

Parallelizzazione di un BloomFilter per la  
classificazione di email di spam  
Parallel Programming for Machine Learning

Lorenzo Baiardi, Thomas Del Moro

GG MM AAAA

## 1 Introduzione

Il nostro progetto si concentra sulla parallelizzazione di un BloomFilter impiegato per la classificazione di email considerate spam. Per raggiungere questo obiettivo, utilizziamo le librerie Omp e Joblib per implementare il parallelismo.

Le fasi che intendiamo parallelizzare sono il setup iniziale e la fase di filtraggio. Vogliamo analizzare come lo speedup varia in relazione alle dimensioni dell'insieme di dati utilizzato per il training. Allo stesso modo, intendiamo valutare l'effetto dello speedup in funzione del numero di processi impiegati, delle dimensioni del dataset e delle diverse implementazioni utilizzate.

## 2 Analisi del problema

Il BloomFilter rappresenta una struttura dati probabilistica finalizzata a determinare la presenza di un elemento all'interno di un insieme, nel contesto specifico, se una determinata email è considerata spam o meno. La fase iniziale di configurazione del BloomFilter coinvolge il calcolo delle funzioni hash e la creazione del vettore di bit. Quest'ultima operazione si basa sulla dimensione dell'insieme utilizzato per il training e sulla probabilità attesa di ottenere falsi positivi. La formula per il calcolo della dimensione del vettore di bit è la seguente:

$$size = -\frac{n \ln p}{(\ln 2)^2} \quad (1)$$

Dove  $size$  è la dimensione del vettore di bit per il training e  $n$  è la dimensione dell'insieme che si vuole utilizzare.

La formula per il calcolo del numero di funzioni hash è la seguente:

$$h = \frac{size}{n} \ln 2 \quad (2)$$

Dove  $h$  è il numero di funzioni hash da utilizzare per il training.

Dopo aver fornito al BloomFilter il dataset di training, quest'ultimo procede al calcolo delle  $h$  funzioni hash per ciascuna email, impostando a 1 i bit nelle posizioni calcolate. Per determinare se un'email è classificata come spam o meno, il BloomFilter esegue nuovamente il calcolo delle  $h$  funzioni hash e verifica se i bit corrispondenti alle posizioni calcolate sono settati a 1.

### 3 Parallelizzazione

Il codice sequenziale di riferimento da parallelizzare per la fase di setup è il seguente, rispettivamente per la versione c++ e python:

```
1 double BloomFilter::sequentialSetup(std::string items[], std::  
    size_t nItems) {  
2     initialize(nItems);  
3     double start = omp_get_wtime();  
4     for(std::size_t i=0; i < nItems; i++)  
5         add(items[i]);  
6     return omp_get_wtime() - start;  
7 }
```

```
1     def seq_setup(self, items):  
2         self.initialize(items)  
3  
4         # Start sequential setup  
5         start = time.time()  
6         self.add(items)  
7         return time.time() - start
```

Il codice sequenziale di riferimento da parallelizzare per la fase di filter è il seguente, rispettivamente per la versione c++ e python:

```
1 int BloomFilter::sequentialFilterAll(std::string items[],  
    size_t nItems) {  
2     int error = 0;  
3     for(std::size_t i=0; i < nItems; i++)  
4         if(filter(items[i]))  
5             error++;  
6     return error;  
7 }
```

```
1     def seq_filter_all(self, items):  
2         errors = 0  
3  
4         # Start sequential filter  
5         start = time.time()  
6         for item in items:  
7             if self.filter(item):  
8                 errors += 1  
9         return time.time() - start, errors
```

#### 3.1 OpenMP

OpenMP (Omp) è una libreria in linguaggio C progettata per consentire la parallelizzazione di funzioni e cicli for. Nel contesto di questo lavoro, abbi-

amo adottato la funzione `omp parallel for` per parallelizzare le operazioni di setup e filtraggio del BloomFilter.

Successivamente, abbiamo analizzato il tempo impiegato dalle funzioni di setup e filtraggio in relazione al numero di processi utilizzati, confrontandolo con il tempo richiesto nella versione sequenziale.

```
1 double BloomFilter::parallelSetup(std::string items[], std::
  size_t nItems) {
2   initialize(nItems);
3   double start = omp_get_wtime();
4   #pragma omp parallel default(none) shared(bits, items)
    firstprivate(nItems, nHashes)
5   {
6   #pragma omp for
7       for(std::size_t i=0; i < nItems; i++) {
8           MultiHashes mh(this->size, items[i]);
9           for (std::size_t h = 0; h < nHashes; h++) {
10              std::size_t index = mh();
11   #pragma omp critical
12              this->bits[index] = true;
13          }
14      }
15  }
16  #pragma omp barrier
17  return omp_get_wtime() - start;
18 }
```

```
1 int BloomFilter::parallelFilterAll(std::string items[], size_t
  nItems) {
2   int error = 0;
3   #pragma omp parallel default(none) shared(items, error)
    firstprivate(nItems)
4   {
5   #pragma omp for
6       for (std::size_t i = 0; i < nItems; i++) {
7           if (filter(items[i]))
8   #pragma omp atomic
9               error++;
10      }
11  }
12  return error;
13 }
```

