

BloomFilter Speedup on setup and filter: Parallel Programming for Machine Learning

Lorenzo Baiardi, Thomas Del Moro

GG MM AAAA

1 Introduzione

Il progetto consiste nella parallelizzazione di un BloomFilter per la classificazione di email di spam, utilizzando la libreria Omp e Joblib. Le operazioni da parallelizzare sono la fase di setup e la fase di filter. Vogliamo verificare come varia lo speedup al variare della dimensione dell'insieme su cui effettuare il training, come varia lo speedup al variare della numero di processori utilizzati e al variare della implementazione utilizzata.

2 Analisi del problema

Il BloomFilter è una struttura dati probabilistica che permette di verificare se un elemento appartiene a un insieme, in questo caso se un email è di spam o meno. La fase di setup del bloom filter consiste nel calcolo delle funzioni hash e nella creazione del vettore di bit che si basa sulla dimensione dell'insieme su cui si vuole effettuare il training e sulla probabilità di falsi positivi che ci aspettiamo di ottenere. 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.

Una volta passato il dataset di training al bloom filter, questo calcola per ogni email le *h* funzioni hash e setta a 1 i bit corrispondenti alle posizioni calcolate. Per verificare se un email è di spam o meno, il bloom filter calcola le *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:

```

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

```

```

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  }

```

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

```

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

