UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel computing

Parallel implementation of Bloom Filter

Relatore: Leonardo Di Iorio

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduction

The following presentation will show and compare two different implementations of the Bloom Filter data structure:

- **Sequential**
- **OpenMP**

UNIVERSITÀ DEGLI STUDI FIRENZE

Bloom Filter

- A Bloom Filter is a probabilistic data structure employed to determine the membership of an element in a set efficiently;
- One of the characteristic elements of Bloom Filter is that it is impossible for a Bloom Filter to provide a false negative but it is possible to provide false positives;
- We can divide the functioning into two steps:
  - **Inizialization**;
  - **Search**;

Inizialization

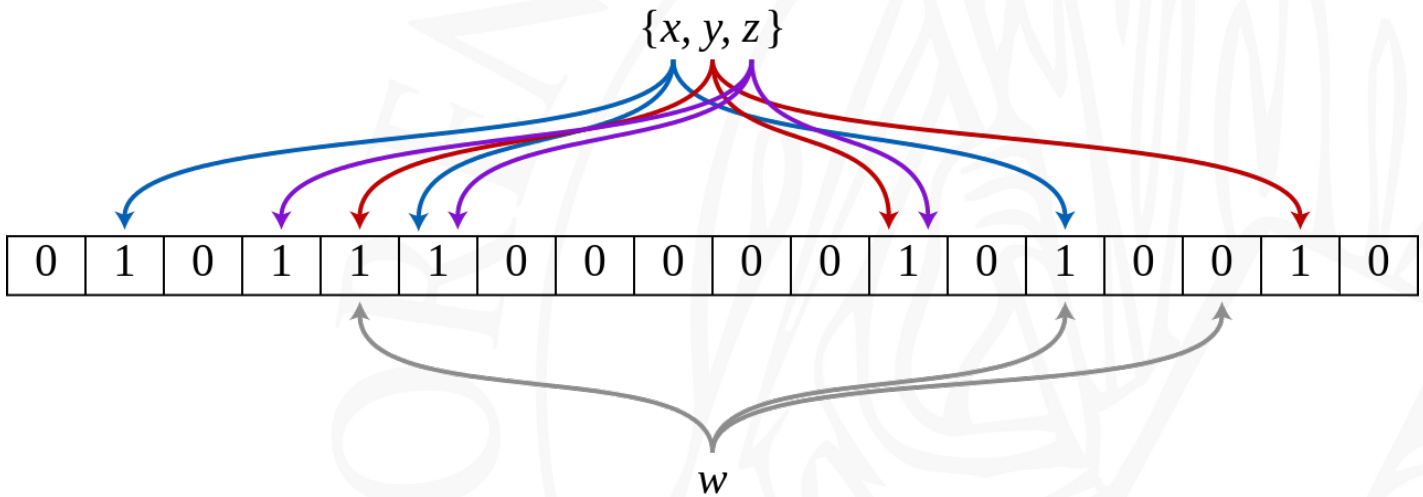
At this stage, the Bloom Filter is filled based on the set of eligible entries and **k** chosen hash functions. More specifically:

- An empty Bloom filter is an array of **m** bits all set to 0;
- **k** different hash functions must be defined;
- Each of the hash functions maps an element of the set to one of the **m** positions in the array with a uniform distribution;

Inizialization

- To add an element, all **k** hash functions are computed to obtain the **k** positions in the array
- Then the bits in those **k** positions are set to 1;

In this implementation the set of elements used to fill the filter is a set of strings.



Search

In this step, it is checked whether an element placed in the filter input belongs or not to the set of allowed elements:

1. To test an element on the set, the **k** hash functions previously defined are computed on the element and then the **k** positions of the array of **m** bits are checked;
2. If even one of the checked bits in the array is 0 then the element is not within the admissible set;
3. If all of the checked bits are set to 1 then it means that either the element is within the set or the filter is returning a false positive.

Sequential

The sequential version of the code is implemented in C++. Despite this, the code is not developed with an object-oriented programming approach:

- To avoid slowdowns in execution;
- To achieve code that is as similar as possible between the sequential and parallel versions;

Sequential

So only one class was implemented, called **BloomFilter** with three different methods:

- The constructor;
- The **computeBloomFilter** method;
- The **checkStream** method;

The number of bits composing the filter and the number of hash functions to use are computed through the functions, **getNumberOfBits** and **getHashNumber**.

