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:

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

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:

$$h = \frac{size}{n} \ln 2 \tag{2}$$

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:

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

```
    def seq_setup(self, items):
        self.initialize(items)

        # Start sequential setup
        start = time.time()
        self.add(items)
        return time.time() - start
```

Nella fase sequenziale della configurazione, dopo un'iniziale fase di inizializzazione finalizzata al calcolo del valore ottimale del vettore di bit e del numero di funzioni hash in base al valore di probabilità di falsi positivi, si procede con l'indicizzazione nel vettore di bit per ciascuna email.

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

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

```
    def seq_filter_all(self, items):
        errors = 0

        # Start sequential filter
        start = time.time()
        for item in items:
            if self.filter(item):
                errors += 1
        return time.time() - start, errors
```

Nella fase sequenziale del processo di filtraggio, si inizializza una variabile di conteggio per i falsi positivi, la quale sarà incrementata ogni volta che un'email viene identificata come spam.

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, abbiamo 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     return omp_get_wtime() - start;
17 }
```

Per la fase di setup, abbiamo parallelizzato l'indicizzazione nel vettore di bit per ogni email, utilizzando la direttiva `omp parallel for`, e la direttiva `omp critical` per garantire l'accesso esclusivo al vettore di bit, evitando così l'accesso concorrente.

Per quanto riguarda l'operazione di filtraggio, abbiamo ideato due diverse implementazioni.

```
1  int BloomFilter::parallelFilterAll1(std::string items[], size_t
       nItems) {
2      int error = 0;
3  #pragma omp parallel default(none) shared(items, error)
       firstprivate(nItems)
4      {
```

```
#pragma omp for
        for (std::size_t i = 0; i < nItems; i++) {
            if (filter(items[i]))
#pragma omp atomic
                error++;
        }
    }
    return error;
}
```

La prima realizzazione prevede l'impiego della direttiva `omp parallel for` al fine di parallelizzare il processo di verifica della presenza di un'email all'interno del BloomFilter. In aggiunta, viene utilizzata la direttiva `omp atomic` per assicurare l'accesso esclusivo alla variabile incrementale utilizzata per il conteggio dei falsi positivi.

```
int BloomFilter::parallelFilterAll2(std::string items[], size_t
     nItems) {
    int error = 0;
    int threadError = 0;
#pragma omp parallel default(none) shared(items, error)
    firstprivate(nItems, threadError)
    {
#pragma omp for nowait
        for (std::size_t i = 0; i < nItems; i++) {
            if (filter(items[i]))
                threadError++;
        }
#pragma omp atomic
        error += threadError;
    }
    return error;
}
```

La seconda realizzazione, al contrario, impiega una variabile incrementale dedicata per ciascun thread, e la somma di tali variabili al termine dell'operazione di filtraggio. Inoltre, si fa uso della direttiva `omp atomic` per assicurare l'accesso esclusivo alla variabile incrementale globale durante questa fase. Verificheremo in seguito la differenza tra queste due implementazioni.

3.2 Joblib

Joblib è una libreria Python ideata per abilitare la parallelizzazione di funzioni e cicli for. La funzione `Parallel` prende come input sia il numero di processi da utilizzare che la funzione da parallelizzare. Nell'ambito di questo studio, abbiamo sfruttato la funzione `Parallel` per parallelizzare sia le fasi di preparazione (setup) che quelle di 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.

```
    def par_setup(self, items, n_threads, chunks=None):
        self.initialize(items)

        # Split items in chunks
        chunks = np.array_split(items, chunks if chunks else
    n_threads*2) # *4

        # Start parallel setup
        start = time.time()
        Parallel(n_jobs=n_threads)(delayed(self.add)(chunk) for
     chunk in chunks)
        return time.time() - start
```

Analogamente all'approccio con Omp, abbiamo parallelizzato l'indicizzazione nel vettore di bit per ciascuna email mediante l'utilizzo della funzione `Parallel`, suddividendo il vettore di email in chunk corrispondenti al numero di processi impiegati. In aggiunta, il vettore di bit viene memorizzato in memoria attraverso l'utilizzo della funzione `memmap`, consentendo sia di condividerlo tra i vari processi che di conservare i risultati ottenuti in memoria.

