Parallel Computing: Implementation of Bloom Filter using Joblib

Annalisa Ciuchi
E-mail address
annalisa.ciuchi@edu.unifi.it

Abstract

Bloom filters are probabilistic data structures, widely utilized in scenarios where memory efficiency is critical. In this project, the implementation and comparison of a sequential Bloom filter with a parallelized version using joblib is presented. The evaluation of the performance gains achieved through parallelization is analyzing both insertion and lookup times. The experimental results demonstrate the effectiveness of parallel computing in optimizing the Bloom filter.

1. Introduction to Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure that allows fast membership queries, with the trade-off of potential false positives.
The idea behind a Bloom filter is to use multiple hash functions to map elements into a fixed-size bit array. When inserting an element, the corresponding bit positions derived from the hash functions are set to 1. To check if an element is present, the same hash functions are applied, and if all respective bits are set to 1, the element is considered present (though false positives can occur).

2. Sequential Implementation

The sequential Bloom filter implementation follows a straightforward approach:

1. It initializes a bit array of a fixed size.
2. During insertion, it computes multiple hash values for each element and updates the corresponding bit positions.
3. During lookup, it checks whether all bit positions derived from the element's hash values are set to 1.

The code implementation iterates through the given elements, performing insertions and lookups in a single-threaded manner, as shown in `sequential_BloomFilter.py`.
The implementation in `sequential_BloomFilter.py` is structured as follows

```python
import hashlib

def calcola_hash(string, n, j):
    sha256 = hashlib.sha256()
    sha256.update((string + str(j)).encode('utf-8'))
    return int(sha256.hexdigest(), 16) % n

class BloomFilter:
    def __init__(self, dimensione, numero_hash):
        self.array_bit = [0] * dimensione
        self.dimensione = dimensione
        self.numero_hash = numero_hash

    def inizializza(self, elementi):
        for j in range(self.numero_hash):
            for i in range(len(elementi)):
                self.array_bit[calcola_hash(elementi[i],
   self.dimensione, j)] = 1

    def verifica(self, elemento):
        for i in range(self.numero_hash):
            if self.array_bit[calcola_hash(elemento, self.dimensione,
   i)] == 0:
                return False
        return True
```

Listing 1. Sequential Bloom Filter Code

The `inizializza` method iterates through the dataset, applying multiple hash functions and updating the bit array. The `verifica` method checks membership by ensuring all corresponding bits are set to 1. While simple and effective, this approach can be computationally expensive when dealing with large datasets.

3. Parallel Implementation

To improve performance, an implementation of a parallelized version of the Bloom filter using Python's joblib library (`parallel_BloomFilter.py`) is presented. The key optimizations include:

1. Parallelized hash computation: Instead of computing hashes sequentially, hash computations are distributed across multiple threads using `joblib.Parallel`. This allows multiple elements to be processed simultaneously. Once the hashes are computed, the bit array is updated in a single batch operation to minimize memory access overhead. Directly parallelizing the insertion process would introduce memory contention and synchronization overhead (to avoid race condition) so that could negate any performance gains.

2. Parallelized lookup: The verification process is parallelized by applying `joblib.Parallel` to check multiple elements concurrently, reducing lookup time.

3. Use of NumPy for efficiency: The bit array is implemented using NumPy to allow efficient memory operations.

The implementation is as follows:

```
from joblib import Parallel, delayed
import hashlib
import numpy as np

def calcola_hash(stringa, n, j):
    sha256 = hashlib.sha256()
    sha256.update((stringa + str(j)).encode('utf-8'))
    return int(sha256.hexdigest(), 16) % n

class BloomFilterParallelo:
    def __init__(self, dimensione, numero_hash, numero_thread):
        self.array_bit = np.zeros(dimensione, dtype=np.uint8)
        self.dimensione = dimensione
        self.numero_hash = numero_hash
        self.numero_thread = numero_thread

    def inizializza(self, elementi):
        risultati = Parallel(n_jobs=self.numero_thread, backend="loky")(
            delayed(lambda el: [calcola_hash(el, self.dimensione, j) for
      j in range(self.numero_hash)])(el)
            for el in elementi
        )
        for hash_list in risultati:
            np.put(self.array_bit, hash_list, 1)

    def verifica(self, elemento):
        return all(self.array_bit[calcola_hash(elemento,
      self.dimensione, j)] for j in range(self.numero_hash))

    def verifica_parallela(self, elementi):
        return Parallel(n_jobs=self.numero_thread, backend="loky")(
            delayed(self.verifica)(e) for e in elementi
        )
```

Listing 2. Parallel Bloom Filter Code

This implementation leverages joblib's Parallel to distribute the computation of hash functions and lookup operations across multiple CPU cores, reducing the overall execution time. Specifically, in the `inizializza` method, multiple hash values for each element are computed concurrently using:

```
risultati = Parallel(n_jobs=self.numero_thread, backend="loky")(
    delayed(lambda el: [calcola_hash(el, self.dimensione, j) for j in
      range(self.numero_hash)])(el)
    for el in elementi
)
```

Listing 3. Parallel Bloom Filter Code

This allows each element's hash calculations to be processed in parallel rather than sequentially, speeding up the initialization phase. Similarly, the lookup process benefits from parallelization via:

```
return Parallel(n_jobs=self.numero_thread, backend="loky")(
    delayed(self.verifica)(e) for e in elementi
)
```

Listing 4. Parallel Bloom Filter Code

which enables simultaneous membership checks for multiple elements, reducing the verification time. The use of loky as the backend ensures that computations are efficiently executed across multiple CPU cores. This parallelization is particularly beneficial for large datasets, as it eliminates bottlenecks caused by sequential processing, making the Bloom filter significantly more scalable and efficient.

