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 parte sequenziale del setup, dopo una fase di inizializzazione che serve per calcolare il valore ottimale del vettore di bit e il numero di funzioni hash basate sul valore di probabilità di falsi positivi, si procede con l'indicizzazione nel vettore di bit per ogni 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 parte sequenziale del filtraggio, viene inizializzata una variabile di conteggio per i falsi positivi, che verrà incrementata ogni volta che un'email viene classificata 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.

```
double BloomFilter::parallelSetup(std::string items[], std::
    size_t nItems) {
    initialize(nItems);
    double start = omp_get_wtime();
#pragma omp parallel default(none) shared(bits, items)
    firstprivate(nItems, nHashes)
    {
#pragma omp for
        for(std::size_t i=0; i < nItems; i++) {
            MultiHashes mh(this->size, items[i]);
            for (std::size_t h = 0; h < nHashes; h++) {
                std::size_t index = mh();
#pragma omp critical
                this->bits[index] = true;
            }
        }
    }
    return omp_get_wtime() - start;
}
```

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.

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

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

La prima implementazione prevede l'uso della direttiva `omp parallel for` per parallelizzare la verifica della presenza di un'email all'interno del BloomFilter, e della direttiva `omp atomic` per garantire l'accesso esclusivo alla variabile incrementale 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 implementazione prevede invece l'uso di una variabile incrementale per ogni thread, e la somma di queste variabili al termine dell'operazione di filtraggio, anch'essa utilizzando la direttiva `omp atomic` per garantire l'accesso esclusivo alla variabile incrementale globale.

Verificheremo in seguito la differenza tra queste due implementazioni.

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.

```
    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
```

Come nel caso di Omp, abbiamo parallelizzato l'indicizzazione nel vettore di bit per ogni email, utilizzando la funzione `Parallel`, suddividendo il vettore di email in chunk pari al numero di processi impiegati.

```
    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
```

Per la fase di filtraggio, abbiamo parallelizzato la verifica della presenza di un'email all'interno del BloomFilter, utilizzando la funzione `Parallel`, suddividendo anche in questo caso il vettore di email in chunk pari al numero di processi impiegati.

Successivamente, abbiamo voluto analizzare l'effetto della dimensione del chunk anche oltre il numero di processi impiegati, per valutare se un suo aumento potesse portare o meno un vantaggio sia in termini di tempo che 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 effettuati su un dataset da 10000 fino a 10000000 email, con una probabilità di falsi positivi di 0.10, 0.05 e 0.01. Per una prima valutazione dei tempi di esecuzione, dello speedup e del False Positive Rate, prendiamo in considerazione il valore di 0.05 per il FPR. Gli altri valori di FPR sono visualizzabili in appendice 7. Il numero di processi presi in considerazione varia da 1 (sequenziale) al massimo numero di processi disponibili sulla macchina utilizzata cioè 16. Le email sono state generate casualmente tramite il generatore di email contenuto nel file `email_generator.py`. Poiché i test variano da 10000 a 10000000 email, il numero di funzioni hash sono circa 5 e la dimensione del vettore di bit varia da un minimo di 62353 a un massimo di 62352243.