### 3.2 Joblib

Joblib è una libreria Python progettata per consentire la parallelizzazione di funzioni e cicli for. La funzione `Parallel` accetta come input il numero

di processori da impiegare e la funzione da parallelizzare. Nell'ambito di questa ricerca, abbiamo adoperato la funzione `Parallel` per parallelizzare le operazioni di setup e filtraggio del BloomFilter.

Successivamente, abbiamo analizzato il tempo richiesto dalle funzioni di setup e filtraggio in relazione al numero di processi utilizzati, confrontandolo con il tempo necessario nella versione sequenziale.

```
1  def par_setup(self, items, n_threads, chunks=None):
2      self.initialize(items)
3
4      # Split items in chunks
5      chunks = np.array_split(items, chunks if chunks else
6                               n_threads*2) # *4
7
8      # Start parallel setup
9      start = time.time()
10     Parallel(n_jobs=n_threads)(delayed(self.add)(chunk) for
11                                chunk in chunks)
12     return time.time() - start
13
14
15 def par_filter_all(self, items, n_threads):
16     # Split items in chunks
17     chunks = np.array_split(items, n_threads)
18
19     # Start parallel setup
20     start = time.time()
21     results = Parallel(n_jobs=n_threads)(delayed(self.
22 seq_filter_all)(chunk) for chunk in chunks)
23     end_time = time.time() - start
24
25     # Sum errors
26     t, errs = zip(*results)
27     errors = sum(errs)
28     return end_time, errors
```

## 4 Caratteristiche della macchina

La macchina utilizzata per i test ha le seguenti caratteristiche:

- **CPU:** Intel Core i7-1360P (4 P-Core, 8 E-Core, 12 Cores, 16 Threads)
- **RAM:** 16GB
- **Sistema Operativo:** Windows 11

## 5 Test

I test sono stati effettuati su un dataset da 10000 fino a 10000000 email, con una probabilità di falsi positivi di 0.10, 0.05 e 0.01. Le email sono state generate casualmente tramite le funzioni contenute

### 5.1 FPR: 0.05

Poiché i test variano da 10000 a 10000000 email, il numero di funzioni hash e la dimensione del vettore di bit sono rispettivamente 5, e da un minimo di 62353 a un massimo di 62352243.

