Bloom Filters Parallelization

Davide Cremonini, Hannes Wiedenhofer and Davide Sbetti [1]

Index terms— **Bloom filter, Parallel Computing, OpenMP**

I. Introduction

This paper analyses different approaches for the parallelisation of Bloom filters, space-efficient data structures used to efficiently identify the presence of an element in a set.

More specifically, we focused on CPU parallelisation, implementing various approaches using OpenMP, comparing a sequential baseline with an OpenMP baseline, an implementation using atomic operations and an algorithm based on the reduction strategy and a filter for each thread.

The various implementations were tested on two different machines, with the results showing how the atomic and reduction approaches obtained the best performance and speedup.

In section II we review the basic concepts related to the idea of Bloom filters and their characteristics, while in section III related works are presented. Section IV outlines the details of the various implementations that were tested, with the results of our experiments being presented in section V. Finally, our work is summarised in section VI, which reports also future directions.

II. Background

Bloom Filters are space-efficient probabilistic data structures which are used to efficiently identify whether an element is present in a set. They solve the approximate set membership problem. The components of a Bloom filter are the following:

- an array of n bits
- k independent random hash functions, each with a range between *0* and *n-1*

When a new element is inserted, k positions belonging to the Bloom filter's array are obtained by applying the k hash functions to the input element, with the corresponding bits being set to 1. A description of this process is outlined in figure 1.

In order to check whether an element is present in the set, the hash functions are calculated on the chosen element and the value of the corresponding bits is observed. If all the bits are set to 1, the conclusion that can be drawn is that the element *could* be present in the set. On the other hand, if at least one bit is set to 0, the element *is for sure* not in the set.

As outlined above, Bloom filters are used to solve the approximate set membership problem, meaning that the result derived from the filter could not be correct. In particular, we can notice that each bit can be set to 1 by different elements resulting in the same value for the given hash function and output range. For this reason, we can observe how the Bloom filter could report false positives, although there cannot be false negatives.

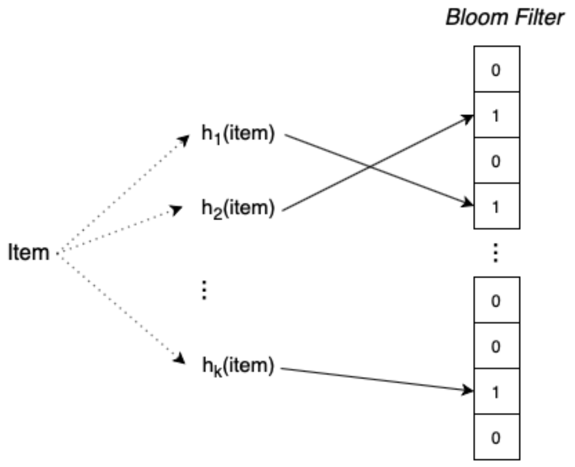


Fig. 1: Overall process

Different studies have been conducted on the false positive rate reported by Bloom filters. In particular, Mitzenmacher and Upfal [1] report how, after having hashed all m elements, the probability of a false positive has as upper bound the following value:

$$\begin{gathered}(1-p)^k \\ \text{with } p = e^{-km/n}\end{gathered} \tag{1}$$

where k is the number of hash functions, m the number of elements to be inserted in the Bloom filter, which has a size of n bits.

This upper bound holds for large datasets and is useful to ensure the correctness of the different implementation.

III. State of the art

There exist multiple papers and studies focusing on the parallelisation of Bloom filters.

[2] focused on an implementation of standard bloom filters using OpenMP and atomic operations, in an algorithm similar to our Atomic implementation, without exploring other strategies.

On the other hand, [3] compares different Bloom filters implementations, reporting if the they can run in parallel and the overall complexity of the algorithm, without going deeply into the details of the implementations.