```

```

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  }

```

3.1 Omp

Omp è una libreria C che permette di parallelizzare funzioni e cicli for. La funzione `omp parallel` prende in input il numero di processori da utilizzare e la funzione da parallelizzare. In questo elaborato abbiamo utilizzato la funzione `omp parallel for` per parallelizzare la funzione `setup` e la funzione `filter` del bloom filter. Abbiamo poi successivamente elaborato quanto tempo impiegano le funzioni `setup` e `filter` a seconda del numero di processori utilizzati rispetto al tempo impiegato nella sua 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 for default(none) shared(bits, items)
  firstprivate(nItems, nHashes)
5     for(std::size_t i=0; i < nItems; i++)
6         add(items[i]);
7     return omp_get_wtime() - start;
8 }

```

3.2 Joblib

Joblib è una libreria Python che permette di parallelizzare funzioni e cicli for. La funzione Parallel prende in input il numero di processori da utilizzare e la funzione da parallelizzare. In questo elaborato abbiamo utilizzato la funzione Parallel per parallelizzare la funzione setup e la funzione filter del bloom filter. Abbiamo poi successivamente elaborato quanto tempo impiegano le funzioni setup e filter a seconda del numero di processori utilizzati rispetto al tempo impiegato nella sua 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
  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 che varia da 0.10 a 0.01.

6 Risultati

6.1 FPR: 0.10

6.1.1 Setup

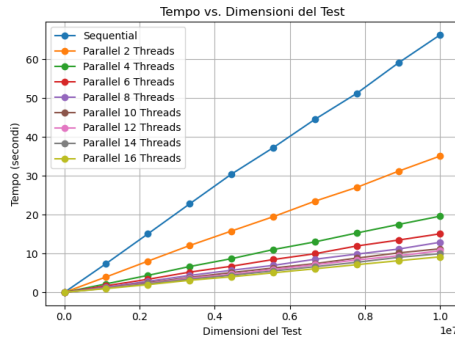


Figure 1: Speedup setup Omp

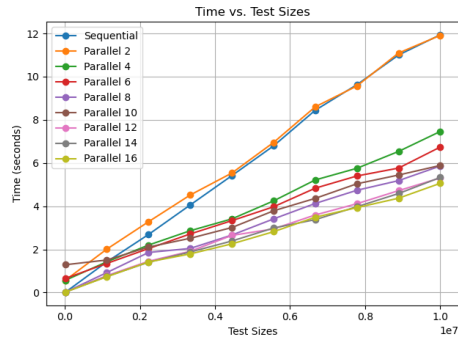


Figure 2: Speedup setup Joblib

Figure 3: Time setup

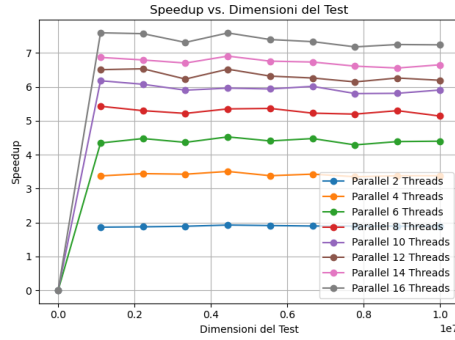


Figure 4: Speedup setup Omp