#### 5.1.1 Setup

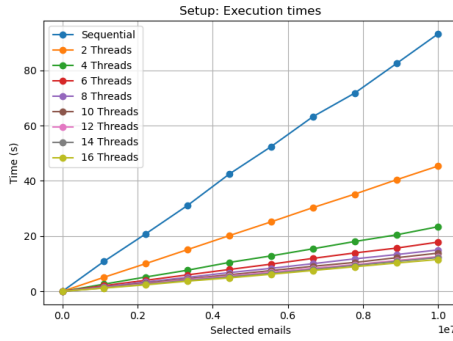


Figure 1: Time setup Omp

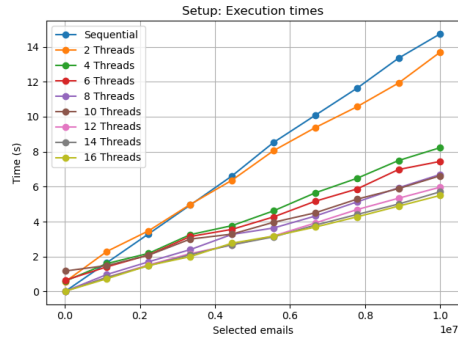


Figure 2: Time setup Joblib

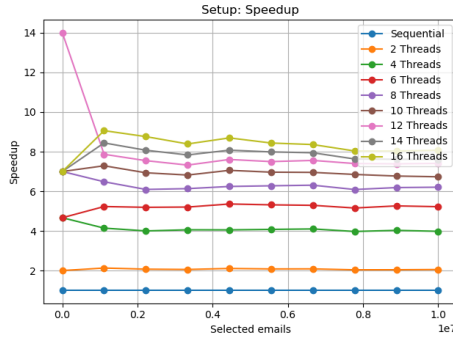


Figure 3: Speedup setup Omp

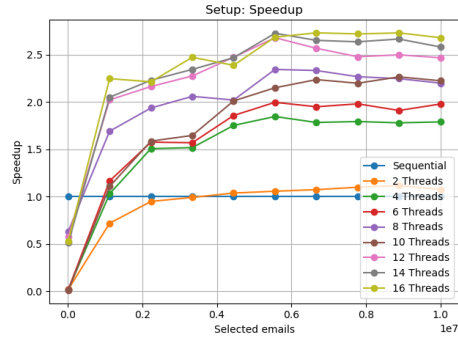


Figure 4: Speedup setup Joblib

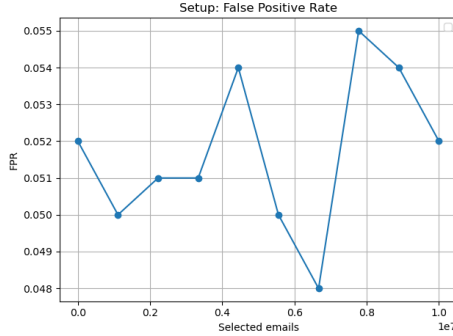


Figure 5: FPR setup Omp

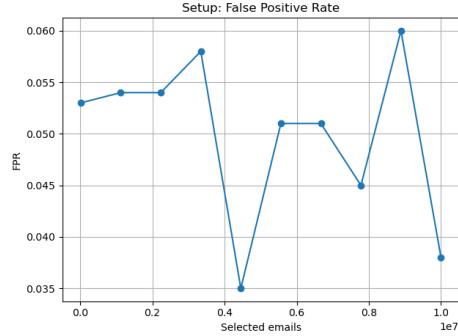


Figure 6: FPR setup Joblib

Per il test di configurazione iniziale, è evidente che la versione sviluppata con OpenMP presenta una performance temporale inferiore rispetto alla controparte. Al contrario, in termini di speedup, la versione OpenMP supera la versione sviluppata con Joblib, raggiungendo uno speedup massimo di 2.70 nella versione Joblib e 9 nella versione OpenMP, utilizzando il massimo numero di thread disponibili.

I risultati massimi sono stati ottenuti sfruttando il numero massimo di thread disponibili. Nel contesto della versione Joblib, i test iniziali con un numero più elevato di thread mostrano un tempo di esecuzione peggiore rispetto alla versione sequenziale, probabilmente a causa del tempo di inizializzazione dei thread. Nonostante l'aumento del numero di thread nella versione Joblib, lo speedup si stabilizza intorno al valore di 2.

Va notato che i valori di FPR (False Positive Rate) sono molto simili tra le due versioni.