Sequential

- The output vector is a binary vector whit a number of bit equal to the size of the set to check through the bloom filter;
- Once an entry has been checked if it's found to be not admissible it's corresponding bit in the output vector will be set to 0;
- The hash function used is the **MurmurHash**.

Sequential

The code for the **computeBloomFilter** method:

```
25  void BloomFilter::computeBloomFilter(){
26      for(int i=0;i<legalStrings.size();i++){
27          for(int j=0;j<numberOfHash;j++){
28          uint32_t h=MurmurHash(&*legalStrings[i].begin(),sizeof(legalStrings[i]),j)%numberOfBits;
29          this->bitVector[h]=1;
30          }
31      }
32  }
```

Sequential

The code for the **checkStream** method:

```
35  void BloomFilter::checkStream(vector<string> check){
36    for(int i=0;i<check.size();i++){
37      for(int j=0;j<numberOfHash;j++)
38      {
39        uint32_t h=MurmurHash(&*check[i].begin(),sizeof(check[i]),j)%numberOfBits;
40        if(bitVector[h]==0){
41          this->output[i]=0;
42        }
43      }
44    }
45  }
```

OpenMP

The parts of the implemented code that need to be parallelized in order to achieve performance improvement are the external for loops:

- In the computeBloomFilter method;
- In the checkStream method.

In both cases the parallelization of the loops has been obtained through the following directive:

- **#pragma omp parallel for**

OpenMP

The code for the openMP version of the **computeBloomFilter** method:

```
25  void BloomFilter::computeBloomFilter(){
26  #pragma omp parallel for
27      for(int i=0;i<legalStrings.size();i++){
28          for(int j=0;j<numberOfHash;j++){
29          uint32_t h=MurmurHash(&*legalStrings[i].begin(),sizeof(legalStrings[i]),j)%numberOfBits;
30          this->bitVector[h]=1;
31          }
32      }
33  }
```