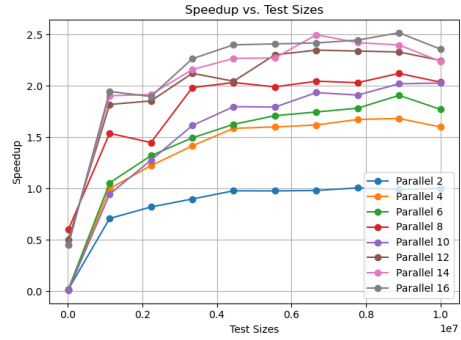


Figure 5: Speedup setup Joblib

Figure 6: Speedup setup

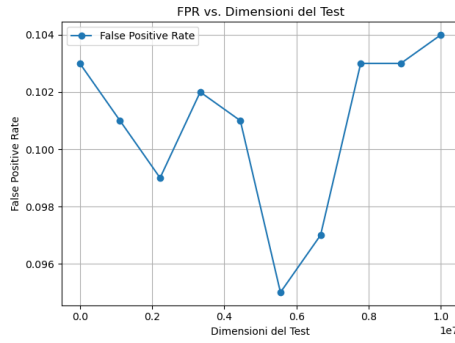


Figure 7: Speedup setup Omp

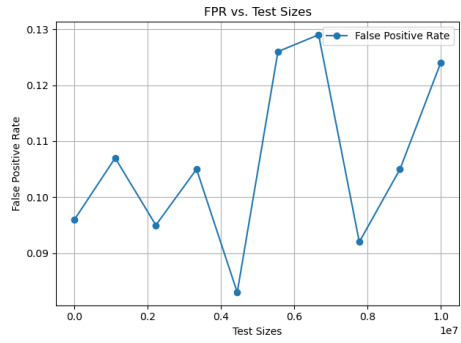


Figure 8: Speedup setup Joblib

Figure 9: Time setup

6.1.2 Filter

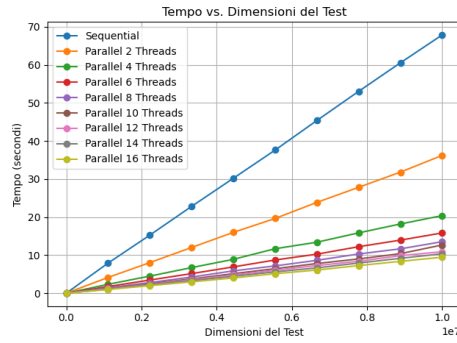


Figure 10: Speedup setup Omp

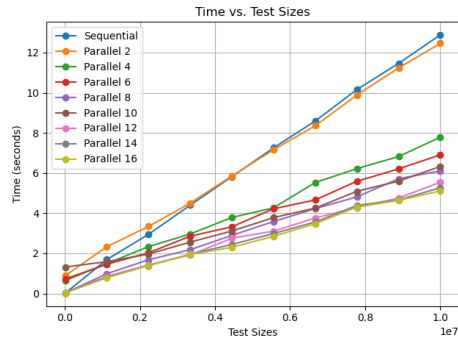


Figure 11: Speedup setup Joblib

Figure 12: Time setup

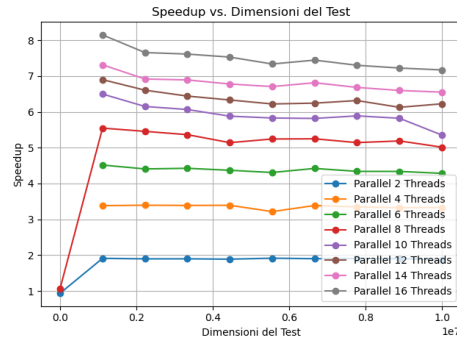


Figure 13: Speedup setup Omp

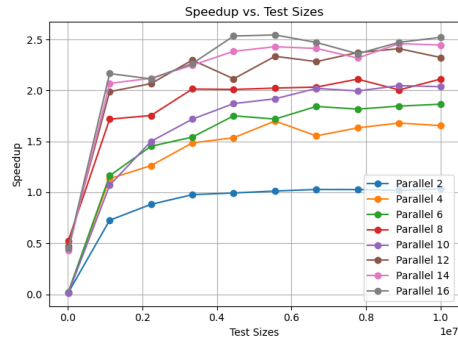


Figure 14: Speedup setup Joblib

Figure 15: Speedup setup

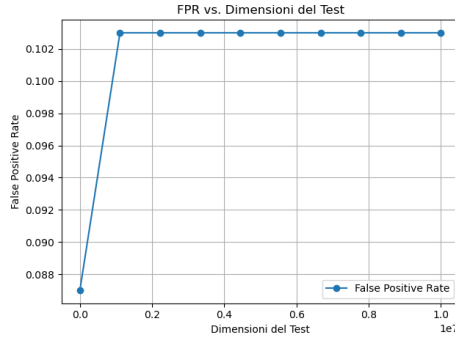


Figure 16: Speedup setup Omp

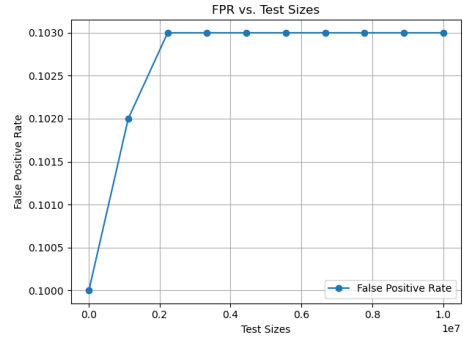


Figure 17: Speedup setup Joblib

Figure 18: Time setup

6.1.3 Chunks

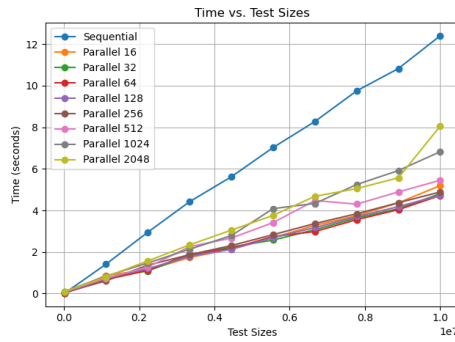


Figure 19: Times setup Chunk