```
    def par_filter_all(self, items, n_threads):
        # Split items in chunks
        chunks = np.array_split(items, n_threads)

        # Start parallel setup
        start = time.time()
        results = Parallel(n_jobs=n_threads)(delayed(self.
    seq_filter_all)(chunk) for chunk in chunks)
        end_time = time.time() - start

        # Sum errors
        t, errs = zip(*results)
        errors = sum(errs)
        return end_time, errors
```

Nella fase di filtraggio, abbiamo parallelizzato la verifica della presenza di un'email nel BloomFilter mediante l'utilizzo della funzione `Parallel`. Anche in questo caso, abbiamo diviso il vettore di email in chunk corrispondenti al numero di processi impiegati.

In seguito, abbiamo voluto esaminare l'impatto della dimensione del chunk anche al di là del numero di processi utilizzati, al fine di valutare se un suo aumento potesse o meno offrire vantaggi in termini di tempo e di speedup.

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 condotti su un dataset composto da 10.000 fino a 10.000.000 di email, considerando probabilità di falsi positivi pari a 0,10, 0,05 e 0,01. Per una valutazione preliminare dei tempi di esecuzione, dello speedup e del False Positive Rate (FPR), focalizziamo l'attenzione sul valore di 0,05 per il FPR.

Gli altri valori di FPR sono riportati nell'appendice **??**.

Il numero di processi considerati varia da 1 (modalità sequenziale) al massimo numero di processi disponibili sulla macchina, ossia 16. Le email sono state generate casualmente attraverso il generatore di email presente nel file `email_generator.py`. Poiché i test coprono un intervallo da 10.000 a 10.000.000 di email, il numero di funzioni hash è approssimativamente pari a 5, e le dimensioni del vettore di bit variano da un minimo di 62.353 a un massimo di 62.352.243.

5.1 OpenMP

5.1.1 Setup

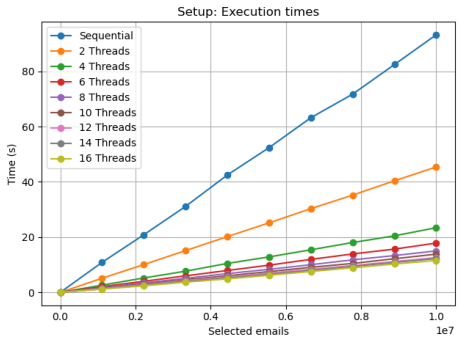

Figure 1: Time setup Omp

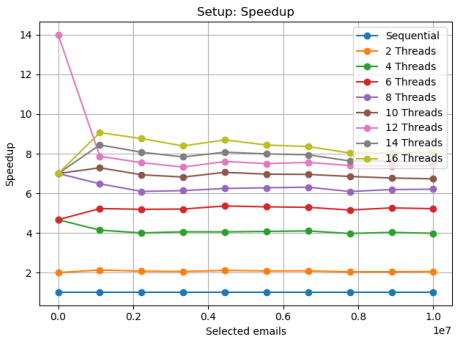

Figure 2: Speedup setup Omp

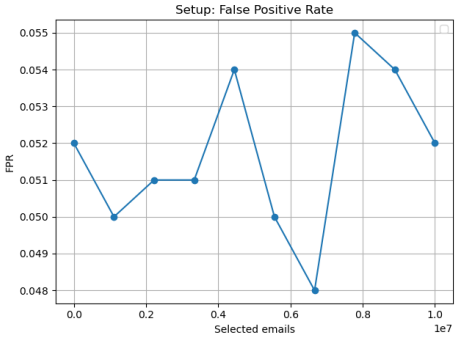

Figure 3: FPR setup Omp

Nell'analisi del tempo di esecuzione e dello speedup, emerge chiaramente come incrementando il numero di processi, il tempo di esecuzione diminuisce significativamente rispetto alla versione sequenziale, raggiungendo un massimo di 9 con l'utilizzo di 16 processi. È osservabile che all'aumentare del numero di processi, l'incremento dello speedup presenta una riduzione, stabilizzandosi al valore di 8. Per quanto concerne il False Positive Rate, i risultati ottenuti si collocano attorno al valore di FPR del 0,05, cioè il valore per il quale è stato configurato per il BloomFilter.