4. Tests and Results

The experiment were conducted on an Intel i9-13900HX CPU and the Bloom Filter was configured to store randomly generated email addresses, which simulate potentially malicious email addresses for a spam detection system.

The verification phase was performed by selecting a subset of stored addresses and introducing additional randomly generated addresses to simulate a mix of known and unknown elements.

The performance was measured in terms of:

- Initialization Time: Time required to construct the Bloom Filter with the given elements and hash functions.
- Verification Time: Time required to check whether a given set of email addresses is present in the filter.

4.1. Initialization times

#Threads:		2	4	8	16
#elements x #hash	Seq (s)	Par (s)	Par (s)	Par (s)	Par (s)
100000x50	3,7807	2,1250	1,2677	0,9454	0,8191
100000x75	6,7890	2,9518	1,8096	1,5310	1,4674
100000x100	11,7187	3,8546	2,4109	2,2297	1,8371

The initialization time decreases significantly when increasing the number of threads from 1 to 8. However, at 16 threads, the speedup gain is minimal, indicating diminishing returns. In fact if the number of threads increases too much, the overhead of managing them begins to offset the computational gains. Performance also improves as we increase the number of hash functions to be performed and this is expected as we increase the workload to be done. The maximum speedup here is 5,26 and it's obtained with 8 threads for 100000 elements x 50 hash.

4.2. Verification times

#Threads:		2	4	8	16
#hash x #elements to verify	Seq (s)	Par (s)	Par (s)	Par (s)	Par (s)
50x50000	1,6414	1,0426	0,6173	0,5311	0,6821
75x50000	3,0968	1,5057	0,8824	0,8746	1,0021
100x50000	5,6425	1,9864	1,1626	1,0955	1,2009

The verification phase also exhibits a strong speedup up to 8 threads. However, at 16 threads, the performance slightly worsens rather than improving further. This phenomenon is due to the diminishing workload per thread: As the workload is split among more threads, the amount of work assigned to each thread decreases and, at some point, the overhead of thread management outweighs the performance benefit. Here the maximum speedup is 5,15 and it's obtained with 8 threads for 100 hash x 50000 elements to verify.

4.3. Increasing number of elements

The graph illustrates the initialization time of a Bloom Filter for two configurations (50 and 75 hash) where the number of elements has been increased compared to previous tests. For both configurations, the execution time drops sharply when increasing the number of threads from 1 to 4, and continues to decrease. After 8 threads, the performance improvement slows down, and at 16 threads, the initialization time remains nearly constant.

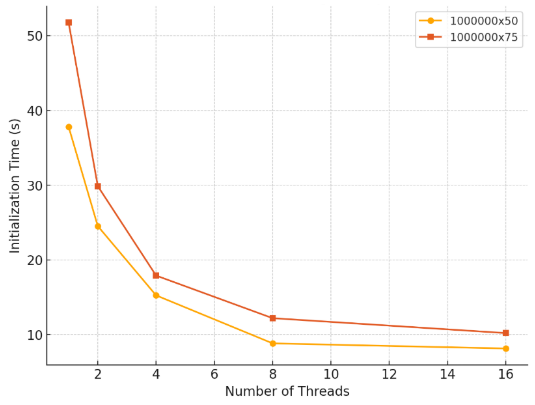

Figure 1.

The following graph represents the verification time for two different configurations of Bloom Filters:

- 100000x50x75000 (100,000 elements, 50 hash functions, 75,000 verification checks)
- 100000x50x100000 (100,000 elements, 50 hash functions, 100,000 verification checks)

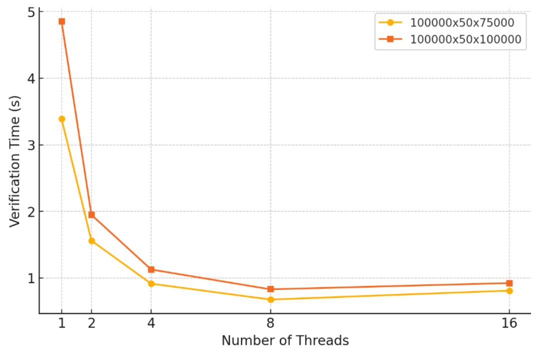

Figure 2.

Both configurations exhibit a significant reduction in verification time as the number of threads increases. The transition from 1 to 4 threads shows a steep decline, and, at 8 threads, the verification time reaches its lowest. At 16 threads, the performance slightly worsens rather than improving further and this is expected as the workload per thread decreases (at some point, the additional threads do not provide enough computational work to justify their overhead).

4.4. Initialization and verification times

This table provides the sum of initialization and verification times for each configuration across different thread counts. These results indicate good parallel scalability up to 8 threads, but diminishing returns beyond that. Higher hash configurations (100000x100) scale better, benefiting from higher per-thread workload and better CPU utilization.

Threads	100000x50	100000x75	100000x100
1	5.4221	9.8858	17.3612
2	3.1676	4.4575	5.8410
4	1.8850	2.6920	3.5735
8	1.4765	2.4056	3.3252
16	1.5012	2.4695	3.0380

Table 1. Total execution time (Initialization + Verification)