Another approach is presented by [4], who propose a parallel implementation of a counting Bloom filter, where each single bit represents a counter of how many times the bit was mapped to an element.

[1]Free University of Bolzano, e-mail: {dcremonini, hwiedenhofer, dsbetti}@unibz.it

A further study was performed by [5], who investigated how a multi-attribute Bloom filter, thus a filter recording multiple values for each element, can be parallelised.

As we can observe, various studies were performed on Bloom filters parallelisation. However, no previous study investigated and compared multiple parallel implementations of a standard Bloom filter on OpenMP.

IV. Bloom Filters Implementations

In this section we present the different implementations that were used in our comparison.

Regarding the parallel implementations, we decided to concentrate on exploiting CPUs to parallelise the problem, focusing so our attention on OpenMP, a portable and scalable model for the development of parallel applications in C/C++ or Fortran. In our case, all algorithms were implemented in C.

A. Sequential Baseline

The first implementation, which consists of the sequential baseline used to compare the parallel approaches, strictly follows the process described in section II.

First of all, the input and test datasets, both contained in a CSV file, are read into memory. Then, the Bloom filter is created, with n 32-bits words allocated into memory, obtaining so a $n * 32$ total size of the filter.

Moreover, starting from a parent seed provided as input, k seeds are generated using the Xoshiro pseudo-random generator [6], approach used to ensure that the seeds generator is consistent across different platforms.

The objects are inserted in the filter, then a check for errors is performed: the algorithm ensures that for each input element the positions in the filter are all set to 1. After that, the same process is repeated using the test set, counting the false positives and calculating the false positive rate. The results are stored in a csv file.

The subsequent approaches partially follow the same procedure. In the associated subsections only the specific changes for each approach are reported, changes mostly related with the insertion technique.

B. OpenMP Baseline

In this approach, during the insertion process, a lock is set over the whole filter by using an OpenMP critical region around the whole insertion function, including the hash calculation. This allows only one thread at a time to access this region.

Moreover, queries are also performed in parallel. A difference, when comparing to the insertion process, is that no lock is required, since checking the bits of the filter to evaluate their values is a reading operation and concurrency can be allowed. This parallelisation of queries have been maintained also in the subsequent approaches.

C. Atomic

In the previous approach, the lock was requested by each thread before calculating the different hash values. However, calculating the hash values on each element is an operation that does not lead to a race condition on the Bloom filter. For this reason, we decided to replace the critical region, present in the insertion process, by an atomic or-operation when setting the single bits. In this way the filter is locked only when setting the bits, but concurrency is still allowed when calculating the hash.

D. Reduction

When applying the reduction strategy, each thread has a separate entire Bloom filter in which all elements assigned to the thread are inserted, without using locks. At the end of the insertion process, when all threads completed insertion, an OR tree reduction is performed, using the OpenMP reduction operator, merging the various Bloom filters in a single one.

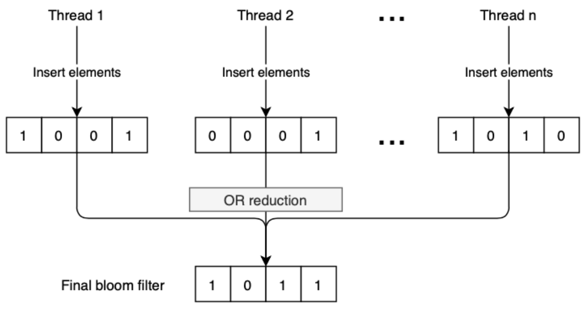
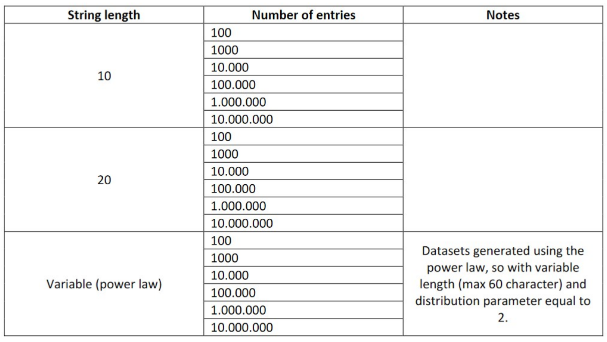