5.1.2 Filter

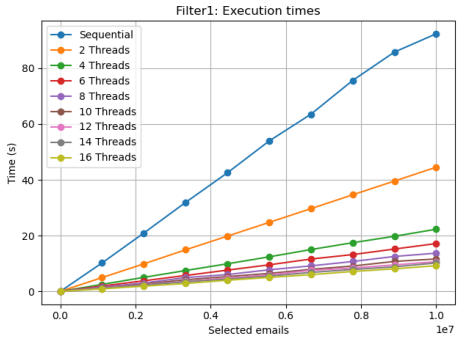

Figure 4: Time Filter Omp

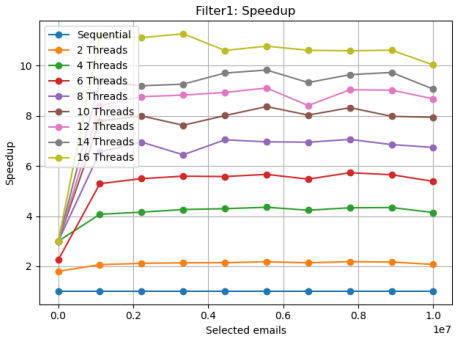

Figure 5: Speedup Filter Omp

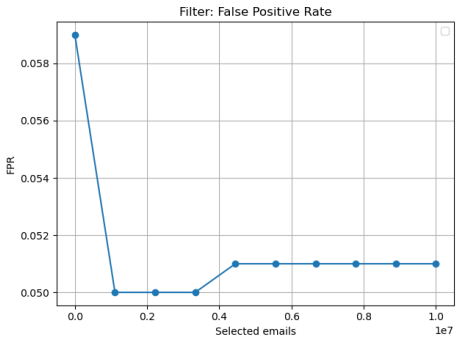

Figure 6: FPR Filter Omp

Analogamente alla fase di setup, nell'incrementare il numero di processi, si osserva una significativa riduzione del tempo di esecuzione rispetto alla modalità sequenziale, con un picco massimo di 11 utilizzando 16 processi. In questa circostanza, tuttavia, la diminuzione dello speedup, al variare dei processi, non è altrettanto marcata rispetto alla fase di setup. Il valore di FPR raggiunge una sorta di plateau una volta che un numero sufficiente di email è stato raggiunto.

5.1.3 Confronto Filter

Qui vengono esaminate le differenze tra le due versioni implementate del filtro.

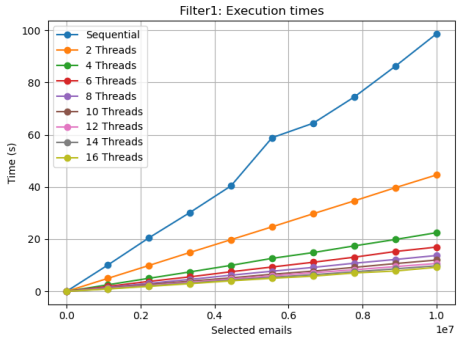

Figure 7: Time Filter 1

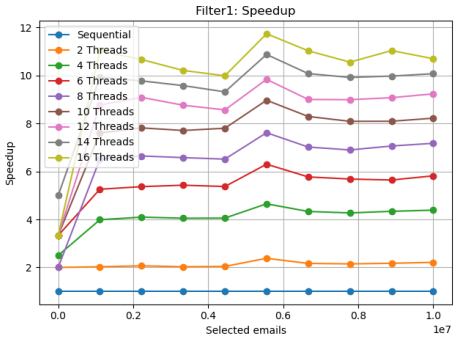

Figure 8: Speedup Filter 1

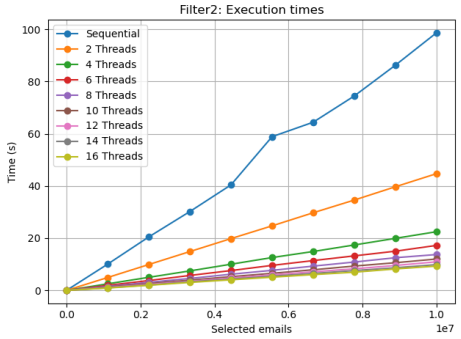

Figure 9: Time Filter 2

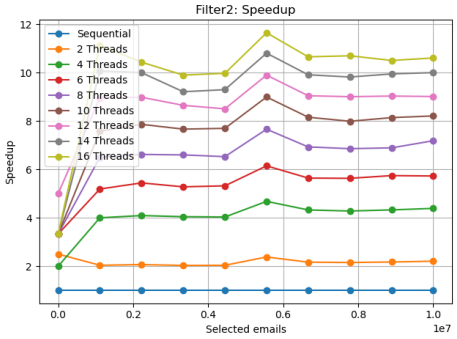

Figure 10: Speedup Filter 2

