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 processori 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:

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

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

```
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;
}
```

Il codice sequenziale di riferimento da parallelizzare per la fase di filter è il seguente:

```
    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
```

```
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;
}
```

3.1 OpenMP

OpenMP (Omp) è una libreria in linguaggio C progettata per consentire la parallelizzazione di funzioni e cicli for. La funzione `omp parallel` richiede

come input il numero di processori da impiegare e la funzione da parallelizzare. 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 processori 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;
            }
        }
    }
#pragma omp barrier
    return omp_get_wtime() - start;
}
```

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 processori 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)
```

```
6
7           # Start parallel setup
8           start = time.time()
9           Parallel(n_jobs=n_threads)(delayed(self.add)(chunk) for
        chunk in chunks)
10          return time.time() - start
```

```
1       def par_filter_all(self, items, n_threads):
2           # Split items in chunks
3           chunks = np.array_split(items, n_threads)
4
5           # Start parallel setup
6           start = time.time()
7           results = Parallel(n_jobs=n_threads)(delayed(self.
       seq_filter_all)(chunk) for chunk in chunks)
8           end_time = time.time() - start
9
10          # Sum errors
11          t, errs = zip(*results)
12          errors = sum(errs)
13          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 Tests

I test sono stati effettuati su un dataset da 1000 fino a 10000000 email, con una probabilità di falsi positivi di 0.10, 0.05 e 0.01.

6 Risultati

6.1 FPR: 0.10

6.1.1 Setup

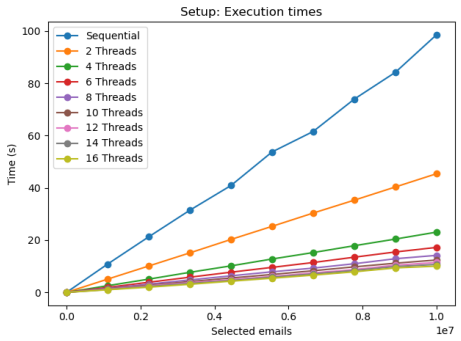


Figure 1: Speedup setup Omp

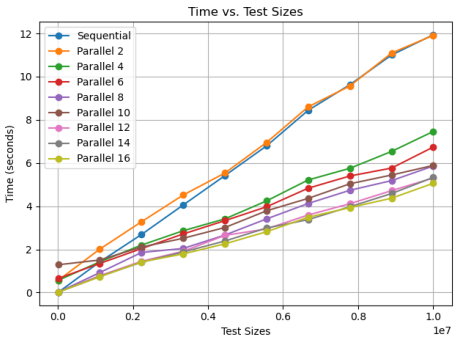


Figure 2: Speedup setup Joblib

Figure 3: Time setup

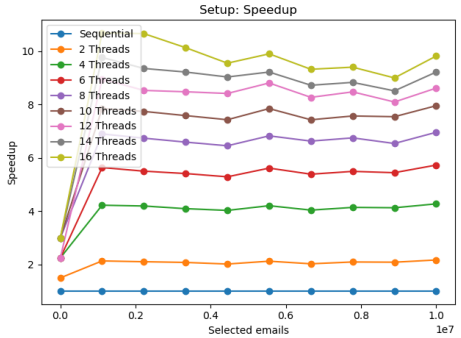


Figure 4: Speedup setup Omp

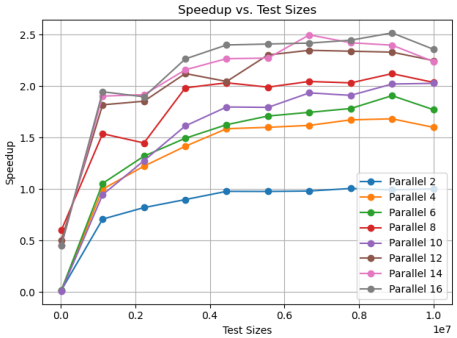


Figure 5: Speedup setup Joblib

Figure 6: Speedup setup

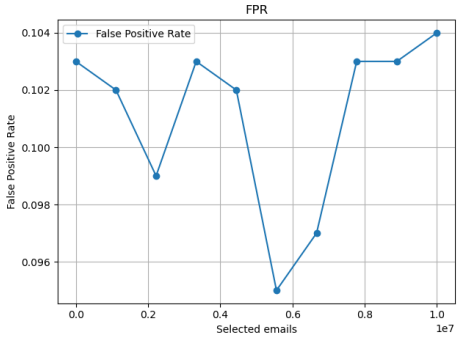


Figure 7: Speedup setup Omp

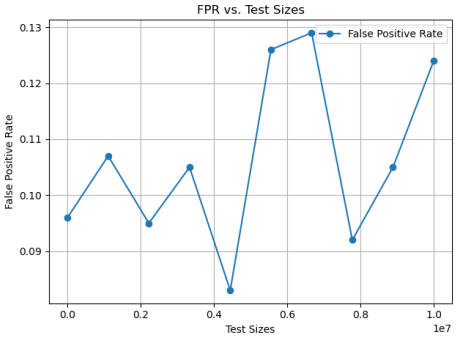


Figure 8: Speedup setup Joblib

Figure 9: Time setup

6.1.2 Filter

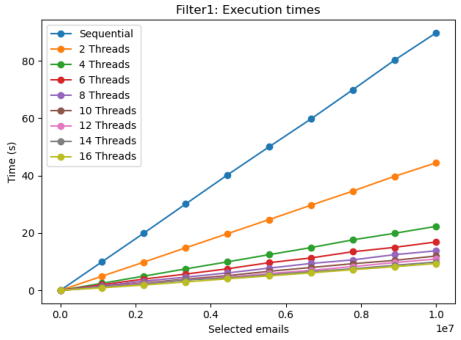