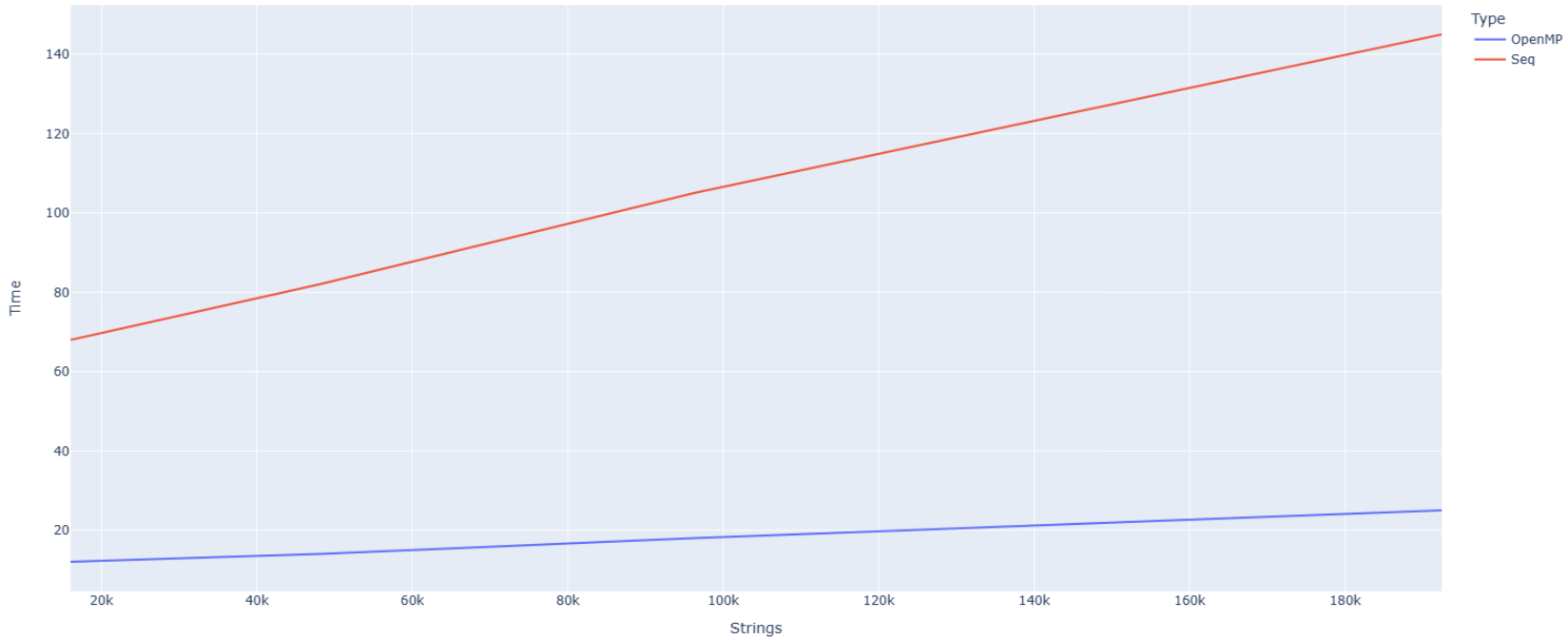
Experimental setup

- CPU Intel i7-10750H;
- The eligible set consists of 192425 strings;
- The set of strings to be checked is a subset of the admissible set;
- For the time measurements we have used **system_clock** from the **chrono** header.

UNIVERSITÀ DEGLI STUDI FIRENZE

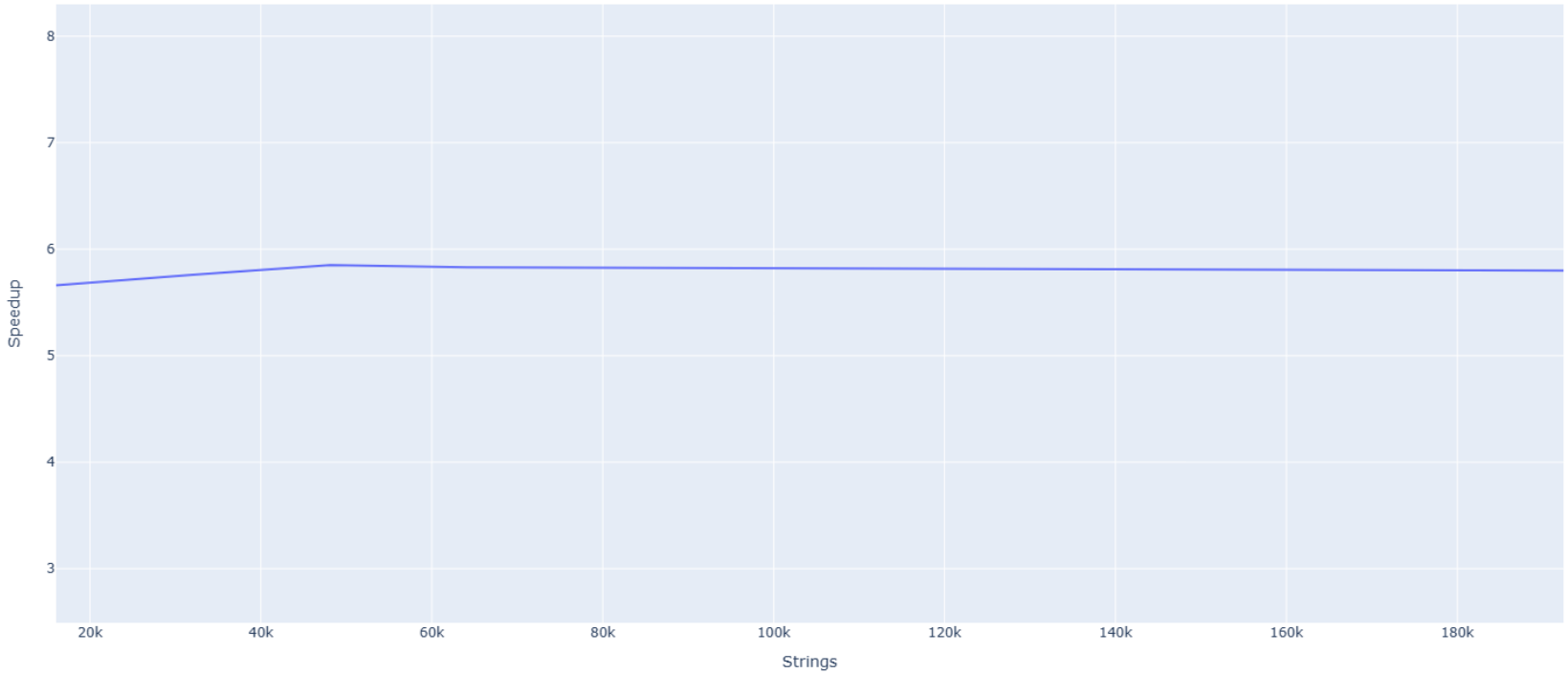
Results of the comparison

The impactful element from the point of view of workload is the number of strings to work on. Therefore, the performance comparison was performed as this value varied.