Come evidente, le due versioni del filtro mostrano performance praticamente identiche sia in termini di tempo di esecuzione che di speedup. Pertanto, a causa di questa similitudine prestazionale, prenderemo in considerazione la prima implementazione per le analisi successive.

5.2 Joblib

5.2.1 Setup

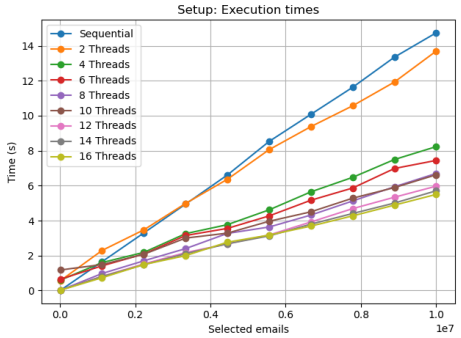

Figure 11: Time setup Joblib

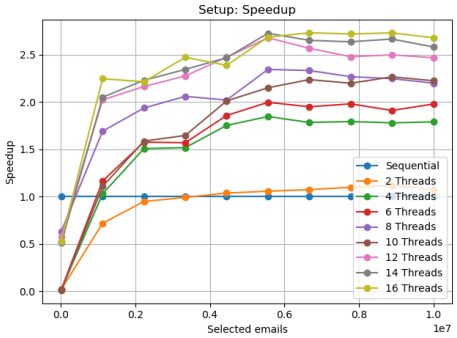

Figure 12: Speedup setup Joblib

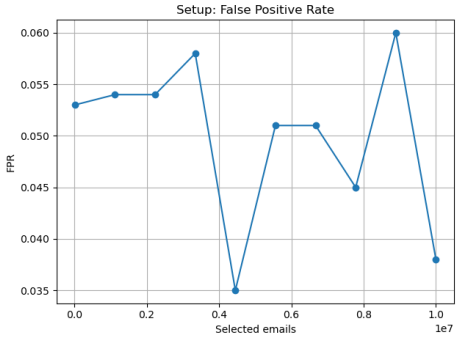

Figure 13: FPR setup Joblib

I risultati ottenuti nella fase di setup con la versione Joblib evidenziano come l'aumento del numero di processori conduca a una riduzione del tempo di esecuzione, seppur in maniera meno significativa rispetto alla versione OpenMP. Lo speedup risulta notevole e superiormente performante rispetto alla versione sequenziale una volta che un certo numero di email è stato raggiunto. Questo fenomeno è attribuibile al tempo di virtualizzazione dei processi, che in questo caso incide negativamente sul tempo di esecuzione per un numero ridotto di email. Il valore massimo di speedup raggiunto si assesta intorno a 2.7. Anche in questo caso, i valori di FPR (False Positive Rate) si collocano attorno al valore di 0.05.

5.2.2 Filter

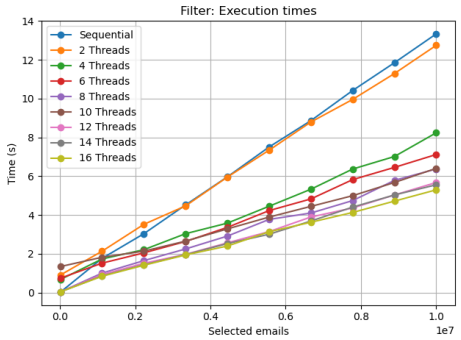

Figure 14: Time Filter Joblib

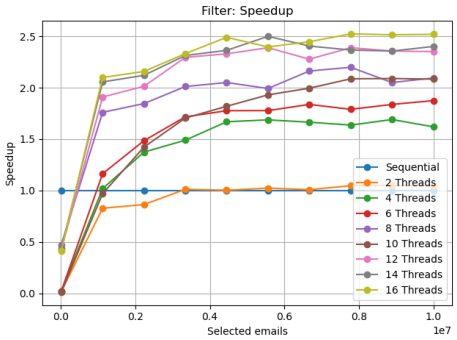