### 5.1.2 Filter

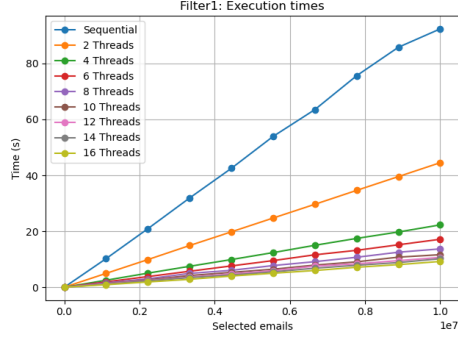


Figure 7: Time Filter Omp

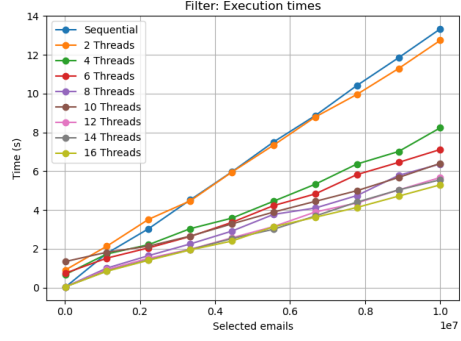


Figure 8: Time Filter Joblib

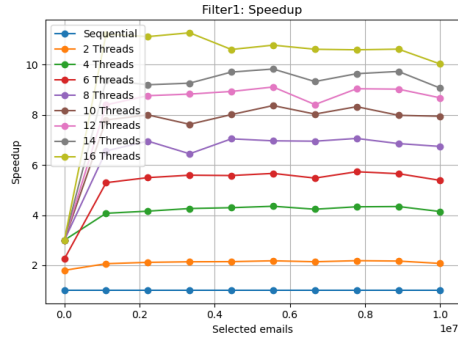


Figure 9: Speedup Filter Omp

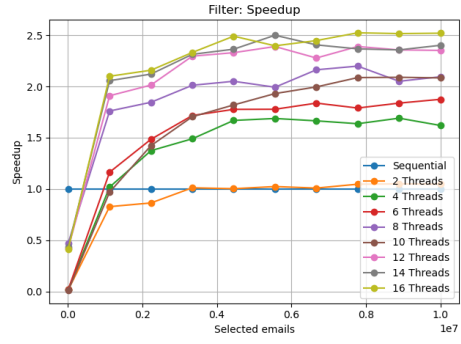


Figure 10: Speedup Filter Joblib

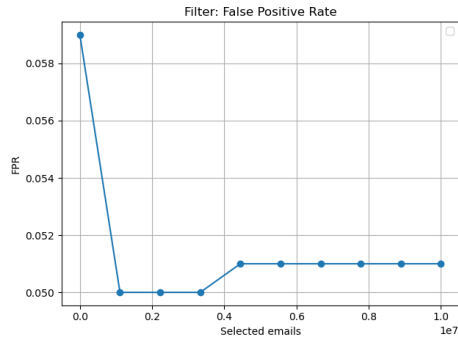


Figure 11: FPR Filter Omp

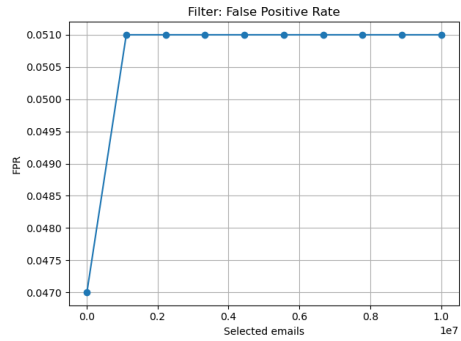


Figure 12: FPR Filter Joblib



Per il test di filtraggio, è evidente che la versione sviluppata con OpenMP presenta una performance temporale inferiore rispetto alla sua controparte. Tuttavia, in termini di speedup, la versione OpenMP supera la versione sviluppata con Joblib, raggiungendo un massimo di 11 nella versione OpenMP e 2.5 nella versione Joblib, sfruttando il massimo numero di thread disponibili.

Anche in questo contesto, i valori di FPR (False Positive Rate) sono praticamente identici tra le due versioni.

### 5.1.3 Chunks

Esploriamo ora l'opzione di eseguire un'operazione di chunking più ampia rispetto al numero di thread disponibili per verificare se è possibile migliorare le performance, focalizzandoci esclusivamente sulla versione Joblib. I valori di riferimento per i chunks sono 16, 32, 64, 128, 256, 512, 1024, 2048.

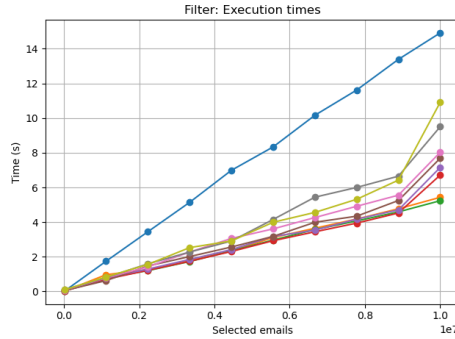


Figure 13: Times setup Chunks

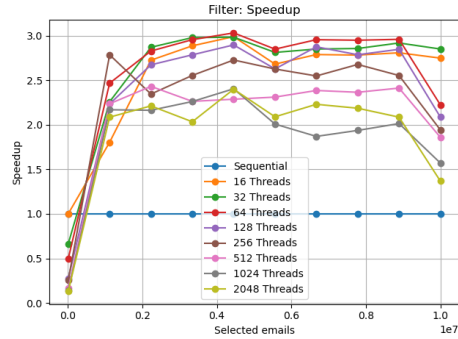


Figure 14: Speedup setup Chunks

I risultati ottenuti indicano che l'applicazione dell'operazione di chunking ha determinato miglioramenti significativi nelle performance di speedup. In particolare, l'aumento del numero di chunk ha contribuito a un miglioramento dello speedup, raggiungendo un massimo di 3. Tuttavia, oltre una soglia di 64 chunk, si è verificato un declino nelle performance.

## 6 Conclusioni

I risultati ottenuti mostrano che la parallelizzazione delle operazioni di setup e filtraggio del BloomFilter consente di ottenere uno speedup significativo in relazione al numero di processori impiegati. Specificamente, in termini di tempo, la versione parallelizzata con Joblib è risultata più efficiente rispetto alla versione parallelizzata con Omp. Viceversa, la versione parallelizzata con Omp ha mostrato un miglioramento più significativo in termini di speedup. L'operazione di chunking ha permesso di ottenere uno speedup leggermente migliore rispetto alla versione senza chunking per alcuni valori di chunk size.