Fig. 2: Reduction approach

V. Experiments

A. Datasets

We decided to synthetically generate various types of string datasets using Python scripts, process that allowed us to create the following datasets:



String length	Number of entries	Notes
10	100	
	1000	
	10.000	
	100.000	
	1.000.000	
	10.000.000	
20	100	
	1000	
	10.000	
	100.000	
	1.000.000	
	10.000.000	
Variable (power law)	100	Datasets generated using the power law, so with variable length (max 60 character) and distribution parameter equal to 2.
	1000	
	10.000	
	100.000	
	1.000.000	
	10.000.000	

The Python scripts used to generated the datasets can be found in the Unibz Gitlab associated with the project (link in the following section).

B. Environment

The different implementations, outlined in the section IV, have been run on the following machines:

	Lenovo Legion 5 17IMH05H	**Ironmaiden UNIBZ machine**
Number of CPUs	1	1
CPU	Intel® Core™ i7 – 10750H	Intel(R) Xeon(R) CPU E5-2667
CPU Base Frequency	2.60 GHz	2.90 GHz
Number of cores	6	32
Hyperthreading	enabled	enabled
OS	Windows 10 20H2	Ubuntu 16.04.7 LTS

C. *Results*

After running the experiments on the machines outlined above, we used Python to plot the results in order to compare the same execution in the various environments.

Each plot is the result of averaging ten runs of the same experiment, discarding the fastest and slowest executions. Moreover, the plotted time (which is in milliseconds) is the average time of a single insertion/query. For queries, we prepared an equally sized dataset, with elements not present in the input one.

C.1 Chunk size

Initially, we performed some experiments using the baseline OpenMP implementation and different chunk size and scheduler combinations, in order to identify the most suitable parameters for the subsequent experiments. Below, we report the results of our experiments, in terms of insert time:

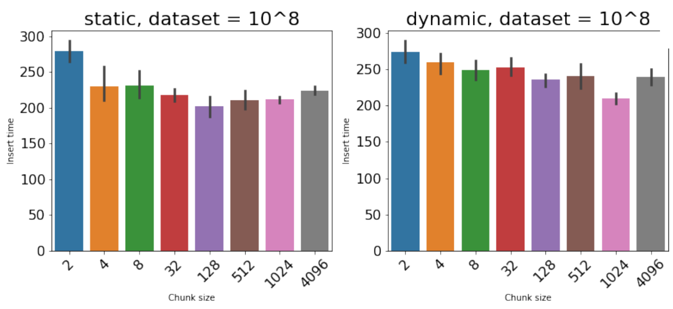


Fig. 3: Chunk size experiments results

After analysing the plot above, we decided to choose 128 as chunk size with a static scheduler, as it appears to be the fastest combination among the tested ones, although the confidence intervals show how the difference is not statistically significant.

C.2 Insert time

After choosing the chunk size, we run the various implementations on the two machines, recording separately the insertion and query time. We report below a comparison of the insertion time for the dataset composed of 20 characters strings, with 10 Millions entries. The results for the other datasets are omitted for conciseness and can be found in the repository.