Figure 15: Speedup Filter Joblib

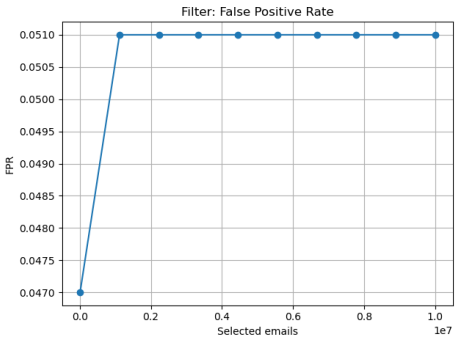

Figure 16: FPR Filter Joblib

Anche nella fase di filtraggio, come nella fase di setup, l'aumento del numero di processi conduce a una riduzione del tempo di esecuzione, seppur in maniera meno significativa rispetto alla versione OpenMP. Infatti il valore massimo di speedup raggiunto si assesta intorno a 2.5. Una volta che un certo numero di email è stato raggiunto, il valore di FPR si stabilizza attorno al valore di 0.05.

5.2.3 Chunks

Ora esamineremo la possibilità di eseguire un'operazione di chunking più estesa rispetto al numero di thread disponibili nella fase di setup per valutare se è possibile migliorare le prestazioni. I valori di riferimento per i chunk sono 16, 32, 64, 128, 256, 512, 1024, 2048.

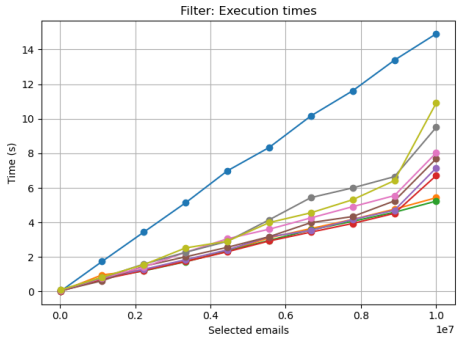

Figure 17: Times setup Chunks

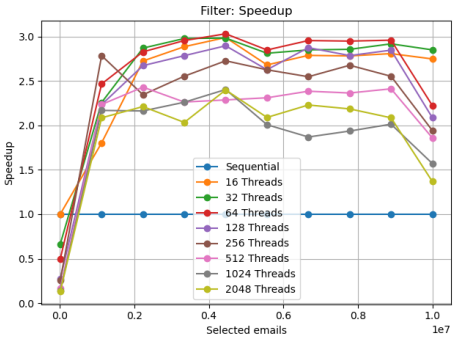

Figure 18: Speedup setup Chunks

I risultati ottenuti suggeriscono che l'implementazione dell'operazione

di chunking ha portato a miglioramenti sostanziali nelle performance di speedup. In particolare, l'aumento del numero di chunk ha contribuito a un notevole miglioramento dello speedup, raggiungendo un massimo di 3. Tuttavia, oltre una soglia di 64 chunk, si è verificato un declino nelle performance.