5.1 OpenMP

5.1.1 Setup

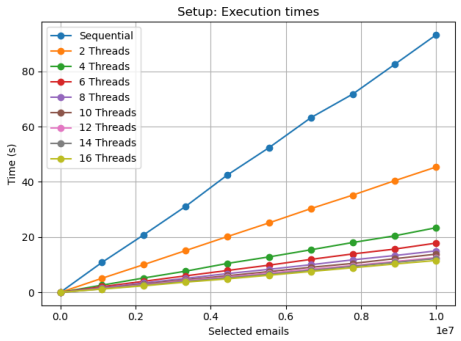

Figure 1: Time setup Omp

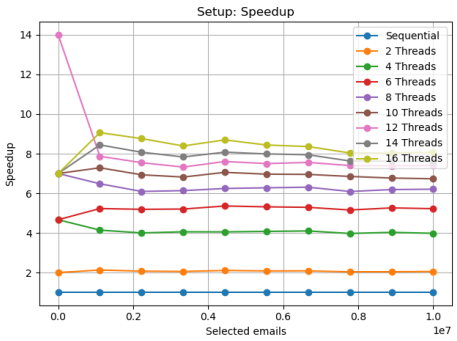

Figure 2: Speedup setup Omp

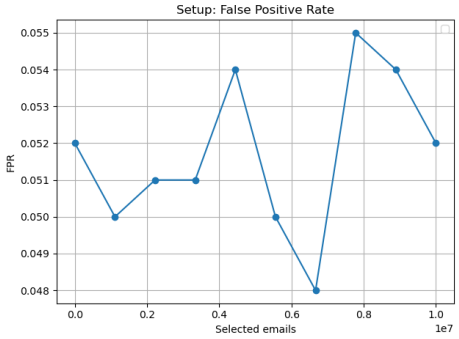

Figure 3: FPR setup Omp

Nella valutazione del tempo di esecuzione e dello speedup, è evidente come aumentando il numero di processi, il tempo di esecuzione diminuisce in modo significativo rispetto alla versione sequenziale, raggiungendo un massimo di 9 con l'utilizzo di 16 processi. Possiamo notare come all'aumentare del numero di processi l'aumento dello speedup diminuisce, raggiungendo il valore di 8. Per quanto riguarda il False Positive Rate, i valori ottenuti si aggirano molto intorno al valore di FPR di 0.05 per il quale è stato configurato il BloomFilter.

5.1.2 Filter

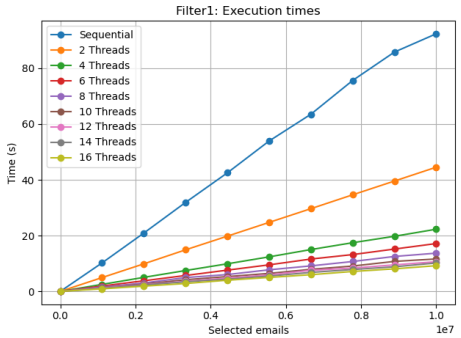

Figure 4: Time Filter Omp

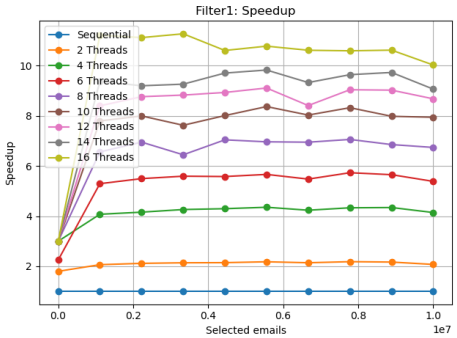

Figure 5: Speedup Filter Omp

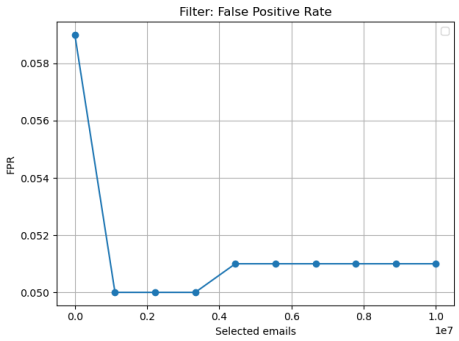

Figure 6: FPR Filter Omp

Come per la fase di setup, anche in questo caso, aumentando il numero di processi, il tempo di esecuzione diminuisce in modo significativo rispetto alla versione sequenziale, raggiungendo un massimo di 11 con l'utilizzo di 16 processi. In questo caso però, la variazione dello speedup non diminuisce in maniera più significativa rispetto alla fase di setup. Il valore di FPR raggiunge il plateau una volta raggiunto un numero sufficiente di email.

5.1.3 Confronto Filter

Qui viene preso in considerazione le differenze tra le due versioni implementate del filtro.