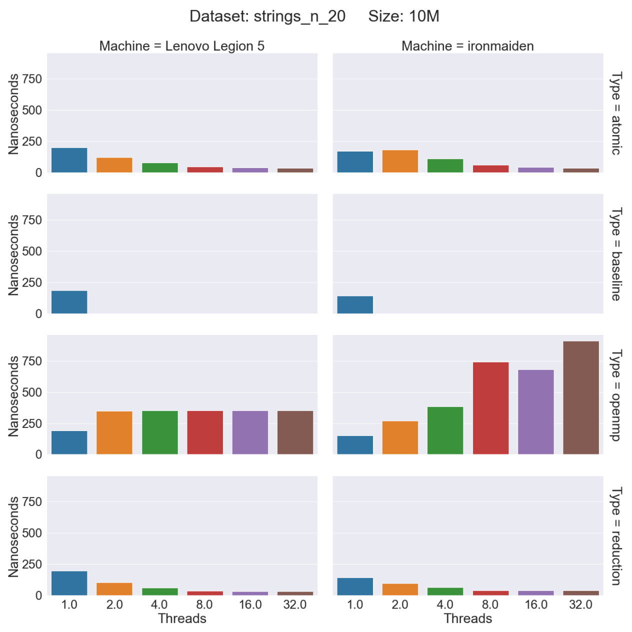


Fig. 4: Insert time for 10 Millions 20 characters strings

We can observe how most of the results are as expected. Atomic and Reduction appear to be the fastest implementations, scaling also relatively well with the number of threads. These results can be explained, in case of the Atomic implementation, with the lock being applied only when setting a particular bit in the Bloom filter. Moreover, with a large filter, the probability that more than one thread access the same memory word concurrently is low, resulting in few conflicts. Regarding reduction, the speedup is obtained by the absence of locks, given that each thread has its own filter. After insertion, there is a further step, namely filters' reduction, but also at this point no locks are necessary. The most surprising result is the one reported by OpenMP, which does not show any improvement but, on the other hand, a deterioration in terms of performance which, in case of Ironmaiden, is also related to the number of threads.

This can be explained by considering the fact that, when applying the lock on the entire Bloom filter, this translates the parallel implementation to a sequential one, with in addition the cost of creating and managing the threads.

Moreover, we report the insertion time for the power-law dataset, with the same number of entries as the previous one, for comparison:

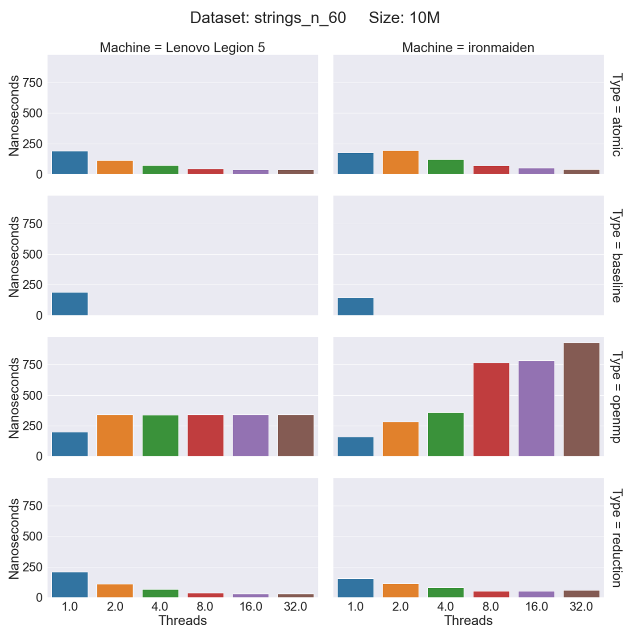


Fig. 5: Insert time for 10 Millions variable length strings

We can observe how a variable length of the strings used does not significantly affect the results.

C.3 Query time

In this section we report the average query times associated with the two datasets analysed previously. As noted before, when performing queries it is not necessary to apply any type of lock, since it is a read-only operation. Moreover, since the elements of the query dataset are not present in the input one, it is important to take into consideration how query time changes with the elements, since whenever a 0 bit is encountered the query algorithm returns false without checking the remaining bits.