5.3 Confronto

5.4 Setup

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

5.5 Filter

Nel 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 quella sviluppata con Joblib, raggiungendo un massimo di 11 nella versione OpenMP e 2.5 nella versione Joblib, sfruttando il massimo numero di thread disponibili. I valori di FPR (False Positive Rate) sono praticamente identici tra le due versioni.

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.

A FPR: 0.01

A.1 Setup

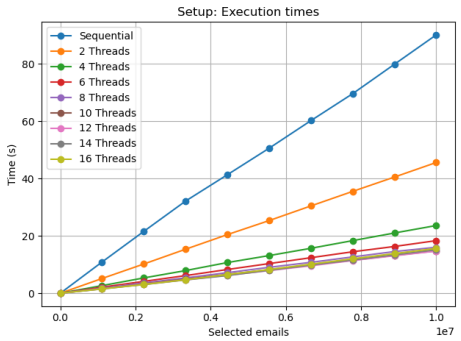

Figure 19: Speedup setup Omp

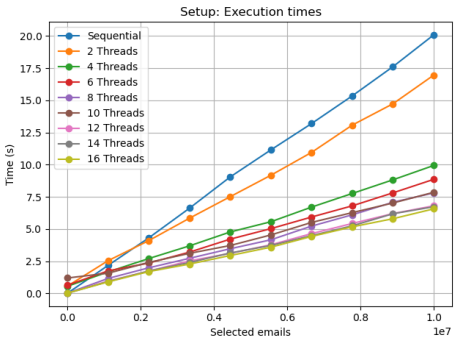

Figure 20: Speedup setup Joblib

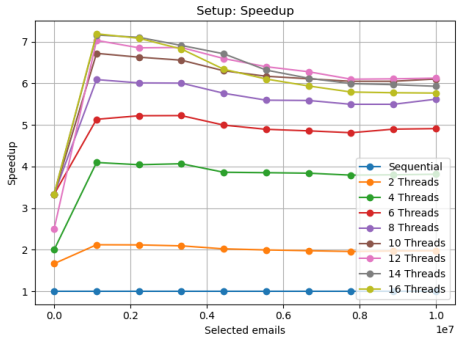

Figure 21: Speedup setup Omp

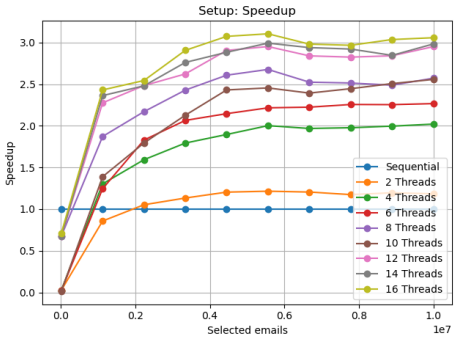

Figure 22: Speedup setup Joblib

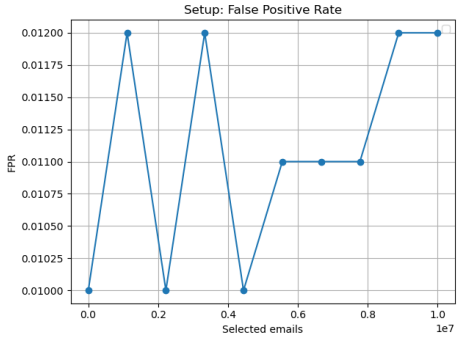

Figure 23: Speedup setup Omp

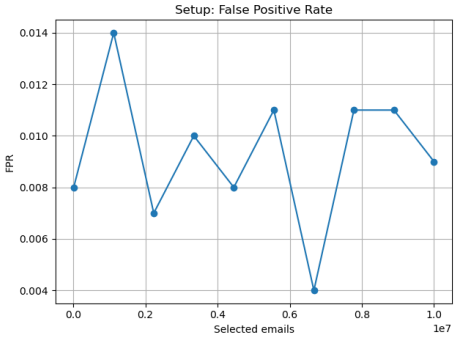

Figure 24: Speedup setup Joblib

A.2 Filter

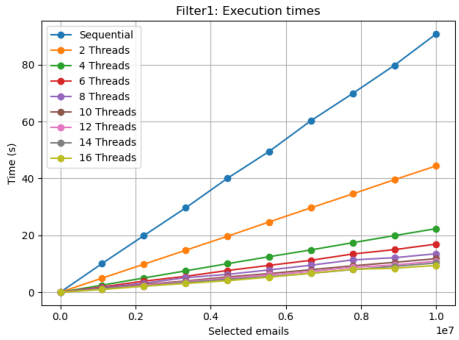

Figure 25: Times filter Omp

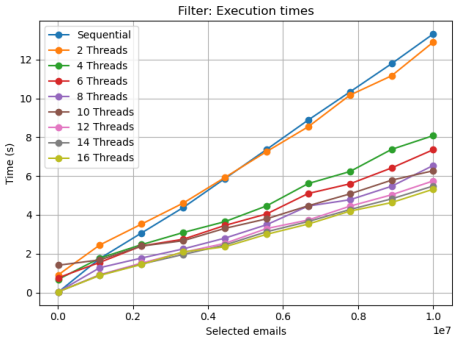

Figure 26: Times filter Joblib

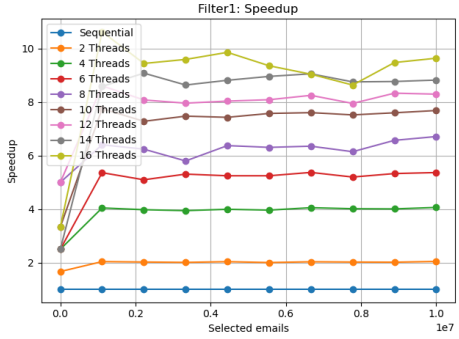

Figure 27: Speedup filter Omp

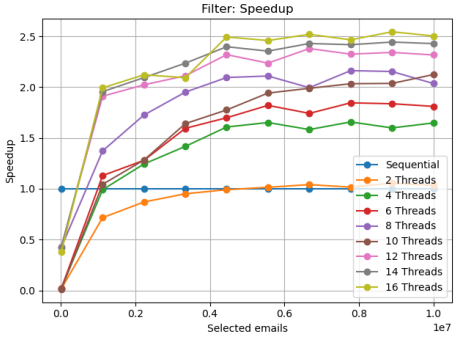

Figure 28: Speedup filter Joblib

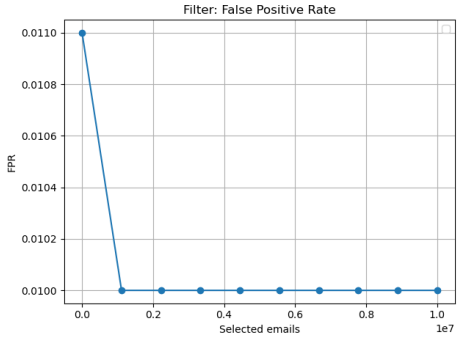