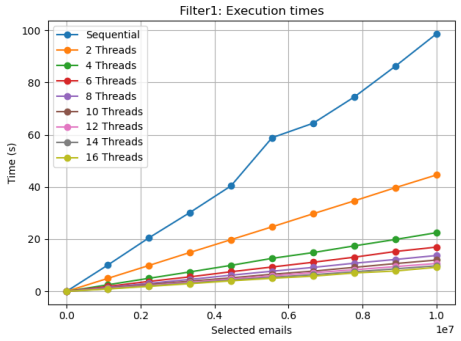

Figure 7: Time Filter 1

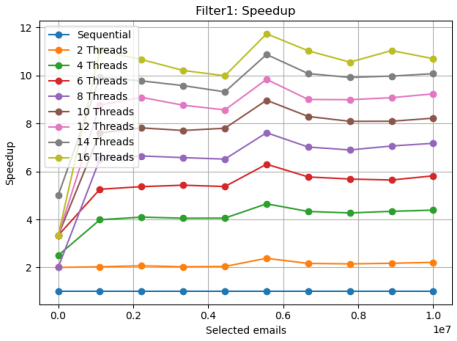

Figure 8: Speedup Filter 1

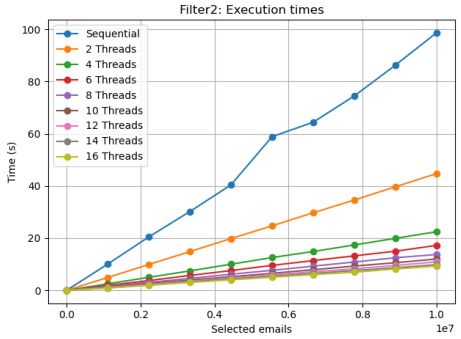

Figure 9: Time Filter 2

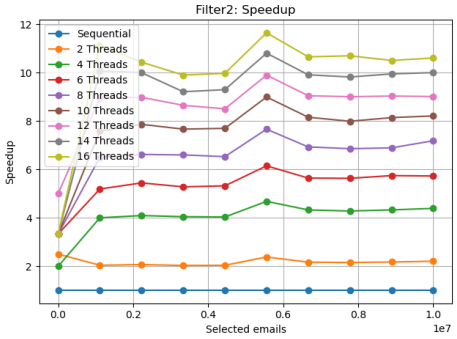

Figure 10: Speedup Filter 2

Come è possibile notare, le due versioni del filtro presentano performance pressoché identiche in termini sia di tempo di esecuzione che di speedup.