We start by considering the query time for the 20-characters strings:

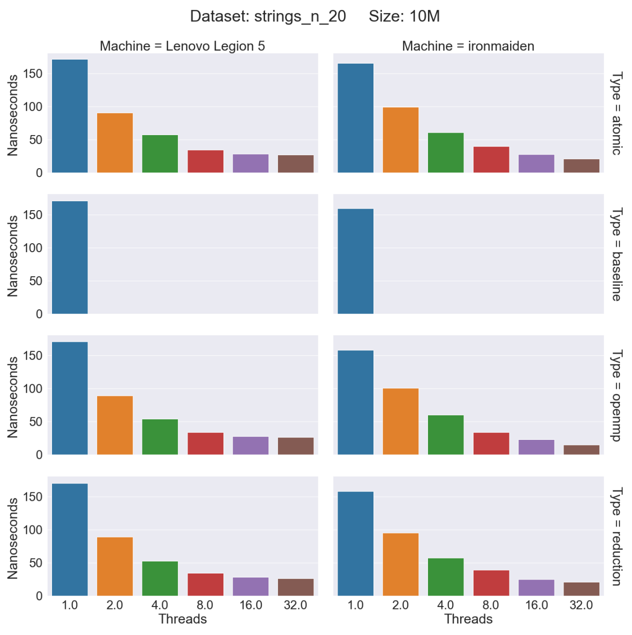


Fig. 6: Query time for 10 Millions 20 characters strings

Moreover, for completeness, we report the time also for the power-law considered dataset:

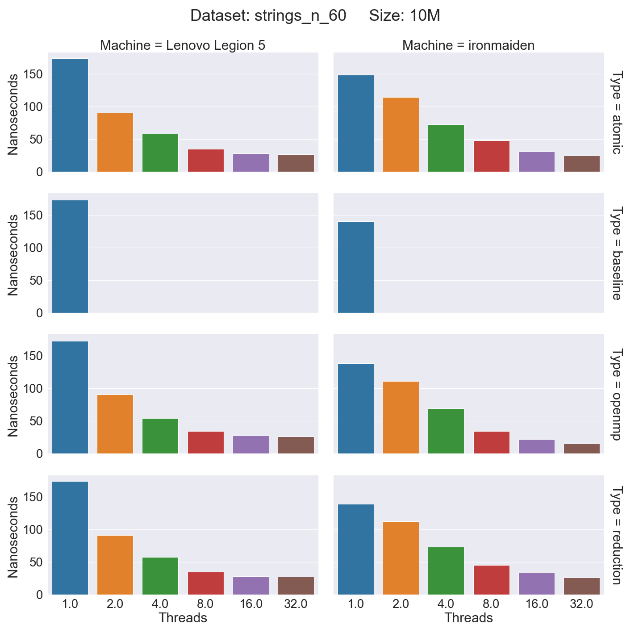


Fig. 7: Query time for 10 Millions variable length strings

In both plots, we can observe how the query time scales well in all implementations, including baseline OpenMP, and datasets. This enforces our hypothesis of the insertion time of baseline OpenMP being affected by the overall lock on the entire filter.

C.4 Speedup

After considering the insertion and query times, we generated also the speedup plots associated with the various implementations.

We report below the results for both datasets:

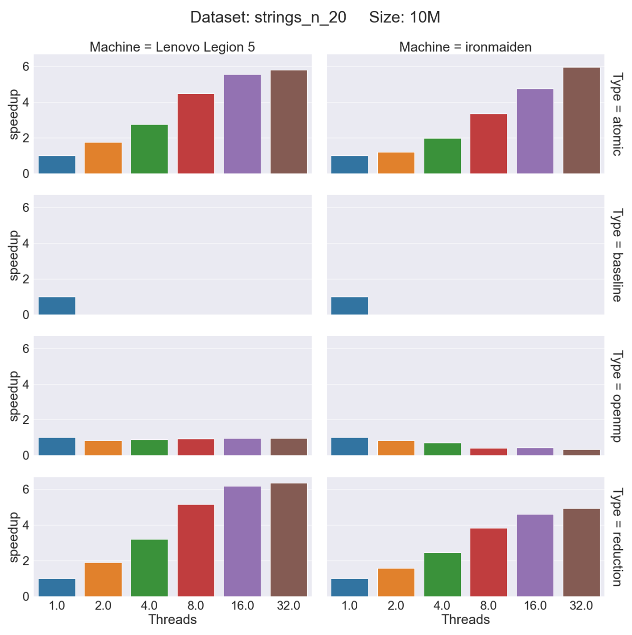


Fig. 8: Speedup for 10 Millions 20 characters length strings

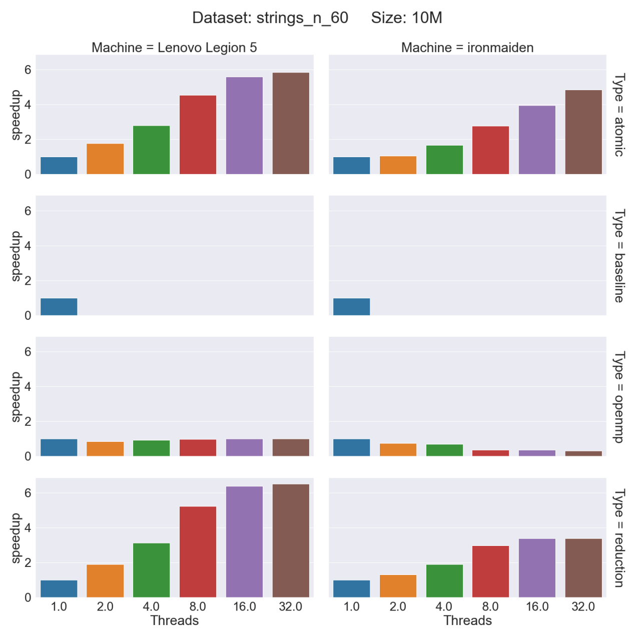


Fig. 9: Speedup for 10 Millions variable length strings

We can observe how the various implementations scale with the number of threads, except for baseline OpenMP, as outlined previously. Moreover, it is interesting to note how, with both datasets, the reduction implementation scales better on the Lenovo machine.

On the other hand, the work load on the different machines could be too low, resulting so in a large contribution of the overheads, related with the creation and management of threads, to the overall results, yielding an imperfect speedup.

VI. Conclusions

In this paper we experimented with the parallelization of Bloom Filters using different approaches. The results showed how it is possible to effectively parallalelize the problem, with the fastest algorithms being atomic and reduction, when considering insert time.

On the other hand, the OpenMP baseline approach of using an entire lock on the filter did not prove suitable for the type of problem, as it translates the parallel implementation to a sequential one.

The outlined experiments could be similarly replicated applying GPU parallelization, for example using CUDA. In this work we focused on CPU parallelization, leaving this possibly faster approach as future work.

References

[1] Michael Mitzenmacher and Eli Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, USA, 2005.

[2] Wenmei Ong, Vishnu Monn Baskaran, Poh Kit Chong, K. K Ettikan, and Keh Kok Yong, "A parallel bloom filter string searching algorithm on a many-core processor," *2013 IEEE Conference on Open Systems (ICOS)*, 2013.

[3] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

[4] Sheng Ni, Rentong Guo, Xiaofei Liao, and Hai Jin, "Parallel bloom filter on xeon phi many-core processors," *Algorithms and Architectures for Parallel Processing Lecture Notes in Computer Science*, p. 388–405, 2015.

[5] Yu Hua and Bin Xiao, "A multi-attribute data structure with parallel bloom filters for network services," *High Performance Computing - HiPC 2006 Lecture Notes in Computer Science*, p. 277–288, 2006.

[6] "xoshiro/xoroshiro generators and the prng shootout," `https://prng.di.unimi.it/`, Last access 5 May 2021.