**.** Thus the probability of a bit not being set is $\left(1-\frac{1}{m}\right)$.

Since the hash functions are independent of each other, the probability that a certain bit is not set by any of the $\boldsymbol{k}$ hash functions is $\left(1-\frac{1}{m}\right)^k$. So, after inserting $\boldsymbol{n}$ objects into the Bloom filter, the probability that it is still set to 0 is $\left(1-\frac{1}{m}\right)^{nk}$

So the probability that a certain bit is set to 1 after n insertions is therefore $1-\left(1-\frac{1}{m}\right)^{nk}$.

For obtaining a false positive while querying for the object, all the $\boldsymbol{k}$ bits must have been set to 1. The probability that this occurs is thus,

$$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^k \approx \left(1-e^{-\frac{kn}{m}}\right)^k$$

Using calculus and other mathematical methods, one can easily show that to minimise the probability of false positives,

$$k = \frac{m}{n} \ln 2$$

False positive ratio is defined as the ratio of the number of false positives found to the number of total query operations performed on the Bloom filter. For a desired false positive ratio, p, we have

$$m = -\frac{n \ln p}{(\ln 2)^2} \text{ and } k = -\frac{\ln p}{\ln 2}$$

Space efficiency

Bloom filters have an advantage over other data structures like hash tables, tries, binary search trees, link lists etc. when we want to handle operations such as inserting an element into the set and finding whether the element exists in the set or not. This is because the size of Bloom filter doesn't depend on the structure of the object but on the number of objects in the Bloom filter. For example, for obtaining a false positive ratio of 0.01 (1%) we require the total number of bits in Bloom filter as

$$m = -\frac{n \ln(0.01)}{(\ln 2)^2} = 9.58506\, n$$

$$\frac{m}{n} = 9.58506$$

Thus, for obtaining false positive ratio of 0.01 (1%) less than 10 bits per element are required in Bloom filter.

Applications in Daily Life

Bloom filters has a lot of applications in our daily life, some of which are listed below.

1. Bloom filter is used intensively in deep packet inspection. In this the packet filtering examines the data part of the packet for spams, viruses or anything defined to be not-complaint. This requires fast search of keywords in the data packet which can be easily done using a Bloom filters once the list of keywords is inserted into it. In this case the insertion and query algorithm do not run in parallel at all. So, it can be implemented in "Data Parallel" fashion without use of locks as any threads wanting to just set the position to 1 in Bloom filter need not acquire a lock as the find operations (read) will only be done after all write operations (insertion) are over. [3]

2. Bloom filters are used to test whether an element exists on disk or not before performing an I/O operation. The Bloom filter can be stored in memory for even faster lookups and will reduce the number of I/O calls significantly for large datasets and hence improve the system performance.

3. Various online social networking sites like Facebook use Bloom filters for typeahead search queries [5].

PARALLEL IMPLEMENTATION – 1

Since the *insertion* and *find* in Bloom filter is like a kind of write and read operations respectively, they cannot be done in parallel simply as this may lead to wrong results.

Insertion of an element

The $\boldsymbol{k}$ different hash values for an element are calculated. These values are then sorted and locks are acquired in increasing order. This is done so as to avoid any deadlock with another *add* or *find* operation. After acquiring the locks, the bit corresponding to the hash position is marked to 1. The locks are then released in the same order in which they were acquired.

Finding an element

In the same way as in insertion of the element, the different hash values after calculation are sorted. The locks on the hash positions are acquired in increasing order. The bits are checked in the same order and if any bit was found to be marked as 0, it is concluded that the object did not belong to the set else the object belongs to the set. After this, the locks are released in the same order in which they are acquired.

Both these operations are executed by different threads running in parallel with the only bottleneck being the lock acquisition.

Correctness (Deadlock avoidance & Absence of False negatives)

False negatives can occur if the "find" operation reads a value to be not set although an add may be in the process of setting it. This can happen if the k different hash values are calculated one by one and the lock for the corresponding bit is acquired at the same time and released immediately after setting. This may not lead to a deadlock but may result in false negatives. So to overcome it all the $\boldsymbol{k}$ different hash functions are acquired first and then only are the bits set one by one. After the setting of bits or checking the bits are set or not is done, the locks for the $\boldsymbol{k}$ locations is released.