## 7 Appendice

### 7.1 FPR: 0.01

Vediamo adesso se aumentando la precisione del filtro, i risultati ottenuti precedentemente cambiano.

#### 7.1.1 Setup

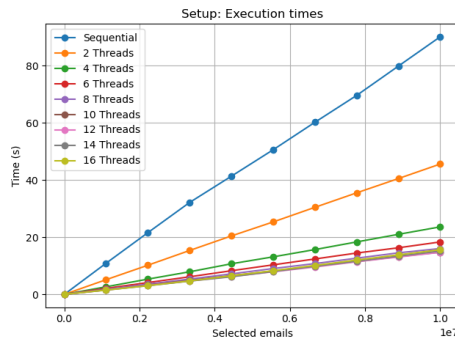


Figure 15: Speedup setup Omp

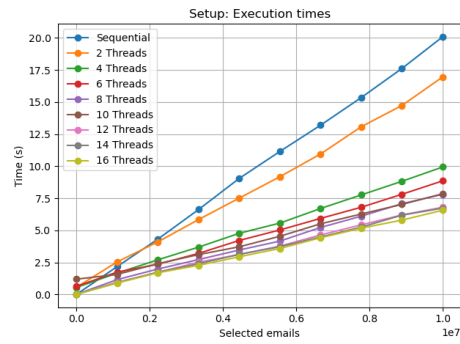


Figure 16: Speedup setup Joblib

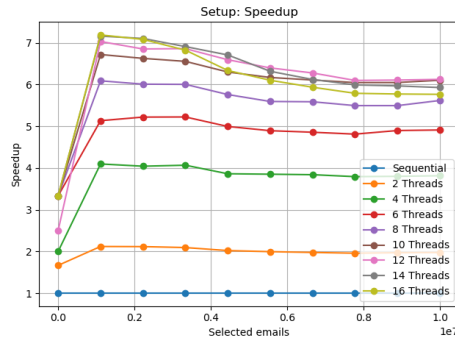


Figure 17: Speedup setup Omp

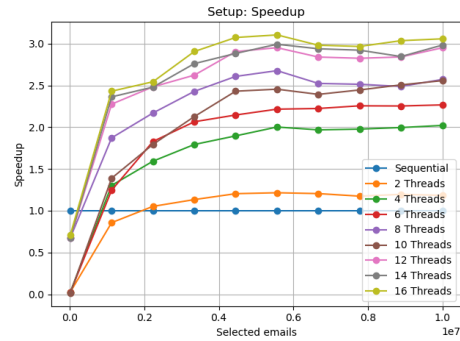


Figure 18: Speedup setup Joblib

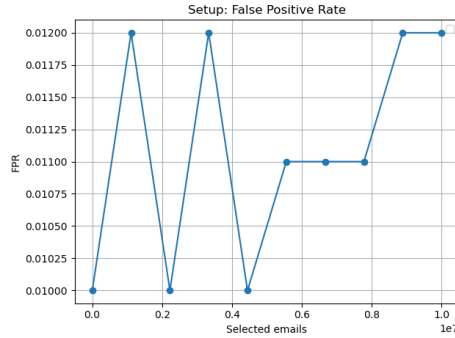


Figure 19: Speedup setup Omp

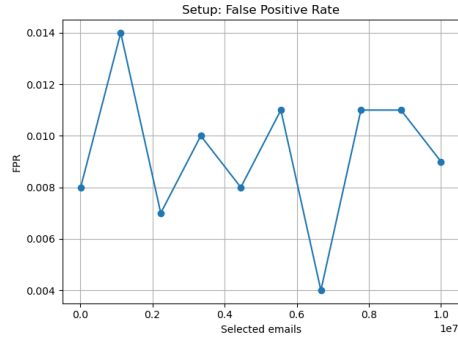


Figure 20: Speedup setup Joblib

In questo contesto, nella fase di setup, i risultati ottenuti indicano un aumento delle tempistiche di esecuzione per la versione Joblib rispetto a  $FP=0.05$ , tuttavia si osserva un miglioramento in termini di speedup, che raggiunge un massimo di 3.1. Al contrario, la versione OpenMP manifesta un peggioramento nelle performance di speedup, raggiungendo un massimo di 7.1 con il massimo numero di thread disponibili. Anche in questo caso, i valori di FPR (False Positive Rate) sono molto simili tra le due versioni, oscillando intorno al valore di 0.01.