5.2 Joblib

5.2.1 Setup

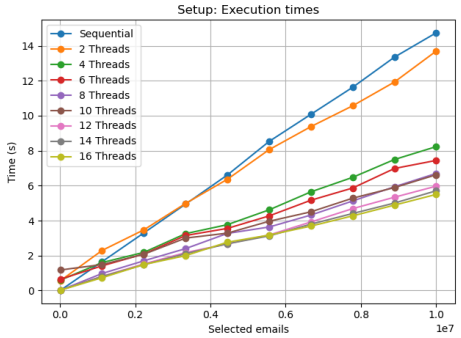

Figure 11: Time setup Joblib

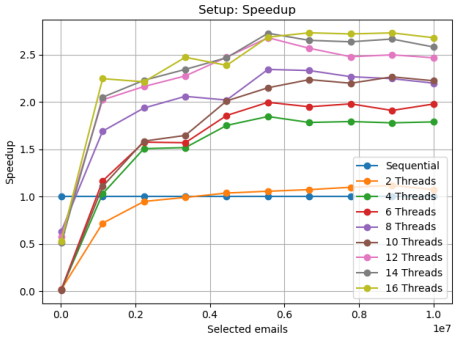

Figure 12: Speedup setup Joblib

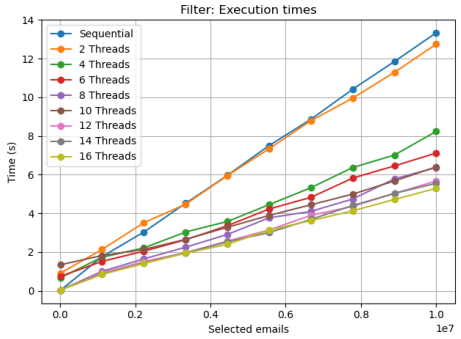

Figure 14: Time Filter Joblib

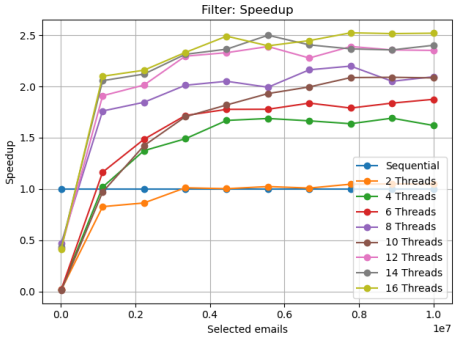

Figure 15: Speedup Filter Joblib

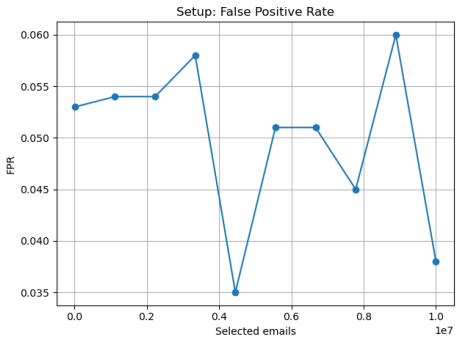

Figure 13: FPR setup Joblib

I risultati ottenuti con la versione Joblib mostrano come l'aumento del numero di processori

5.2.2 Filter

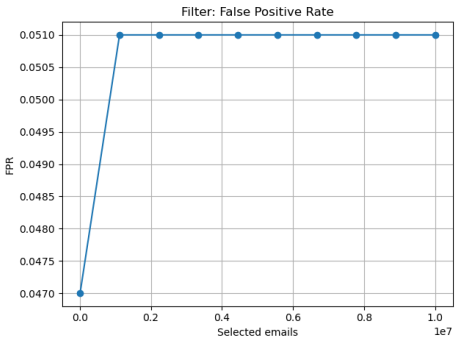

Figure 16: FPR Filter Joblib

5.2.3 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.

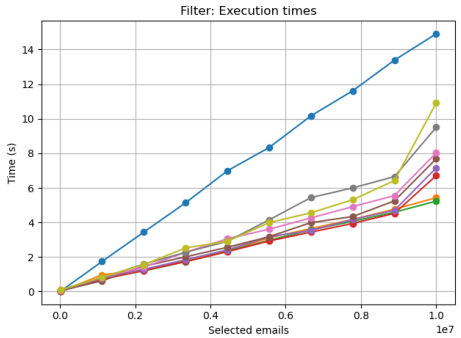

Figure 17: Times setup Chunks

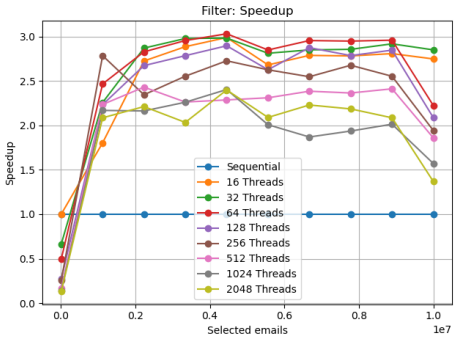

Figure 18: Speedup setup Chunks

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.

5.3 Confronto

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.

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.

