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:

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

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.

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:

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

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

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.

3.2 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.

```
    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)

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

```
    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
```

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

6.1.2 Filter

6.1.3 Chunks

6.2 FPR: 0.05

6.2.1 Setup

6.2.2 Filter

6.2.3 Chunks

6.3 FPR: 0.01

6.3.1 Setup

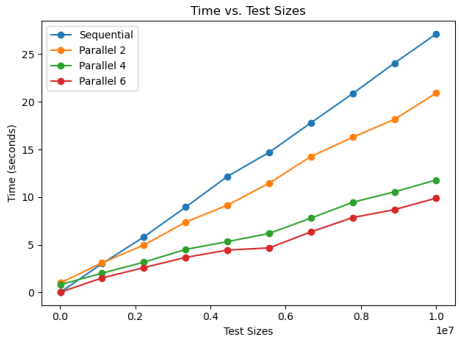

Figure 1: Speedup setup Omp

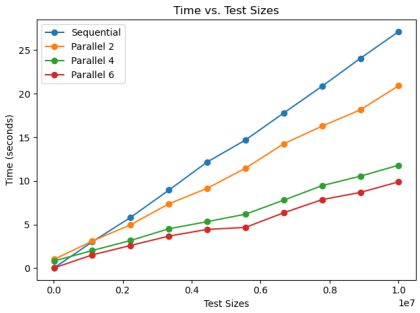

Figure 2: Speedup setup Joblib

Figure 3: Time setup

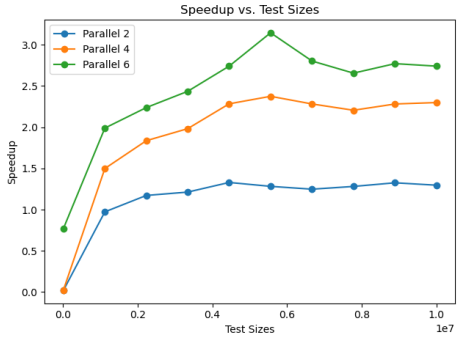

Figure 4: Speedup setup Omp

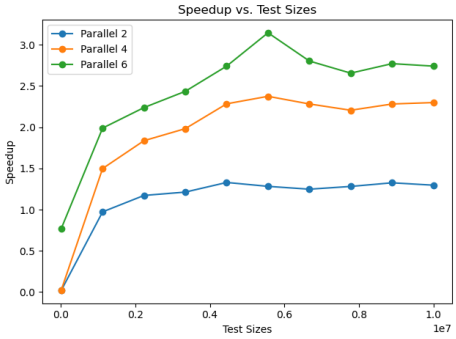

Figure 5: Speedup setup Joblib

Figure 6: Speedup setup

Figure 7: Speedup setup Omp

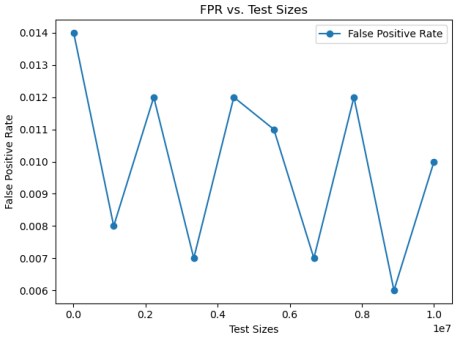