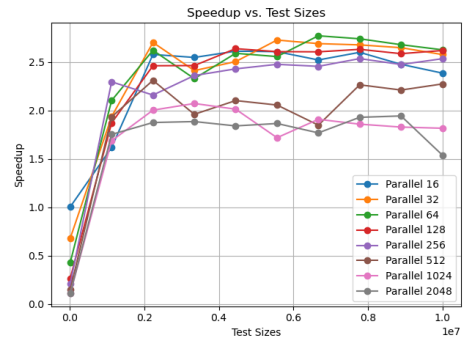


Figure 20: Speedup setup Chunk

Figure 21: Time setup

6.2 FPR: 0.05

6.2.1 Setup

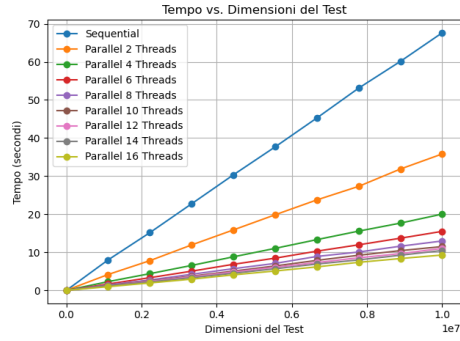


Figure 22: Speedup setup Omp

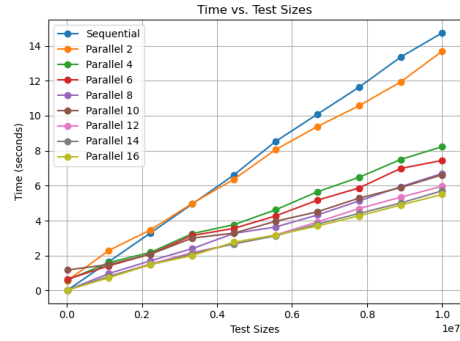


Figure 23: Speedup setup Joblib

Figure 24: Time setup

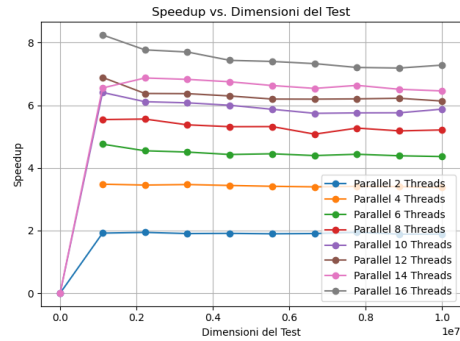


Figure 25: Speedup setup Omp

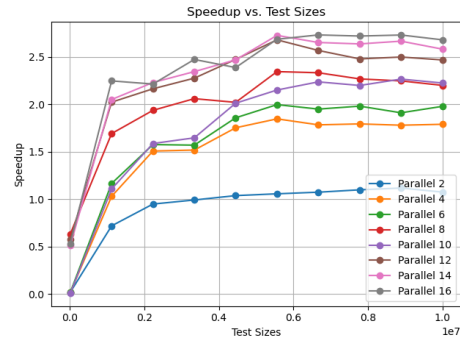


Figure 26: Speedup setup Joblib

Figure 27: Speedup setup

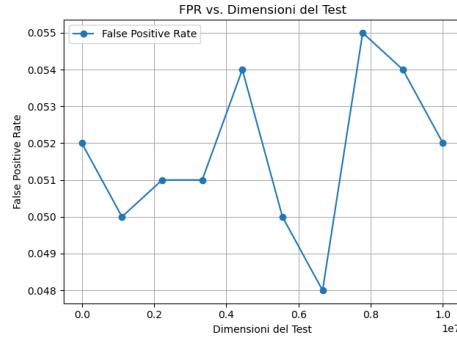


Figure 28: Speedup setup Omp



Figure 29: Speedup setup Joblib

Figure 30: Time setup

6.2.2 Filter

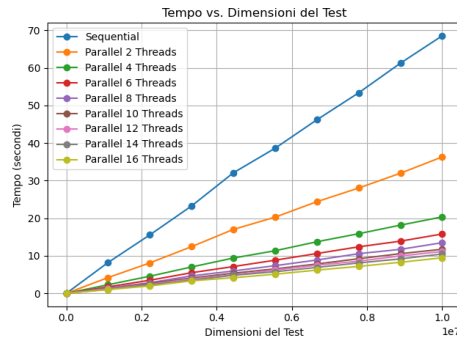


Figure 31: Speedup setup Omp

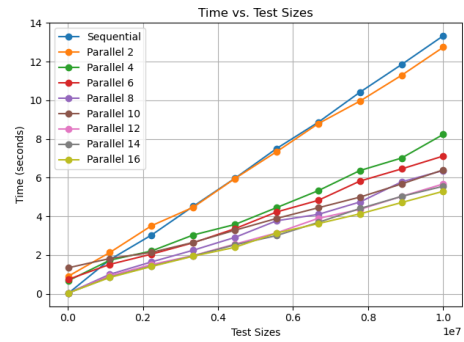


Figure 32: Speedup setup Joblib

Figure 33: Time setup

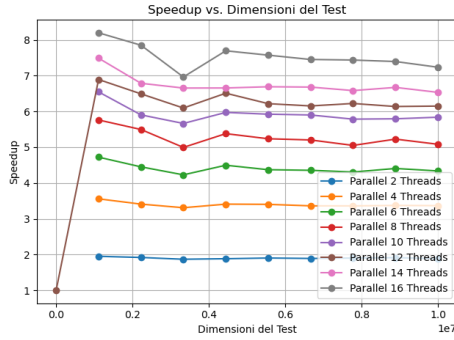


Figure 34: Speedup setup Omp