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

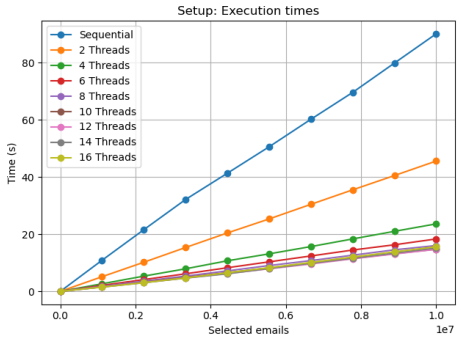

Figure 19: Speedup setup Omp

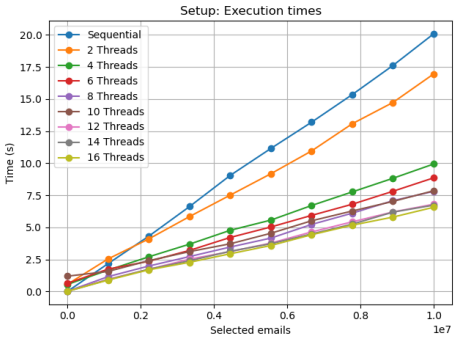

Figure 20: Speedup setup Joblib

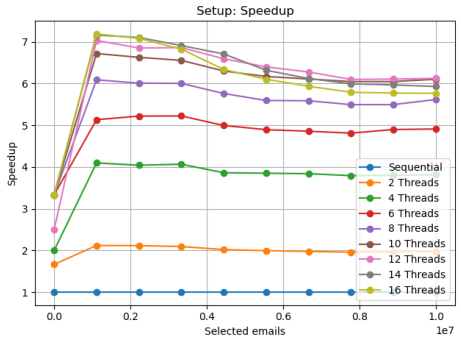

Figure 21: Speedup setup Omp

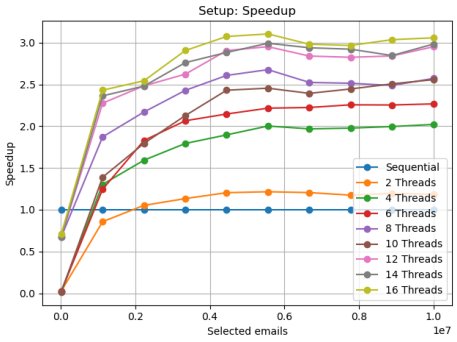

Figure 22: Speedup setup Joblib

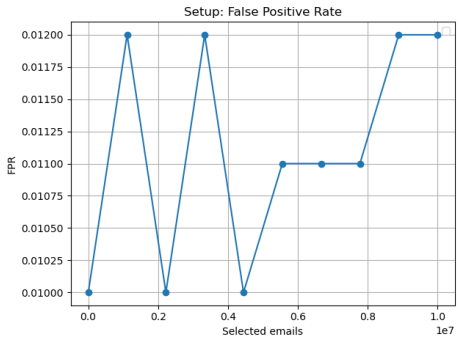

Figure 23: Speedup setup Omp

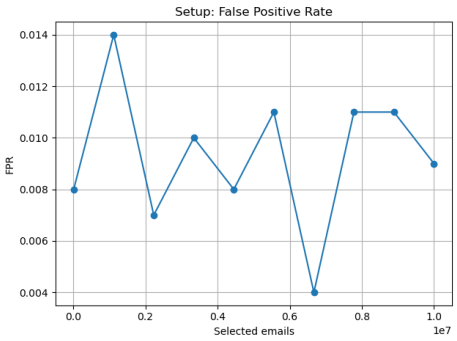

Figure 24: Speedup setup Joblib

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

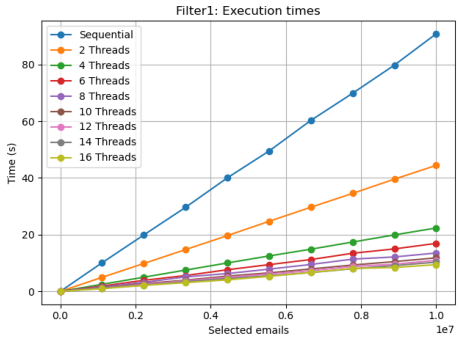