Figure 29: FPR filter Omp

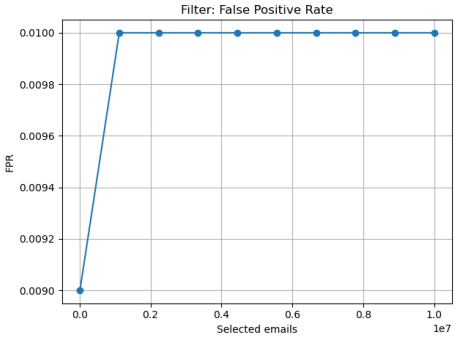

Figure 30: FPR filter Joblib

A.3 Chunks

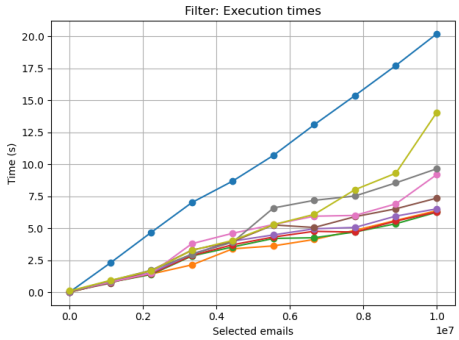

Figure 31: Times chunks Joblib

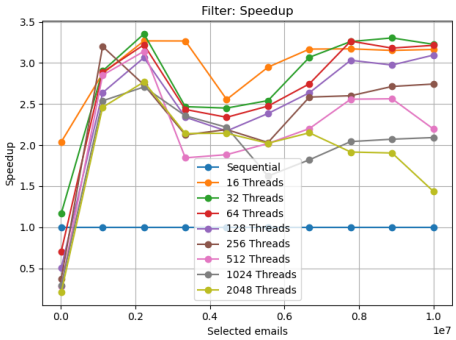

Figure 32: Speedup chunks Joblib

B FPR: 0.10

B.1 Setup

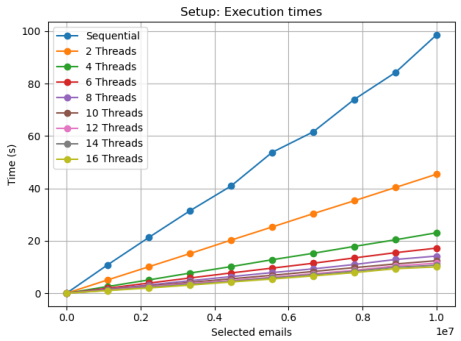

Figure 33: Time setup Omp

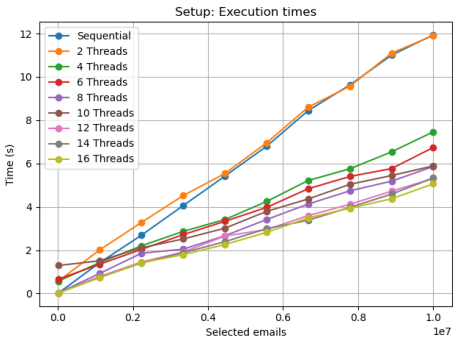

Figure 34: Time setup Joblib

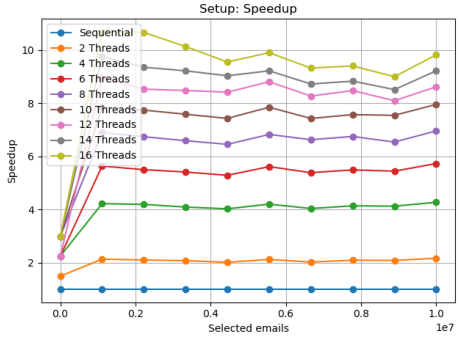

Figure 35: Speedup setup Omp

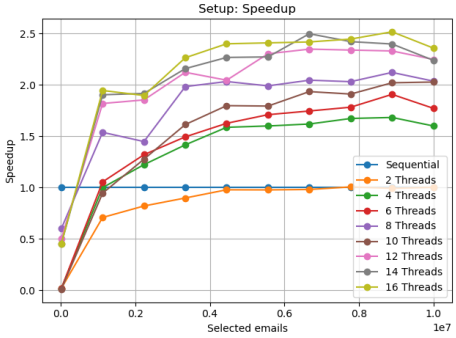

Figure 36: Speedup setup Joblib

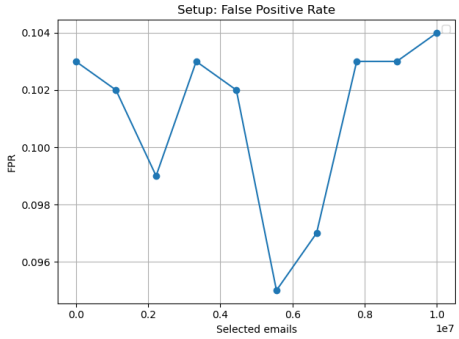

Figure 37: FPR setup Omp

Figure 38: FPR setup Joblib

B.2 Filter

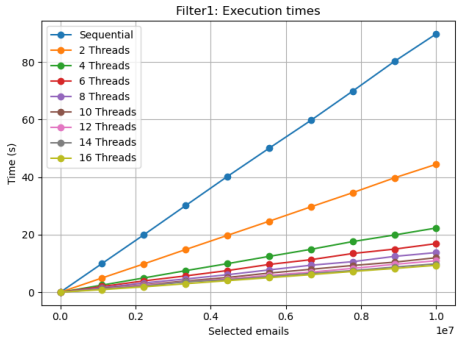

Figure 39: Time setup Omp

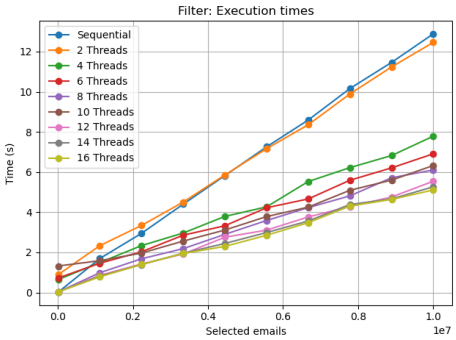

Figure 40: Speedup setup Joblib

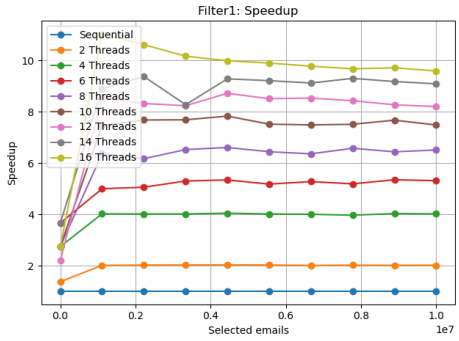

Figure 41: Speedup setup Omp

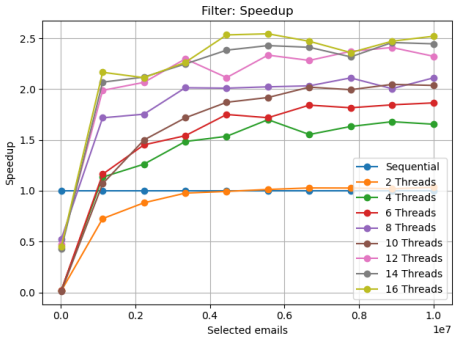

Figure 42: Speedup setup Joblib

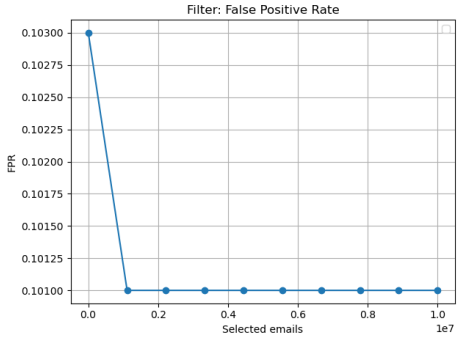

Figure 43: FPR Filter Omp

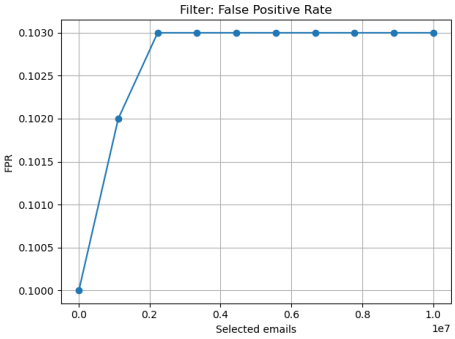

Figure 44: FPR Filter Joblib

B.3 Chunks

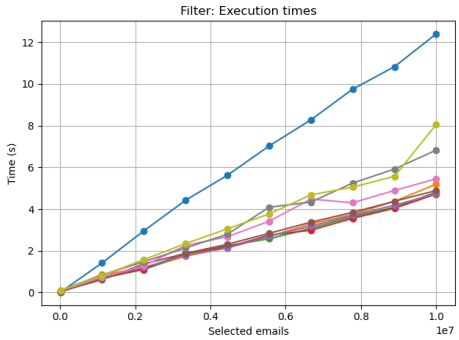

Figure 45: Times setup Chunk

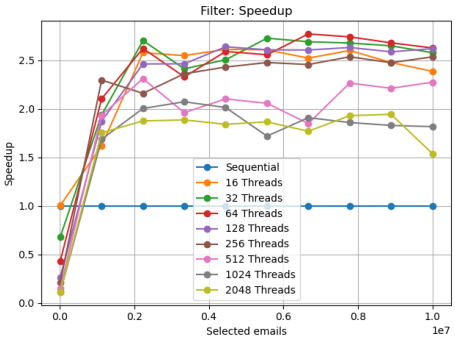

Figure 46: Speedup setup Chunk