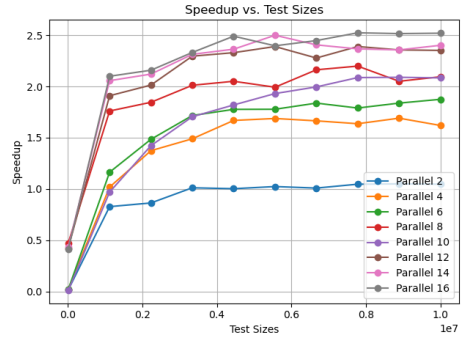


Figure 35: Speedup setup Joblib

Figure 36: Speedup setup

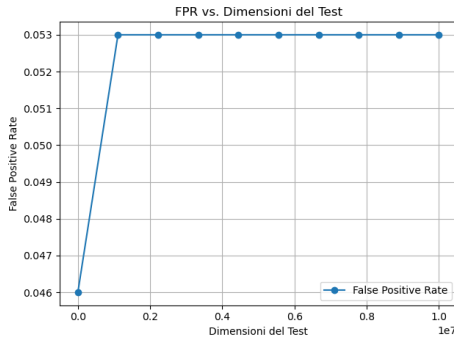


Figure 37: Speedup setup Omp

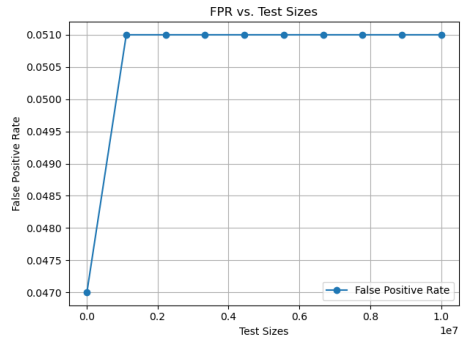


Figure 38: Speedup setup Joblib

Figure 39: Time setup

6.2.3 Chunks

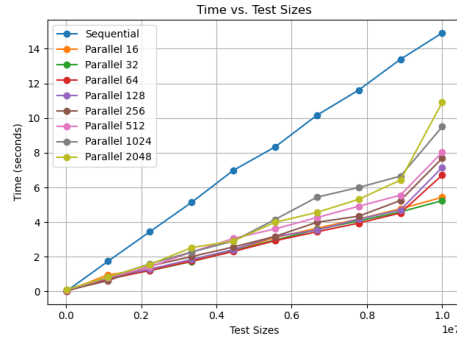


Figure 40: Times setup Chunks

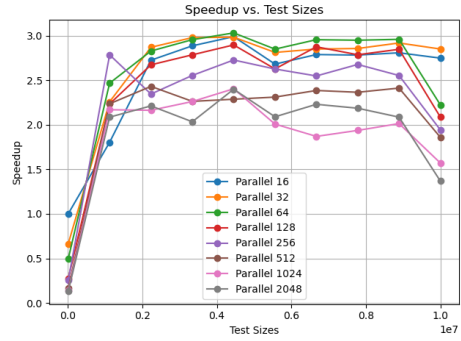


Figure 41: Speedup setup Chunks

Figure 42: Time setup

6.3 FPR: 0.01

6.3.1 Setup

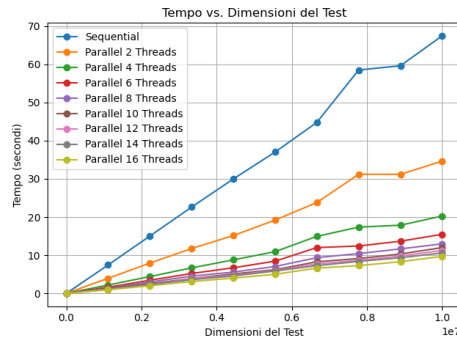


Figure 43: Speedup setup Omp

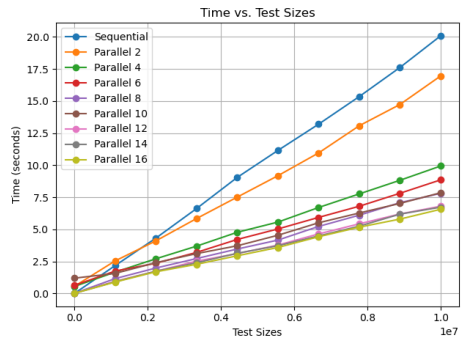


Figure 44: Speedup setup Joblib

Figure 45: Time setup

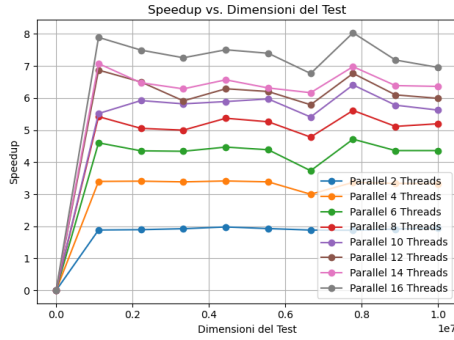


Figure 46: Speedup setup Omp

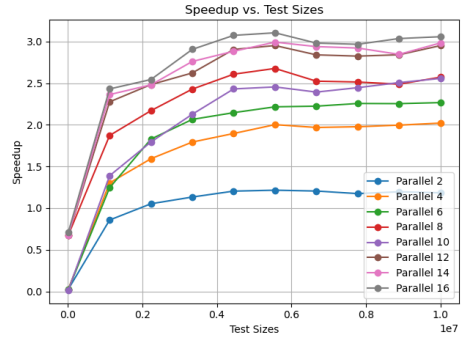


Figure 47: Speedup setup Joblib

Figure 48: Speedup setup

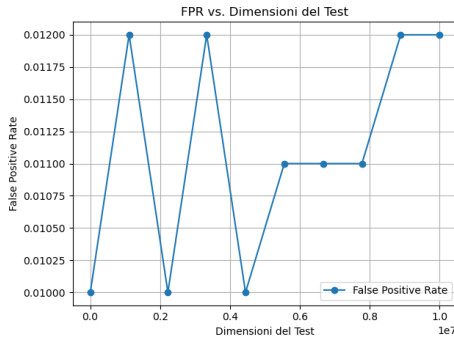