### 7.1.2 Filter

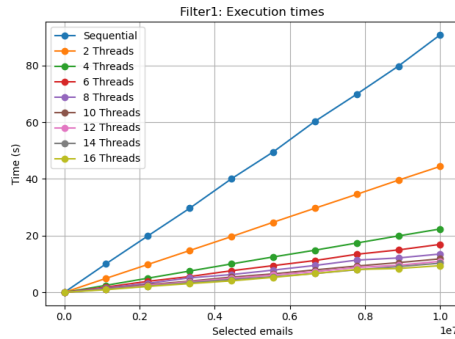


Figure 21: Times filter Omp

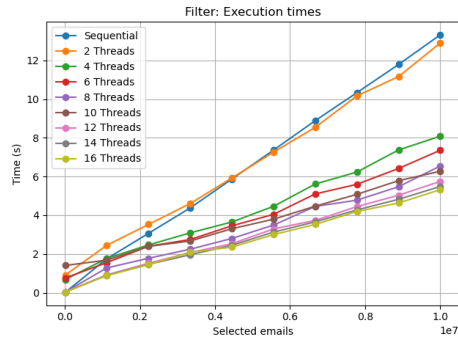


Figure 22: Times filter Joblib

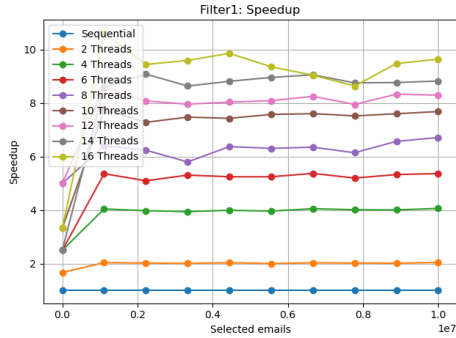


Figure 23: Speedup filter Omp

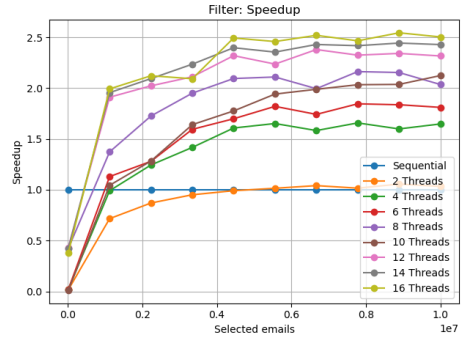


Figure 24: Speedup filter Joblib

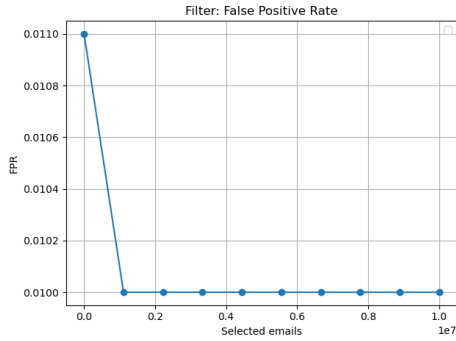


Figure 25: FPR filter Omp

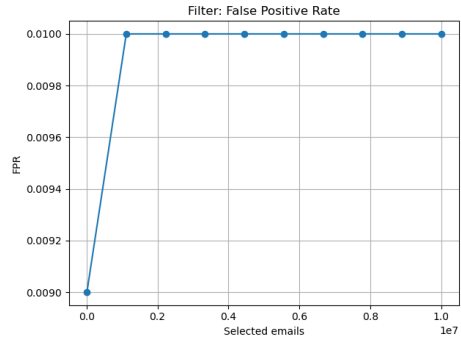


Figure 26: FPR filter Joblib

Con l'aumento della precisione del filtro, i risultati ottenuti indicano un leggero peggioramento delle performance, dovuto al numero maggiore di operazioni da eseguire rispetto a  $FP=0.05$ .

### 7.1.3 Chunks

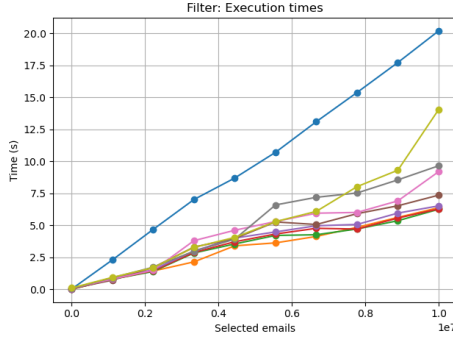


Figure 27: Times chunks Joblib

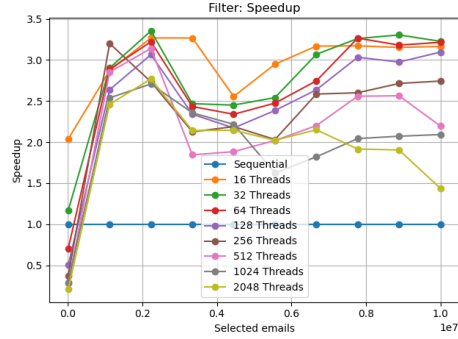


Figure 28: Speedup chunks Joblib

Nella fase di chunking, i risultati ottenuti mostrano un miglioramento delle performance di speedup, raggiungendo un massimo di 3.3. Il valore di soglia per il numero di chunk rimane 64, oltre il quale si verifica un declino nelle performance.