Figure 8: Speedup setup Joblib

Figure 9: Time setup

6.3.2 Filter

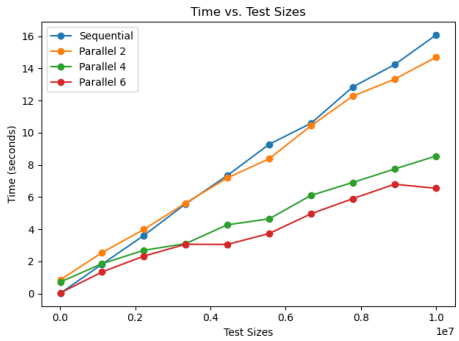

Figure 10: Speedup filter Omp

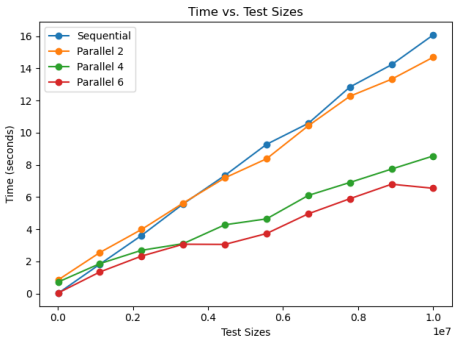

Figure 11: Speedup filter Joblib

Figure 12: Time filter

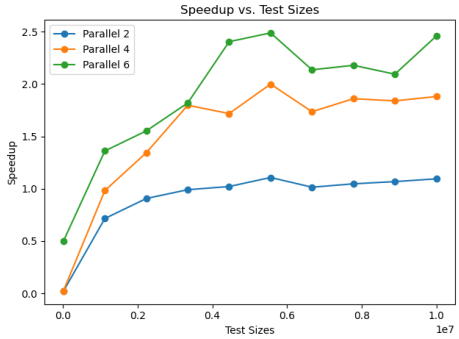

Figure 13: Speedup filter Omp

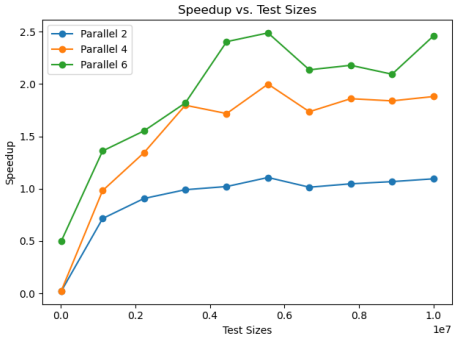

Figure 14: Speedup filter Joblib

Figure 15: Speedup filter

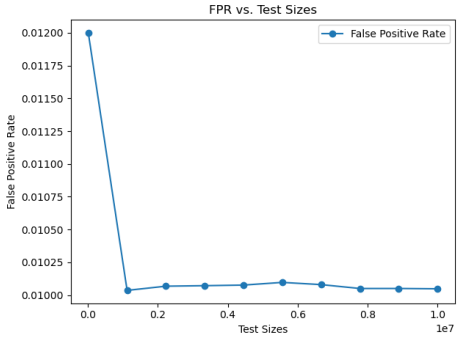

Figure 16: Speedup filter Omp

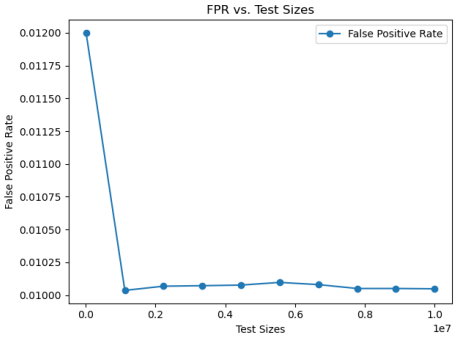

Figure 17: Speedup filter Joblib

Figure 18: Time filter

6.3.3 Chunks

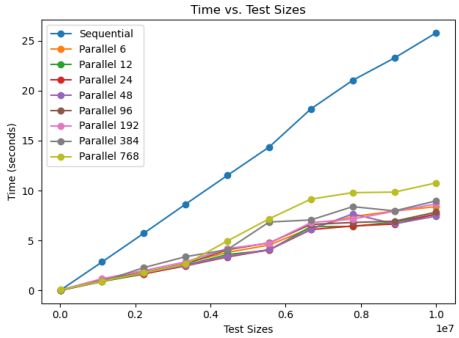

Figure 19: Speedup chunks Omp

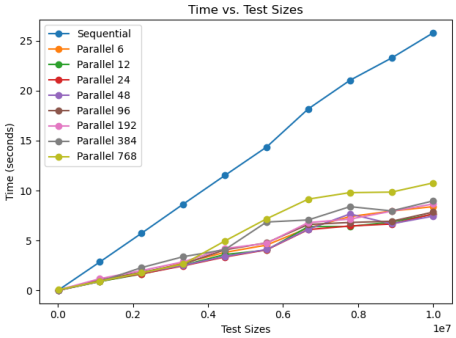

Figure 20: Speedup chunks Joblib

Figure 21: Time chunks

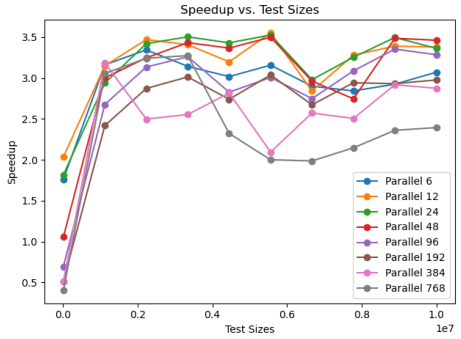

Figure 22: Speedup chunks Omp

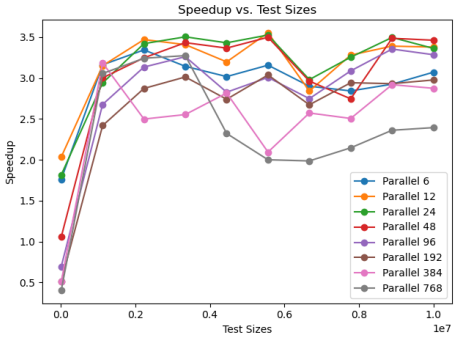

Figure 23: Speedup chunks Joblib

Figure 24: Speedup chunks

Figure 25: Speedup chunks Omp

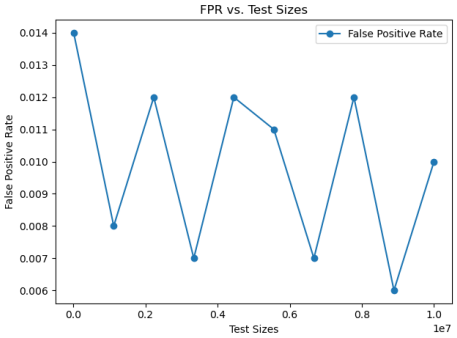

Figure 26: Speedup chunks Joblib

Figure 27: Time 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

test	time$_seq$	fpr	time$_par2$	speedup2	time$_par4$	speedup4	time$_par6$	speedup6