If this is executed in no fixed order, it may lead to a deadlock. So the process of acquiring lock, setting and releasing the locks is carried out in fixed increasing order. If the increasing order criteria was removed, then there might be a case that the first thread acquires a lock on a position, say $\boldsymbol{X}$, and other thread acquired a lock on position, say $\boldsymbol{Y}$, but now the first thread tries to acquire the lock on $\boldsymbol{Y}$ and other thread tries to acquire the lock on $\boldsymbol{X}$. Making the acquiring of locks in strictly increasing fashion ensures that such cases do not arise as both threads will now try to acquire the lock on position $\boldsymbol{X}$ only first and then on position $\boldsymbol{Y}$. (assuming $X <= Y$). The same argument to be extended to multiple threads and multiple locking positions as well.

PARALLEL IMPLEMENTATION - 2

There are five major changes to the previous parallel implementation of Bloom filters to increase the efficiency and speedup.

1. Reducing the memory size of the Bloom filter.
2. Using an algorithm to improve the locality effects.
3. Reducing the number of locks by half in case of “insertion”.
4. Making the lock operations on “insertion” operation a mix of fine-grained and coarse-grained locking.
5. Making the “find” operation lock-free.

Detailed Analysis

1. Reducing the memory size of the Bloom filters.

In the previous implementation, our Bloom filter was using a boolean variable for each location which consumes

$$Max.number\ of\ objects\ *\ No.of\ hash\ functions\ used\ *\ 1\ byte$$
$$(Size\ of\ bool\ =\ 1\ byte)$$

In our new algorithm, we use a **64**-bit integer to represent 64 Bloom filter locations. To be more precise i.e. we use 64 bit-vectors of size 64 to construct Bloom filters.

For example: Number 25 when written as 64-bit integer is
0000000000000000000000000000000000000000000000000000000000011001
which means position 1, 4, 5 will be set in out Bloom filter. Now the Bloom filter consumes

$$\frac{Max.number\ of\ objects\ *\ No.of\ hash\ functions\ used\ *\ 8\ byte}{64}$$
$$(Size\ of\ unsigned\ long\ long\ =\ 8\ bytes)$$

Thus the size of our Bloom filter reduces by 8 (= $64/8$). This will result in improved cache effects. This part will not have much effect in increasing the speedup but overall time taken by the Bloom filter will be reduced.

2. Using an algorithm to improve locality effects.

This algorithm is adopted from [4]. Below is a brief explanation of the same.

Let us assume the size of the Bloom filter is $\boldsymbol{m}$ bits of memory for **n** objects. Let us also assume the number of hash functions used is $\boldsymbol{k}$. Now, initially we used $\boldsymbol{k}$ polynomial hash functions for the strings with different values of primes and moduli. So each hash function uniformly map a location in $[1, m]$ in the Bloom filter.

Now, we split each hash functions into two categories – primary hash function and secondary hash function. In the primary hash function, we use the same algorithm as described before. For the secondary hash function, we use a different hash function which maps in the range $[1, w]$, where $\boldsymbol{w}$ will be described below. The insertion and find operations will be modified as below:

```
void insert (object obj)
{
   for X in 1 to k/2
   {
      H1 = primary_hash(obj)
      H2 = secondary_hash(obj)
      //1 <= H1 <= m and 1 <= H2 <= w
      set position H1 in bloom_filter to 1
      H2 += H1
      UPPER = ceil(H1/w)
      if (H2 > UPPER)
      {
         H2 -= w
      }
      set position H2 in bloom_filter to 1
   }
}

bool find (object obj)
{
   for X in 1 to k/2
   {
      H1 = primary_hash(obj)
      H2 = secondary_hash(obj)
      //1 <= H1 <= m and 1 <= H2 <= w
      if (position H1 is not set in bloom_filter)
      {
         return false
      }
      H2 += H1
      UPPER = ceil(H1/w)
```

```
            if (H2 > UPPER)
            {
               H2 -= w
            }
            if (position H2 is not set in bloom_filter)
            {
               return false
            }
         }
         return true
      }
```

In the above algorithm, the object can be datatype, structure or class for which hash function can be computed using any algorithm. The role of upper is that we set the second bit always in maximum of w bits space from first one. This may seem that the hash functions are now not independent on each other and the above mathematical analysis of the Bloom filter might fail, but careful mathematical analysis and the properly chosen value of w can ensure that the claim still holds true. A more detailed mathematical analysis can be found in [4].