Figure 49: Speedup setup Omp

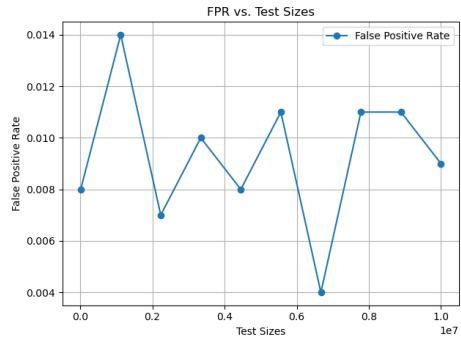


Figure 50: Speedup setup Joblib

Figure 51: Time setup

6.3.2 Filter

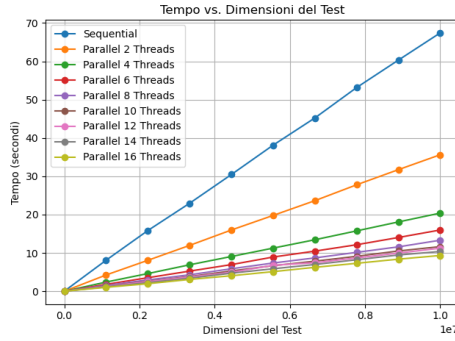


Figure 52: Times filter Omp

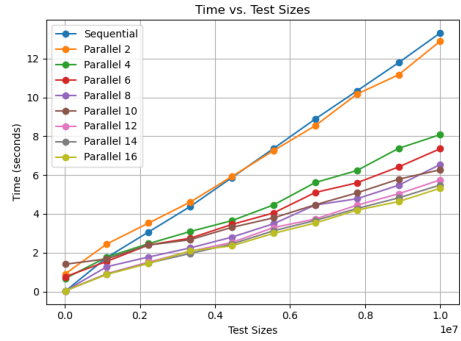


Figure 53: Times filter Joblib

Figure 54: Time filter

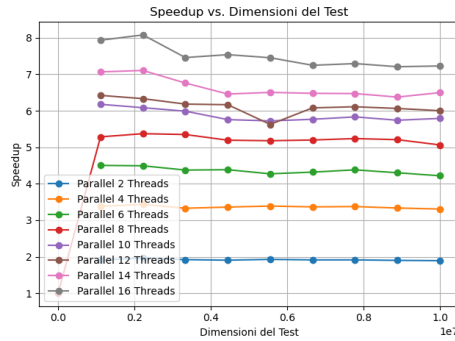


Figure 55: Speedup filter Omp

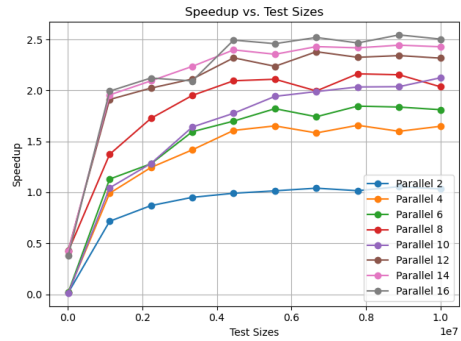


Figure 56: Speedup filter Joblib

Figure 57: Speedup filter

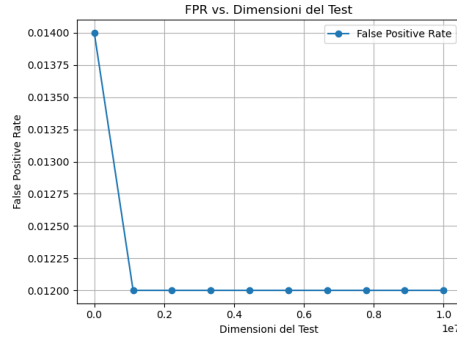


Figure 58: FPR filter Omp

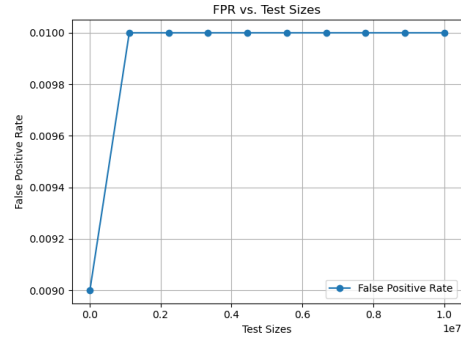


Figure 59: FPR filter Joblib

Figure 60: Time filter

6.3.3 Chunks

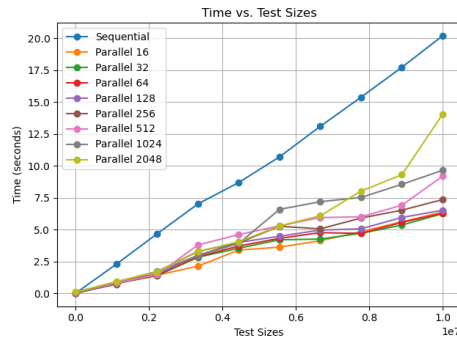


Figure 61: Times chunks Joblib

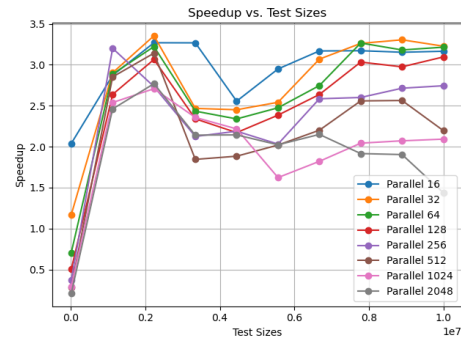


Figure 62: Speedup chunks Joblib

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.

8 Dati