Figure 25: Times filter Omp

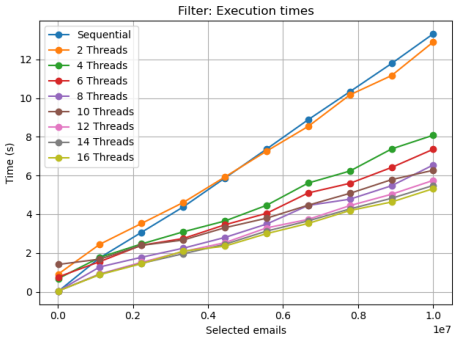

Figure 26: Times filter Joblib

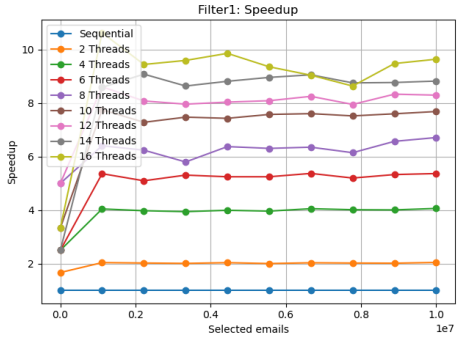

Figure 27: Speedup filter Omp

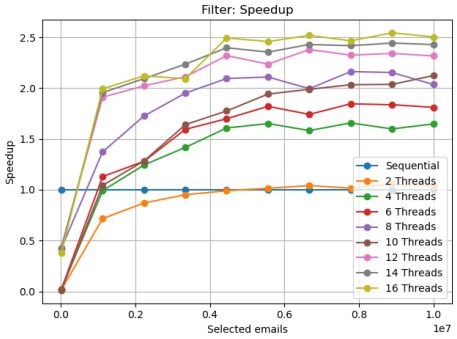

Figure 28: Speedup filter Joblib

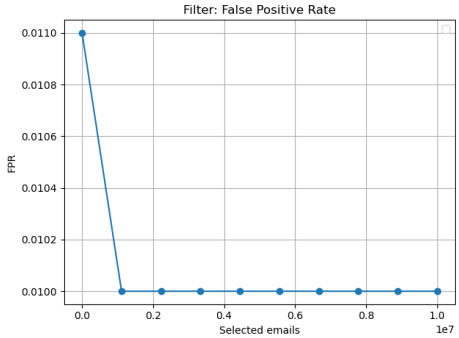

Figure 29: FPR filter Omp

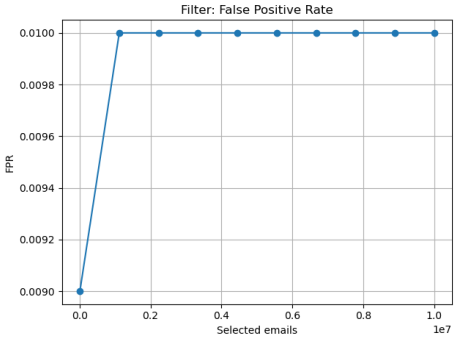

Figure 30: FPR filter Joblib

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

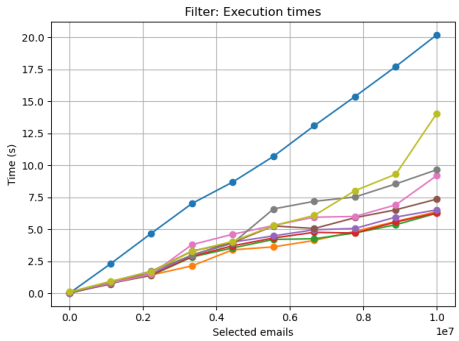

Figure 31: Times chunks Joblib

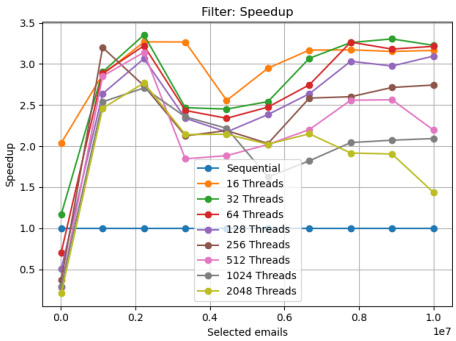

Figure 32: Speedup chunks Joblib