## 7.2 FPR: 0.10

Viceversa vediamo se diminuendo la precisione del filtro, i risultati ottenuti cambiano.

### 7.2.1 Setup

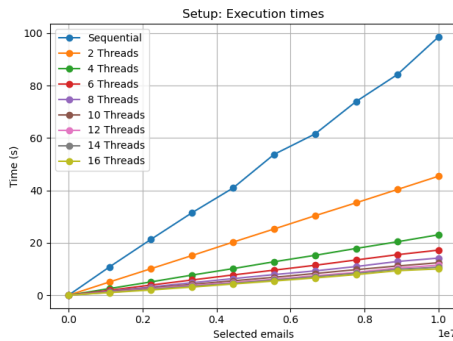


Figure 29: Time setup Omp

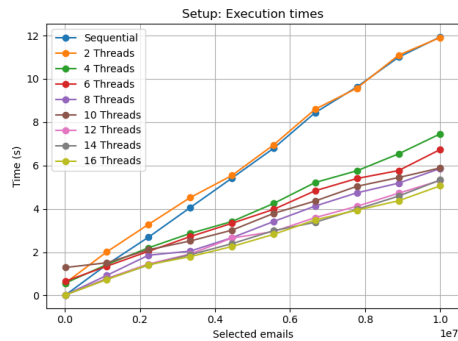


Figure 30: Time setup Joblib

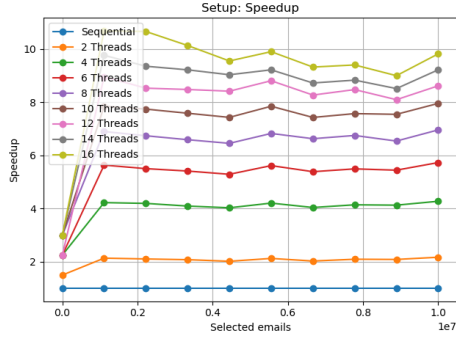


Figure 31: Speedup setup Omp

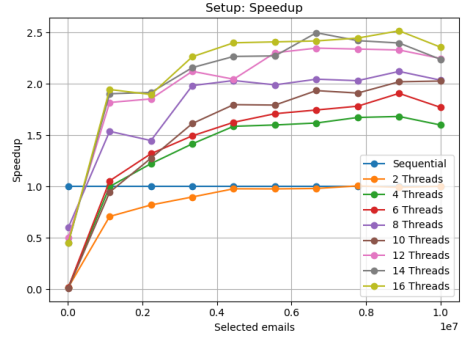


Figure 32: Speedup setup Joblib

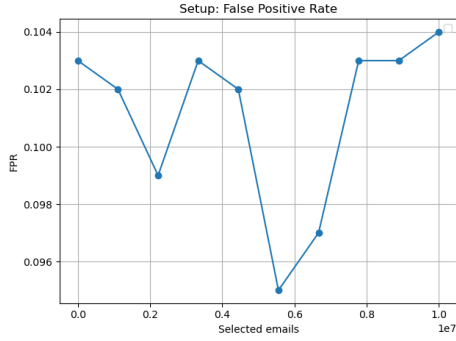


Figure 33: FPR setup Omp



Figure 34: FPR setup Joblib

In termini di tempo nella fase di setup, i risultati ottenuti indicano un peggioramento delle performance per la versione OpenMP, rispetto al valore di  $FPR=0.05$ . La versione Joblib, invece, mostra un peggioramento nelle fasi iniziali del test per poi stabilizzarsi intorno al valore di 2.5 per il massimo numero di thread disponibili.