Figure 10: Speedup setup Omp

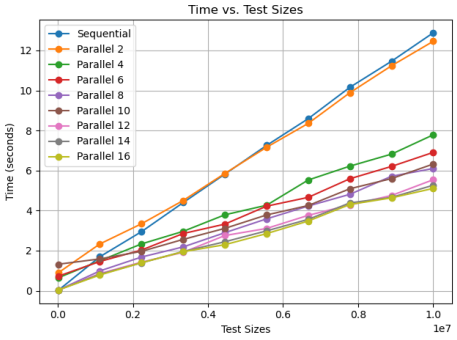


Figure 11: Speedup setup Joblib

Figure 12: Time setup

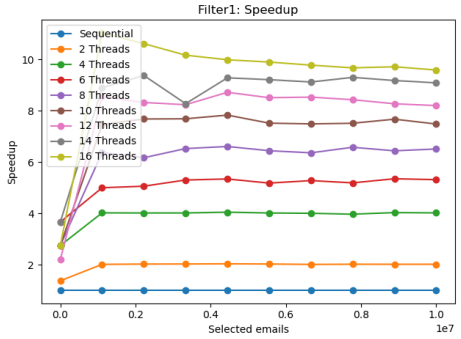


Figure 13: Speedup setup Omp

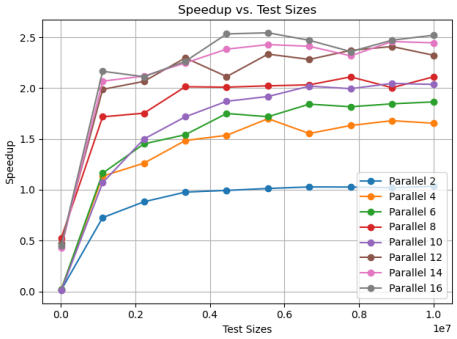


Figure 14: Speedup setup Joblib

Figure 15: Speedup setup

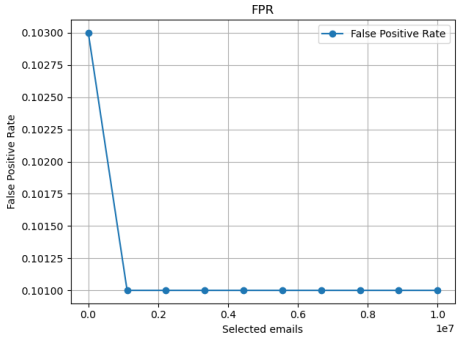


Figure 16: Speedup setup Omp

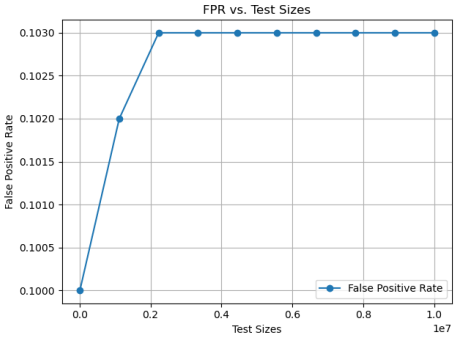


Figure 17: Speedup setup Joblib

Figure 18: Time setup

6.1.3 Chunks

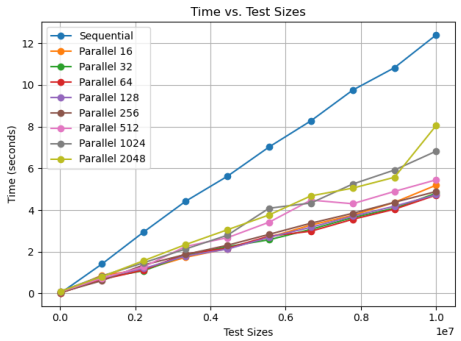


Figure 19: Times setup Chunk

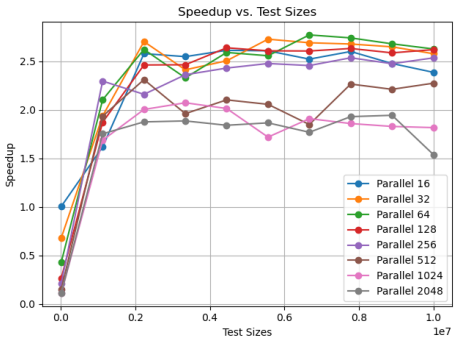


Figure 20: Speedup setup Chunk

Figure 21: Time setup

6.2 FPR: 0.05

6.2.1 Setup

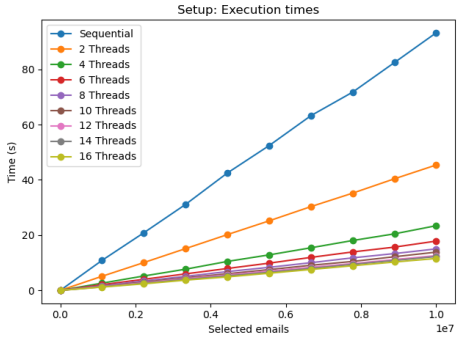


Figure 22: Speedup setup Omp

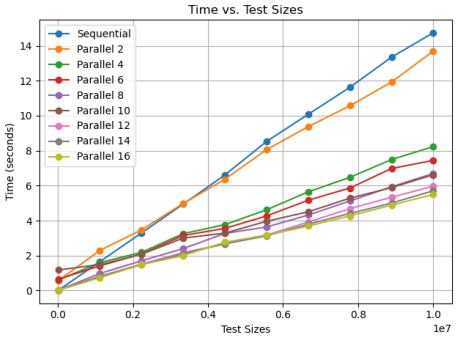


Figure 23: Speedup setup Joblib

Figure 24: Time setup