Choosing w large enough and such that it is equal to the cache block size, would not only give better performance with respect to false positives but also increase the cache hit rate. This is because the second memory location to be searched or set in Bloom filter would be present in the cache as the array is brought in chunks w bytes from memory into cache while in the previous algorithm, the chances that the next hash function also maps in w space neighbourhood of the previous hash function is very unlikely, assuming they are independent of each other. So, the running time of the Bloom filter will decrease. Now, this will also increase locality in each thread and might help in increasing the speedup, but only marginally.

3, 4. Reducing the locking overhead in "insertion".

Now using the above 2 claims, we chose the size of w as $\mathbf{16}$. Since, we use 64-bit vectors to construct Bloom filters and primary and secondary hash values are within a distance of **w (16)**, locks are required to be acquired only for primary hash value and not for secondary hash value. Since 64 is divisible by 16, the lock for secondary hash value will always be same as the primary hash value. Thus the overhead of acquiring and leaving the locks is reduced by half. This gives more chances of parallelism in our algorithm and also better speedup as well.

Initially we used fine-grained model of locking the positions in the Bloom filter i.e. each location had a lock for itself. In the new algorithm, we use a mix of coarse-grained and fine-grained locking. Coarse grained locking has the disadvantage of acting as a bottleneck as only one thread can operate at a moment on the data structure but has the advantage of lesser overheads as compared to fine grained locking due to the number of locks that are acquired and released. On the other hand, fine grained locking has the advantage of letting several threads work on the data structure simultaneously. So, we tried to incorporate the benefits of both the locking methods in our algorithm.

In our earlier model, the chances of two threads acquiring the lock on same position was less as two objects hashing to the same value is less (provided the hash functions are chosen properly). In our new algorithm, since we incorporated coarse grained locking the chances of two primary hash functions locking the same positions will increase. But similar to the mathematical argument shown in [4] regarding the false positive ratio, we claim that a well-chosen value of $\boldsymbol{w}$ will almost give the same probability as the first one within some error limit which is too small.

Also, one needs to keep in mind that while implementing the algorithm, we require Reentrant locks as multiple threads may acquire the lock on the same position. This is because each 64 bit-vector has a coarse-grained lock as described above. If we want to avoid using Reentrant locks, we can then acquire the locks for the unique hash positions only. This will have an overhead of making the array of hash values, where lock is to be acquired, unique after sorting. Reentrant locks have internal overheads of maintaining the count by each thread. The overall speedup results were measured on experimental basis and in our final implementation, we use Reentrant locks.

Our experimental results confirm our theoretical and intuitive ideas mentioned above. The speedup of the parallel version of the algorithm increased and the running time also improved a lot due to more cache hits & better locality effects.

5. Making the find operation lock free.

We argue that making the “find” operations lock free will not introduce false negatives although it may increase the false positives by a small amount.

We observe that a false negative is a case when an object is in the set but the Bloom filter says that the object is not in the set. This case will never occur as we make sure that we set all the bits of the Bloom filter in one go, not bit by bit for the $\boldsymbol{k}$ hash positions. So, any thread executing the find operation on an object that is present in the set will find all the bits being set (as partial setting is not done). So false negatives cannot occur.

But, there is a chance that false positives ratio may increase. This can happen when the thread first operates on the find query partially, find all the bits set to one till now and when the next bit which was earlier set to zero (and would have indicated that the object doesn’t exist in the Bloom filter) is set to one by other thread executing in parallel. The chances of such cases occurring are very less when the size of the hash table is quite large but it increases slightly with the increase in number of threads. This is because as the number of threads executing in parallel increases, so does the chance of another thread executing the add operation and setting that bit to one. But, on experimental basis it has been seen observed that in most cases the results are same as the sequential version of the code but in some cases the false positive ratio increased but not much.

The below table show the number of false positives obtained in the sequential and Parallel implementation – 2 when the false positive ratio was set to 0.0025 (0.25%).

