Parallel implementation of BloomFilter using OpenMP

Leonardo Di Iorio
E-mail address
leonardo.diiorio@stud.unifi.it

Abstract

In the following paper the Bloom Filter data structure will be analysed. Two different implementation will be shown: a sequential implementation of the code and a parallel implementation on a Intel i7-10750H CPU using the OpenMP framework. The goal of this paper is to show the improvements in terms of performance that can be obtained through code parallelization.

1. Introduction

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 now turn to expose the structure of the filter.
We can divide the operation into two steps:

- **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 which maps an element of the set to one of the `m` positions in the array with a uniform distribution. 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 our 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 chinks to the set of allowed elements. 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. If even one of the checked bits in the array is 0 then it can be said with certainty that the element is not within the set. If, on the other hand, all of the checked bits are set to 1 then it means that either the element is within the set or the bits were set to 1 during the insertion of other elements, thus returning a false positive.

2. Implementation

In this section the two different implementation of the filter will be shown.

2.1. Sequential

C++ is used to implement the code's sequential version. Despite this, the code is written to be cache-friendly rather than using an object-oriented programming methodology. This is actually a crucial step to do in order to avoid execution slowdowns and to produce code that is as similar as possible between the sequential and parallel versions. So only one class was implemented, called `BloomFilter` with three different methods:

- The constructor;
- The `computeBloomFilter` method, which implements the ; inizialization step

described above;

- The `checkStream` method, which implements the search step described above.

The constructor arguments are defined as follows: the number of bits composing the filter, the number of hash functions to use, the vector composed by admissible strings, the bit vector and the output vector. The number of bits composing the filter and the number of hash functions to use are computed through two functions, `getNumberOfBits` and `getHashNumber`, which return the optimal value of these parameters given the size of the admissible set. 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`, which is a non-cryptographic hash function. We now turn to show the implementation of the two methods described above. The figure below shows the code for the `computeBloomFilter` method. It's possible to see that for every element of the vector of legal strings computes the `k` hash functions and set to 1 int the bit vector the bit corresponding to the value returned by every hash function.

```
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  }
```

Figure 1. Implementation of *computeBloomFilter* method

Figure 2 shows the code of the `checkStream` method, which implements the search step. Similarly to the previous code, for every string to check all the hash functions are computed. If one of the bits corresponding to the values returned by the hash functions is found to be 0, then the string under observation is not admissible and the corresponding bit in the output vector is set to 0.

2.2. OpenMP

OpenMP is an API (application program interface) for writing parallel shared-memory applications. The parts of the implemented code that need to be parallelized in order to achieve performance improvement are the external for loops both those in the `computeBloomFilter` method both those in the `checkStream` method. In both cases the parallelization of the loops has been obtained through the following directive:

```
#pragma omp parallel for
```

For demonstration purposes, the code for the openMP version of the `computeBloomFilter` method is shown in the figure below. It's possible to see the directive mentioned above at line 26:

```
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  }
```

Figure 3. *computeBloomFilter* method with OpenMP directive

```
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  }
```

Figure 2. Implementation of *checkStream* method

3. Performance

This section will show the performance of the two different implementations. The two performances will be compared in order to make it easier to visualize the improvements achieved through code parallelization

3.1. Experimental setup

The CPU used during the experimentation is an Intel i7-10750H. As anticipated, the constituent elements of the eligible set and the set to be checked are strings. The eligible set consists of 192425 strings. The set of strings to be checked is a subset of the admissible set and it's size was varied during experimentation in order to evaluate the performance of the two implementations under varying workloads. For the time measurements we have used `system_clock` from the `<chrono>` header.

3.2. Results of the comparison

As mentioned above, the impactful element from the point of view of workload is the number of strings to work on. Figure 4 then shows the comparison between the sequential and OpenMP versions of the code as the number of strings changes. Performance is evaluated here with respect to total completion time:

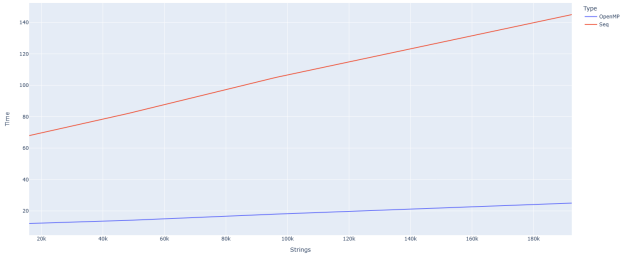

Figure 4. Comparison between the sequential and parallel implementation. On x axis the number of strings, on y axis the time.

It's evident that the parallel version greatly improves performance over the sequential version. The magnitude of this improvement is even clearer in the graph shown in Figure 5. This shows the speedup achieved with the OpenMP version. The speedup is the speed increase when going from a sequential version to a parallel one.

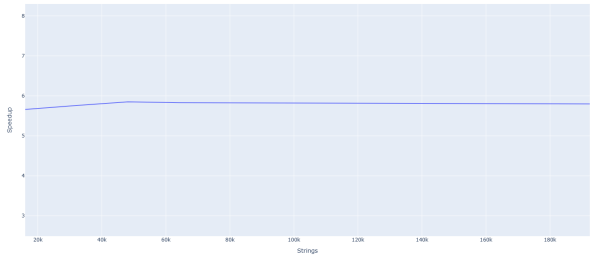

Figure 5. Speedup obtained with the parallel implementation. On x axis the number of strings, on y axis the speedup.

Finally we show in figure 6 how the execution time of the OpenMP version varies with the number of threads:

4. Conclusions

This discussion looked at the comparison of two different implementations of Bloom Filter: sequential and OpenMP. It is clear from the results shown in the previous section how the parallel version outperforms the sequential version, as indeed the nature of the problem suggested.

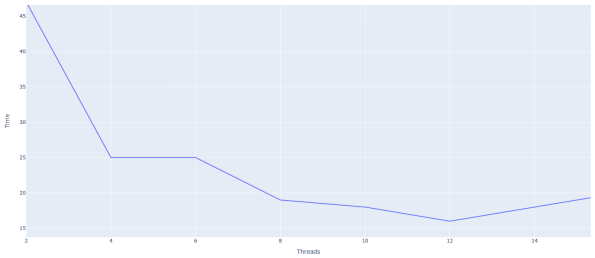

Figure 6. On x axis the number of thread and on y axis the time.