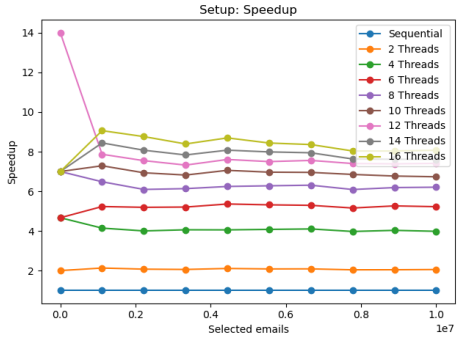


Figure 25: Speedup setup Omp

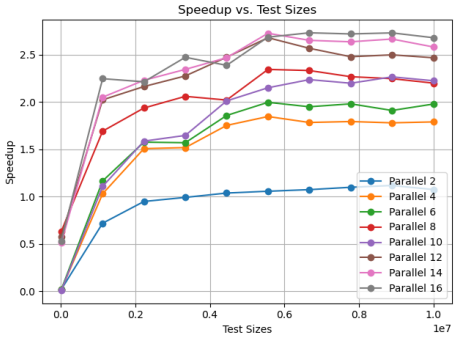


Figure 26: Speedup setup Joblib

Figure 27: Speedup setup

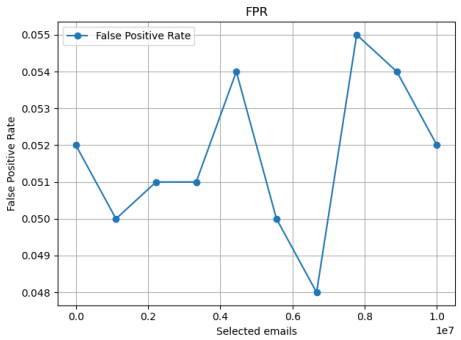


Figure 28: Speedup setup Omp

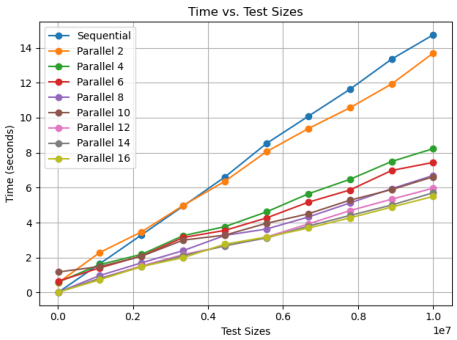
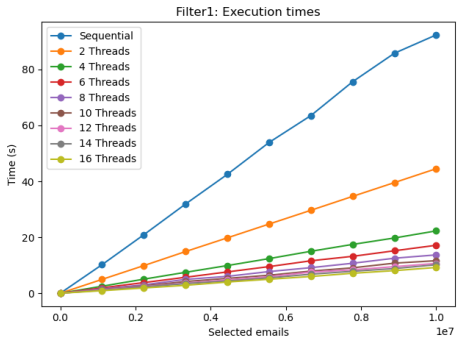


Figure 29: Speedup setup Joblib

Figure 30: Time setup

6.2.2 Filter



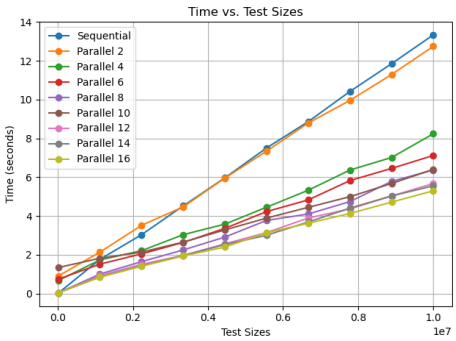
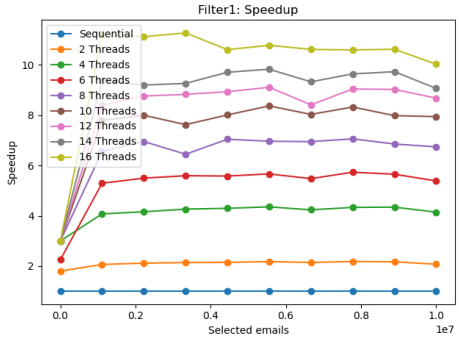


Figure 31: Speedup setup Omp

Figure 32: Speedup setup Joblib

Figure 33: Time setup



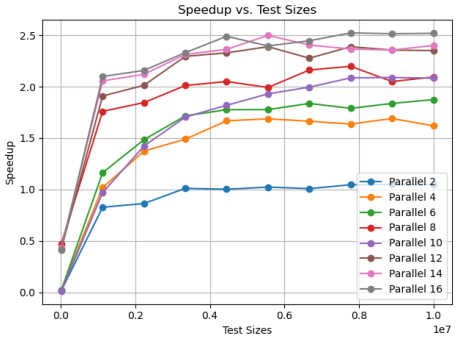


Figure 34: Speedup setup Omp

Figure 35: Speedup setup Joblib

Figure 36: Speedup setup

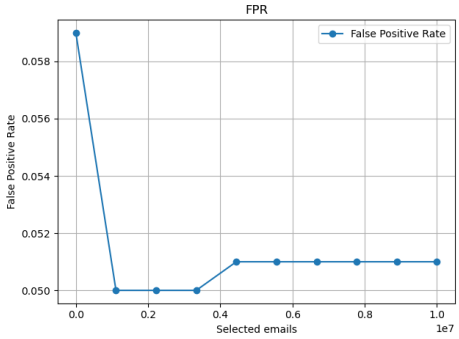


Figure 37: Speedup setup Omp

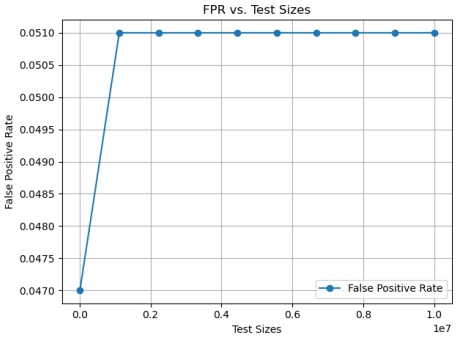


Figure 38: Speedup setup Joblib

Figure 39: Time setup

6.2.3 Chunks