Number of Query operations	**Sequential**	**Parallel Implementation – 2**
100	0	0
1000	0	0
10000	1	1
100000	2	3

Correctness (Deadlock avoidance & Absence of False negatives)

We have explained above in detail that our new algorithm for parallel Bloom filter will not produce false negatives.

Also, we need to ensure that the algorithm is deadlock free. For this, we see that locks are only acquired for "insertion" operations and that too in increasing order which ensures that a deadlock will not occur. This argument is the same as the one for the previous algorithm.

EXPERIMENTAL DETAILS

In our project, the objects were **strings** and the hashing scheme used for them were **polynomial hash functions** where the moduli and multiplying factors were chosen randomly by generation of primes numbers depending on the size of the Bloom filter.

Generation of Test cases

The datasets were generated randomly using the $rand()$ function in C++. At the start of the program, the $srand()$ function was called to make sure that each test case had different data. The sequence of operations (insert and find) was also randomly generated. Each string had a length between 5 and 10 (inclusive).

There were four different types of datasets generated, depending on two parameters:

1. The number of add operations in total.

2. The number of query operations such that the string asked was definitely in the set.

For all the datasets, the initial queries consisted of insertion only. About $1/3$rd of the total insertion queries were given to the Bloom filter initially. After that a mix of insertion and find queries were given.

The different datasets had (a, b) as:

1. 25 – 3

2. 20 – 3

3. 10 – 2

4. 5 – 2

Where the percentage of insertion operations were $\frac{100}{a} * 3$,

and $\frac{100}{b}$ % of the find operations were from strings already present in the set.

Results & Analysis

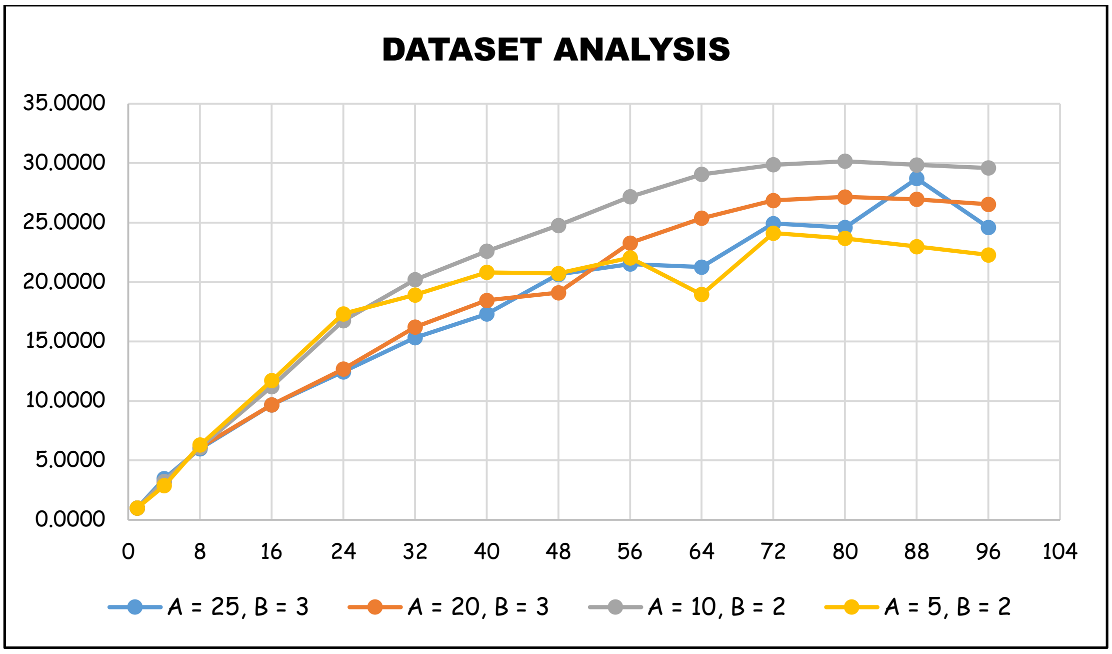
The results given below were obtained on “Intel Xeon” 48-core processor, having support upto 96 threads (using hyperthreading). The programs were written in “C++” and OpenMP was used for parallelising the code.
All the different datasets generated were run for five times and the average timings are reported. The total number of insertion and find operations chosen was 10^8. Files of size $1\,GB$ were generated to check the speedup of the code.
While running the programs, the maximum virtual memory usage was reported to be around 6.5 GB.