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

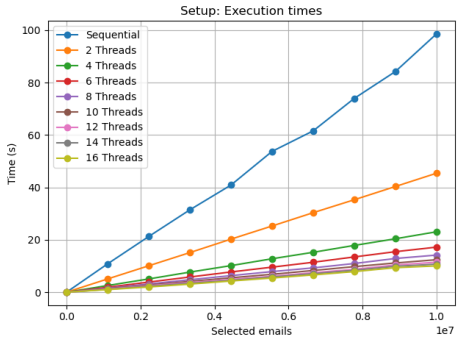

Figure 33: Time setup Omp

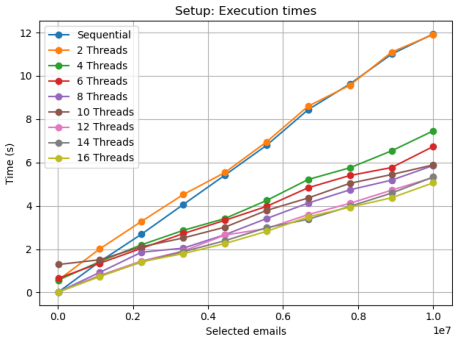

Figure 34: Time setup Joblib

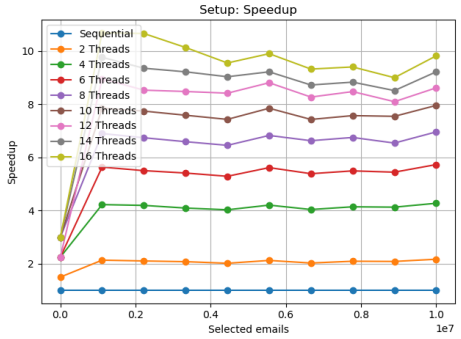

Figure 35: Speedup setup Omp

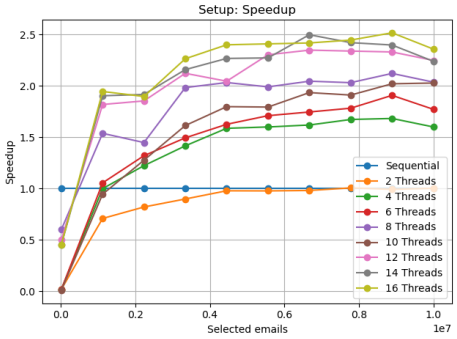

Figure 36: Speedup setup Joblib

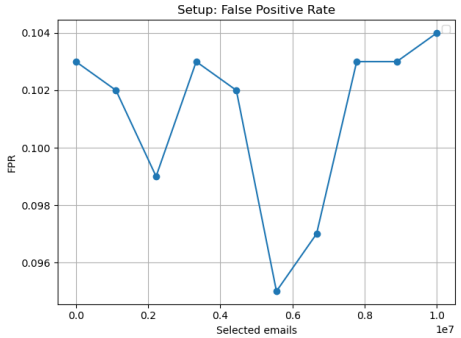

Figure 37: FPR setup Omp

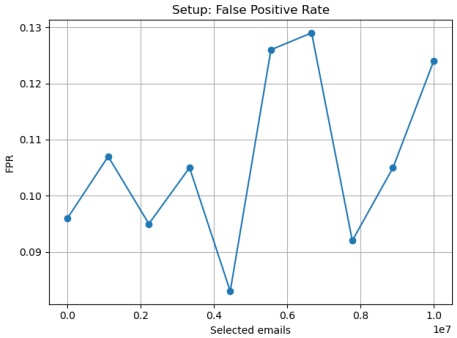

Figure 38: FPR setup Joblib

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

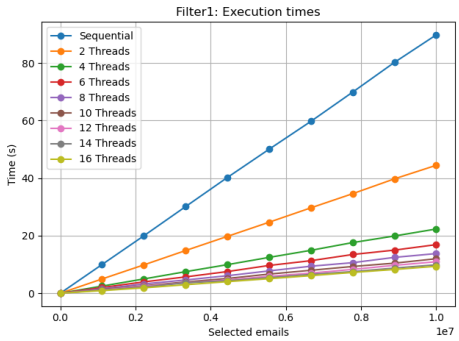

Figure 39: Time setup Omp

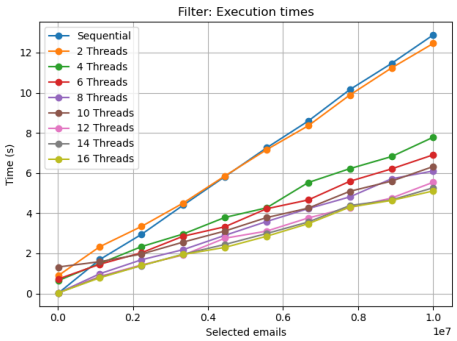