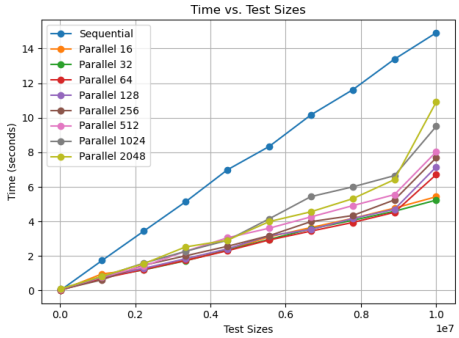


Figure 40: TImes setup Chunks

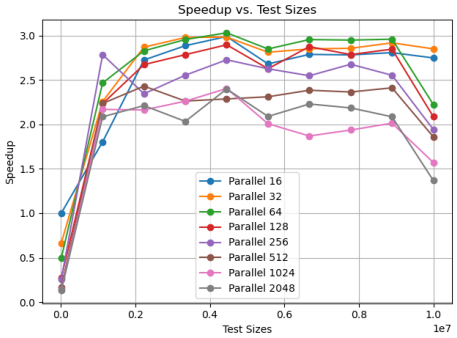


Figure 41: Speedup setup Chunks

Figure 42: Time setup

6.3 FPR: 0.01

6.3.1 Setup

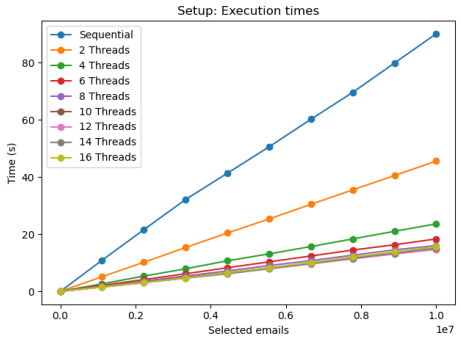


Figure 43: Speedup setup Omp

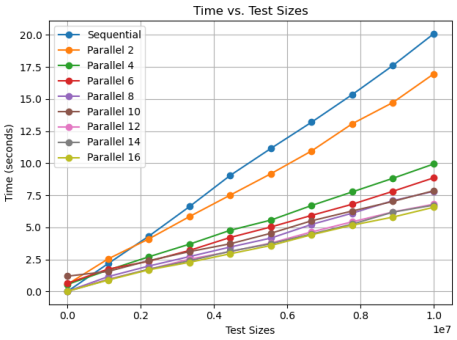


Figure 44: Speedup setup Joblib

Figure 45: Time setup

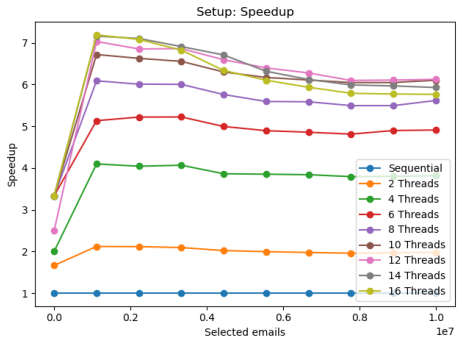


Figure 46: Speedup setup Omp

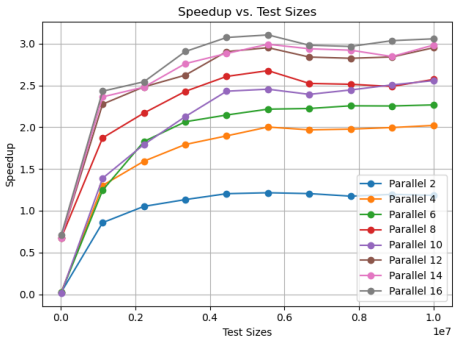


Figure 47: Speedup setup Joblib

Figure 48: Speedup setup

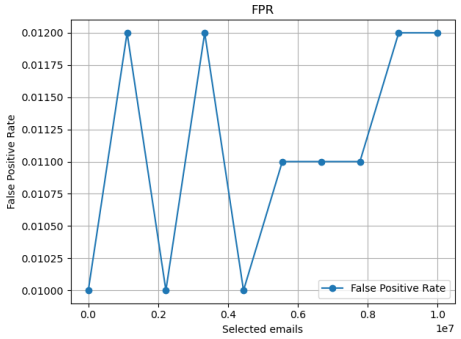


Figure 49: Speedup setup Omp

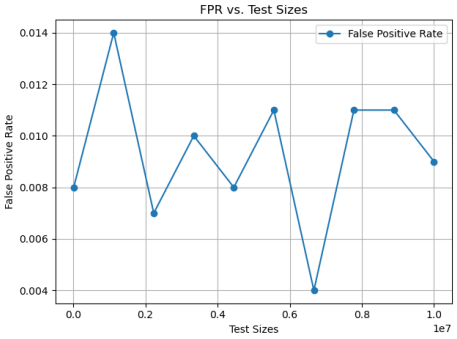


Figure 50: Speedup setup Joblib

Figure 51: Time setup

6.3.2 Filter

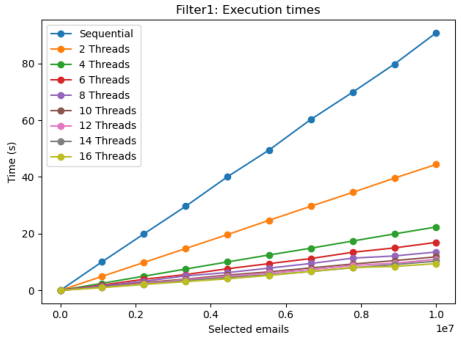


Figure 52: Times filter Omp

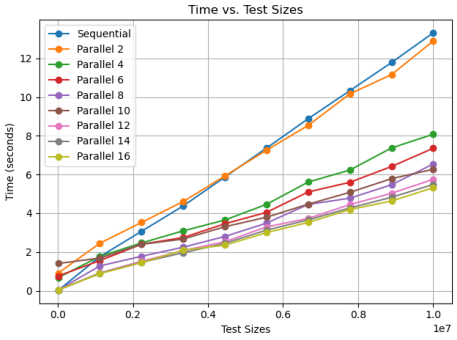