The table below shows the execution time in seconds for the different datasets.

Threads	**A = 25 B = 3**	**A = 20 B = 3**	**A = 10 B = 2**	**A = 5 B = 2**
1	144.0181	154.6631	208.0350	217.4122
4	41.5248	48.6394	64.1870	75.7720
8	24.0476	25.3927	34.1022	34.4934
16	14.8799	15.9892	18.5526	18.5667
24	11.5597	12.1905	12.4167	12.5408
32	9.4009	9.5403	10.3019	11.4930
40	8.3170	8.3794	9.2078	10.4401
48	6.9776	8.0917	8.4011	10.4871
56	6.6926	6.6428	7.6533	9.8632
64	6.7728	6.0962	7.1583	11.4663
72	5.7768	5.7559	6.9635	9.0165
80	5.8571	5.6941	6.8947	9.1848
88	5.0158	5.7369	6.9683	9.4593
96	5.8543	5.8264	7.0261	9.7562

The table below shows the speedup obtained in each case.

Threads	**A = 25 B = 3**	**A = 20 B = 3**	**A = 10 B = 2**	**A = 5 B = 2**
1	1.0000	1.0000	1.0000	1.0000
4	3.46824	3.17979	3.24108	2.86929
8	5.98888	6.09085	6.10034	6.30301
16	9.6787	9.67297	11.2133	11.7098
24	12.4586	12.6872	16.7545	17.3364
32	15.3196	16.2115	20.1938	18.9169
40	17.3161	18.4576	22.5935	20.8247
48	20.64	19.1138	24.7629	20.7314
56	21.519	23.2829	27.1825	22.0428
64	21.2641	25.3706	29.0621	18.961
72	24.9303	26.8703	29.8749	24.1128
80	24.5888	27.1619	30.173	23.6709
88	28.7131	26.9594	29.8544	22.984
96	24.6004	26.5452	29.6089	22.2845



Conclusions

1. The scaling was almost linear till 8 threads, then kept increasing at almost linear till 24 threads, after which it dropped and became sublinear. The best speedup obtained on 48 threads was 24.8.
2. Due to hyperthreading, the scaling theoretically should not be linear which was also seen in the experimental results. The scale increased marginally from 48 threads to 72 threads in all the datasets after which it either remained same or dropped.

LIMITATIONS

1. There is a possibility of starvation in our code, although it is rare in practice. This is because the same thread could be waiting from the start till end doing just one operation in the whole dataset, trying to acquire the lock for the "insertion" operation.

2. The new algorithm proposed works well with the basic variant of Bloom filters, but doesn't scale with different types of Bloom filters. For example, it will not scale well with Counting Bloom filters as the single bit positions can't be used as counters. But idea similar to the one used in this project can be extended by making the single number being divided into 4 bits, thus representing a total of $64/4 = 16$ Bloom filter locations and thus then supporting counting Bloom filters as well. Also, in that case the deletion operation in counting Bloom filter would require a lock. These claims could not be tested experimentally within the stipulated time of the project. But scope is there to improve the algorithm further to make it efficient for all types of Bloom filters.

3. The locking currently used in the design of the algorithm was one available as "Open MP" library function. Making use of "Test-Test-&-Set locks" might have given further increase in speedup as well as the operations are atomic in this case. But experimental results only could verify this claim.

REFERENCES

1. Bloom filters - Wikipedia
   https://en.wikipedia.org/wiki/Bloom_filter
2. Analysis of Bloom filters by Jamie Talbot
   https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff
3. Deep Packet Inspection using Parallel Bloom Filters by Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull and John Lockwood.
   https://www.arl.wustl.edu/~todd/hoti.pdf
4. A New Design of Bloom filter for Packet Inspection Speedup: Yang Chen, Abhishek Kumar, and Jun (Jim) Xu College of Computing, Georgia Institute of Technology
   www.cc.gatech.edu/~jx/reprints/globecom07.pdf
5. Bloom filters usage by Facebook.
   https://www.facebook.com/Engineering/videos/432864835468/