## 7.2.2 Filter

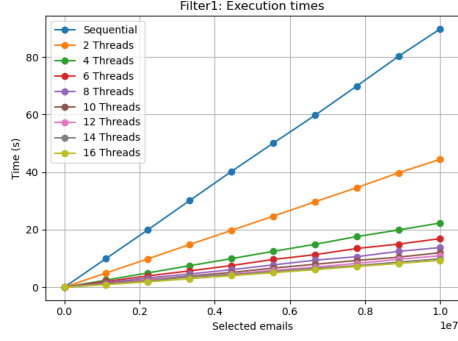


Figure 35: Time setup Omp

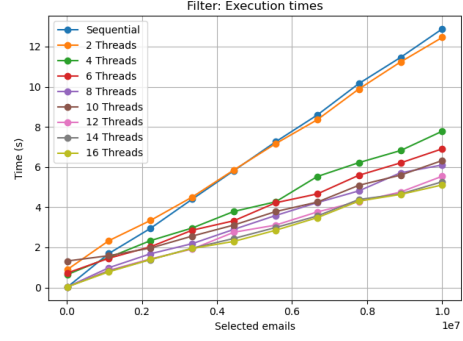


Figure 36: Speedup setup Joblib

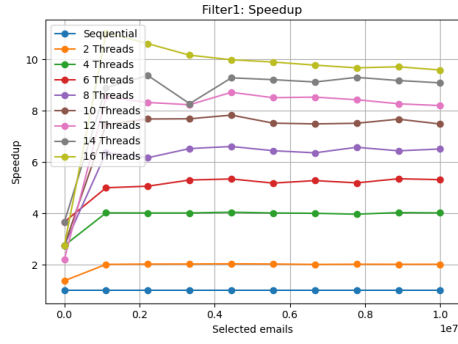


Figure 37: Speedup setup Omp

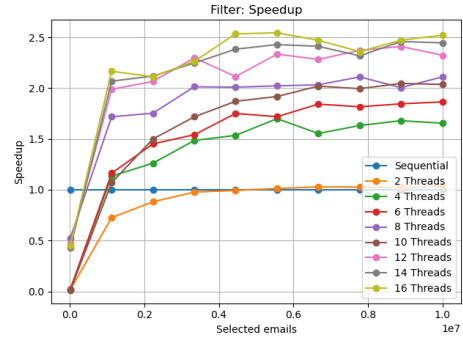


Figure 38: Speedup setup Joblib

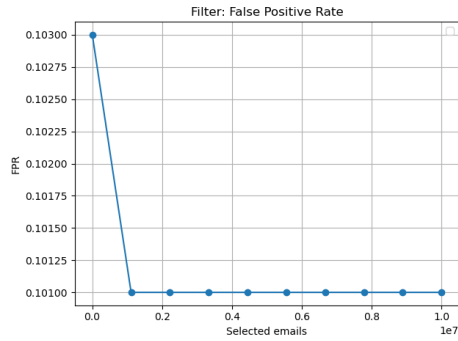


Figure 39: FPR Filter Omp

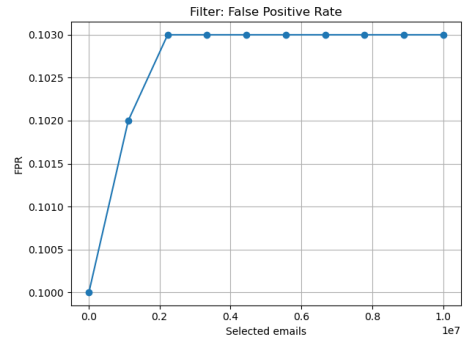


Figure 40: FPR Filter Joblib



Per la fase di filtraggio, i risultati risultano essere molto simili a quelli ottenuti con  $FPR=0.05$ .

### 7.2.3 Chunks

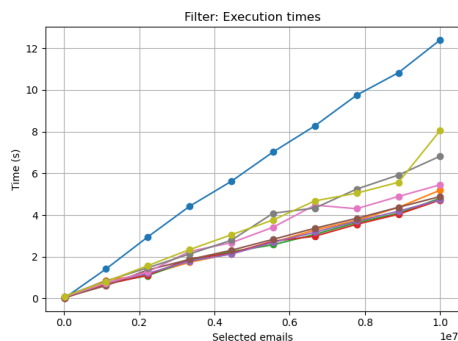


Figure 41: Times setup Chunk

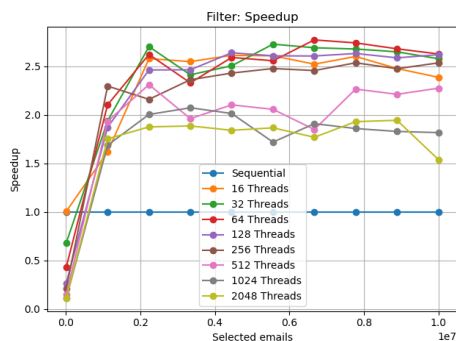


Figure 42: Speedup setup Chunk

Nella fase di chunking, i risultati ottenuti mostrano come il valore di soglia di chunk sia 256, oltre il quale si verifica un declino nelle performance.