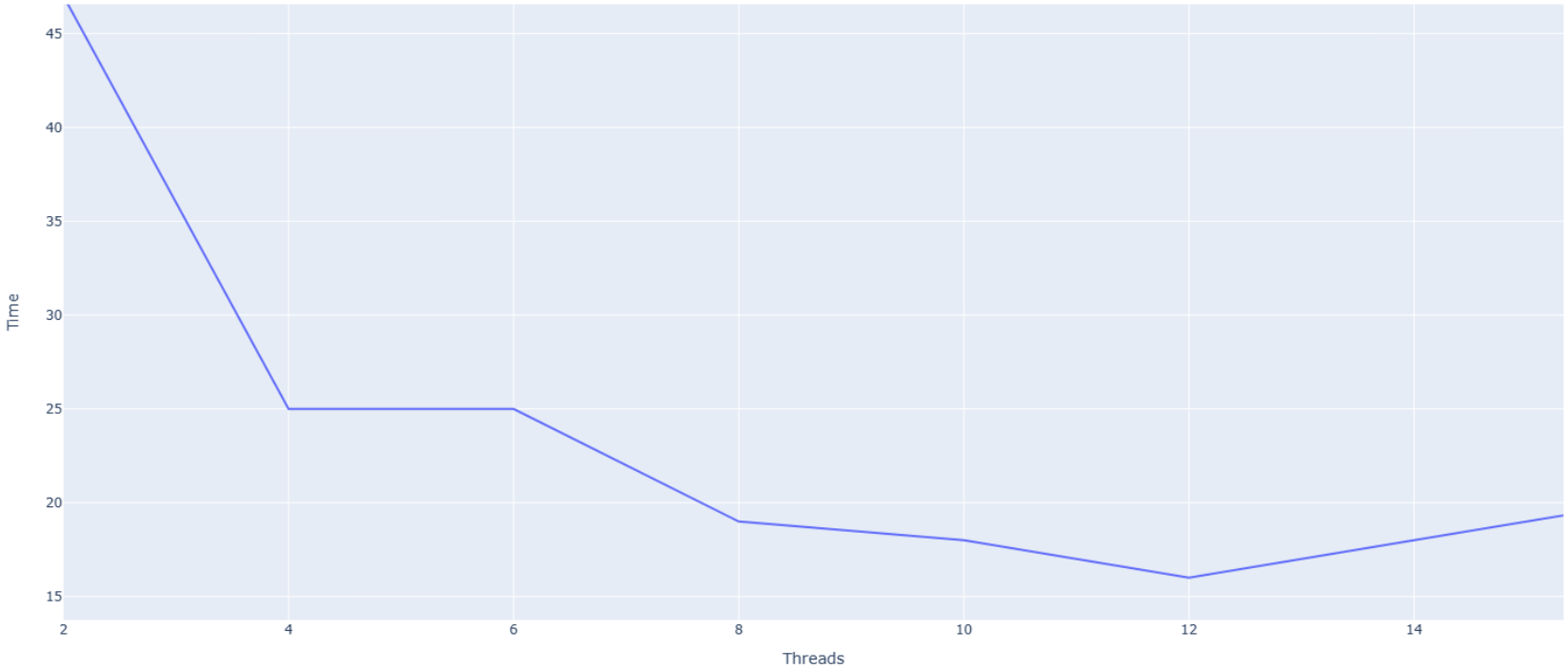
UNIVERSITÀ DEGLI STUDI FIRENZE

Results of the comparison



UNIVERSITÀ DEGLI STUDI FIRENZE

Results of the comparison



Conclusion

- This discussion looked at the comparison of two different implementations of Bloom Filter: sequential and OpenMP;
- We've seen that the parallel version outperforms the sequential version, as indeed the nature of the problem suggested.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Grazie per l’attenzione!