10000	0.020	0.014	1.003	0.020	0.822	0.025	0.027	0.770
1120000	3.023	0.008	3.107	0.973	2.018	1.498	1.520	1.989
2230000	5.824	0.012	4.970	1.172	3.172	1.836	2.603	2.238
3340000	8.948	0.007	7.378	1.213	4.514	1.982	3.675	2.435
4450000	12.171	0.012	9.155	1.329	5.330	2.283	4.441	2.741
5560000	14.695	0.011	11.460	1.282	6.189	2.374	4.673	3.145
6670000	17.796	0.007	14.263	1.248	7.800	2.282	6.350	2.802
7780000	20.860	0.012	16.281	1.281	9.459	2.205	7.856	2.655
8890000	24.063	0.006	18.157	1.325	10.550	2.281	8.685	2.771
10000000	27.127	0.010	20.923	1.296	11.801	2.299	9.899	2.740

test	time$_seq$	fpr	time$_par2$	speedup2	time$_par4$	speedup4	time$_par6$	speedup6
10000	0.018	0.012	0.848	0.021	0.717	0.025	0.036	0.499
1120000	1.824	0.010	2.550	0.715	1.858	0.982	1.341	1.360
2230000	3.611	0.010	3.983	0.907	2.688	1.343	2.327	1.552
3340000	5.573	0.010	5.627	0.991	3.102	1.797	3.065	1.819
4450000	7.341	0.010	7.191	1.021	4.274	1.718	3.054	2.404
5560000	9.277	0.010	8.379	1.107	4.642	1.998	3.730	2.487
6670000	10.580	0.010	10.429	1.014	6.098	1.735	4.955	2.135
7780000	12.841	0.010	12.268	1.047	6.906	1.859	5.895	2.178
8890000	14.228	0.010	13.325	1.068	7.738	1.839	6.797	2.093
10000000	16.081	0.010	14.695	1.094	8.554	1.880	6.543	2.458