Figure 53: Times filter Joblib

Figure 54: Time filter

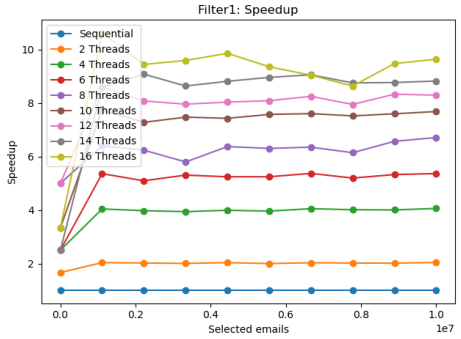


Figure 55: Speedup filter Omp

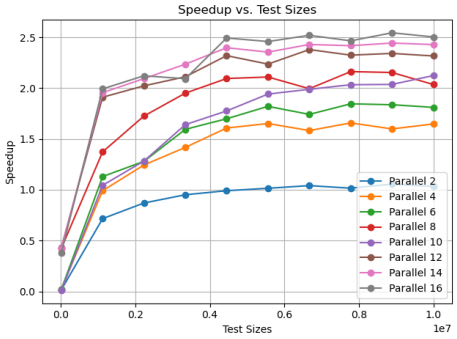


Figure 56: Speedup filter Joblib

Figure 57: Speedup filter

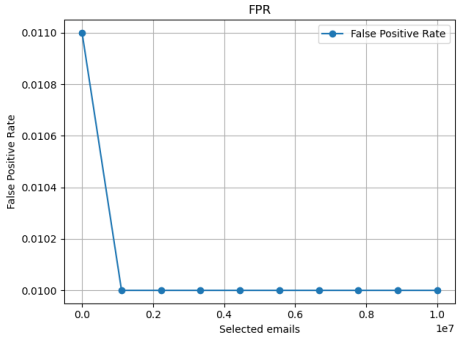


Figure 58: FPR filter Omp

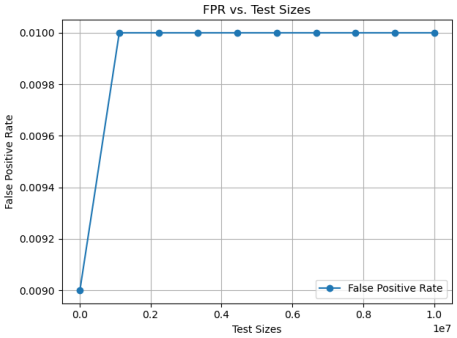


Figure 59: FPR filter Joblib

Figure 60: Time filter

6.3.3 Chunks

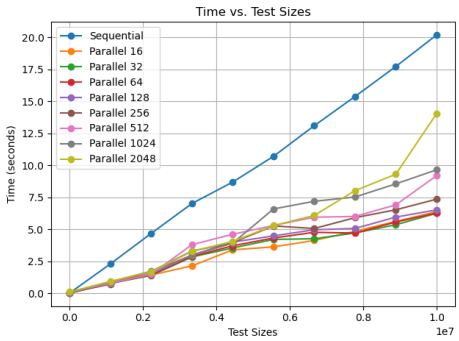


Figure 61: Times chunks Joblib

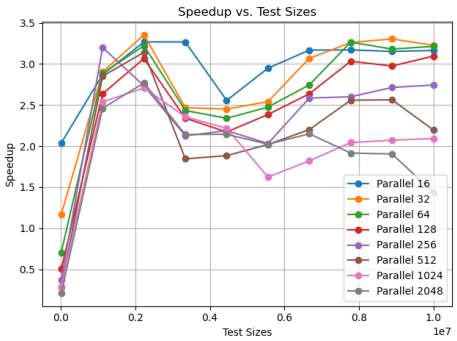


Figure 62: Speedup chunks Joblib

Figure 63: Speedup chunks

7 Conclusioni

Dai risultati ottenuti possiamo vedere come il valore di Speedup massimo si attesti intorno a 3 rispetto alla versione sequenziale. In generale possiamo notare come lo speedup aumenti con l'aumentare della dimensione dell'insieme, fino a un certo punto per la versione Joblib.