Figure 40: Speedup setup Joblib

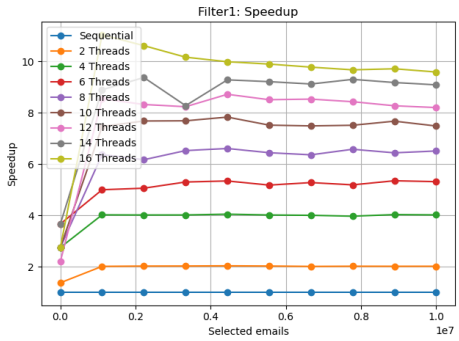

Figure 41: Speedup setup Omp

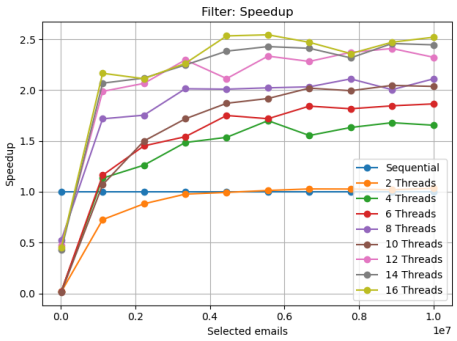

Figure 42: Speedup setup Joblib

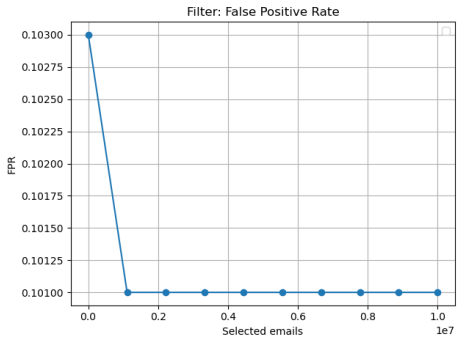

Figure 43: FPR Filter Omp

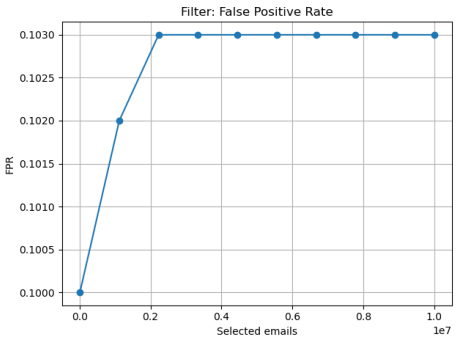

Figure 44: FPR Filter Joblib

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

7.2.3 Chunks

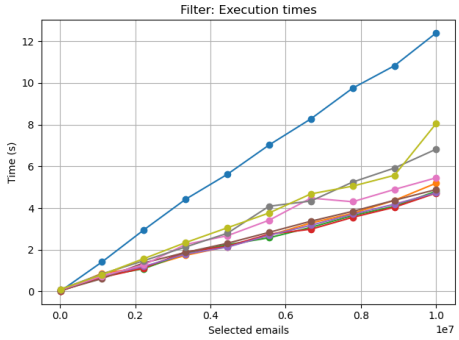

Figure 45: Times setup Chunk

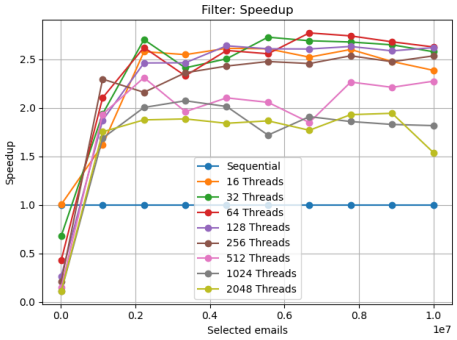

Figure 46: Speedup setup Chunk

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.