BloomFilter Joblib Classificazione email di spam

Parallel Programming for Machine Learning - Baiardi Lorenzo & Thomas Del Moro

Analisi del problema

Il **Bloom Filter** è una struttura dati probabilistica che mira a individuare la presenza di un elemento all'interno di un insieme. Viene utilizzato ad esempio per determinare se un'email è considerata spam o meno nel contesto specifico.

- La fase di configurazione (*setup*) del Bloom Filter coinvolge il calcolo di un certo numero di funzioni hash su ogni elemento dell'insieme ammesso per la costruzione del vettore di bit.
- La fase di filtraggio (*filtering*) consiste invece nel calcolare le stesse funzioni hash sull'elemento da valutare e confrontare i risultati con il vettore di bit in memoria

Codice

Setup e Filtering sequenziali

Nella fase di setup vengono inseriti tutti gli elementi selezionati nel vettore di bit, attraverso il calcolo di più funzioni hash.

Il filtraggio viene eseguito calcolando le stesse hash sugli elementi da valutare e confrontando i bucket ottenuti con quelli già presenti nel vettore.

```python
def add(self, items):
    for item in items:
        for i in range(self.n_hashes):
            index = mmh3.hash(item, i) % self.size
            self.bitarray[index] = True


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

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

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
```

Codice

Setup e Filtering paralleli

Nella fase di setup del vettore di bit, suddividiamo il numero di email in parti uguali, corrispondenti al numero di thread disponibili sulla macchina, assegnando a ciascun thread un chunk specifico.

Durante la fase di filtraggio, ogni thread gestirà un determinato numero di email e calcolerà il numero locale di errori. Successivamente, tutti gli errori generati dai singoli thread verranno sommati per ottenere il numero di errori totale.

```python
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
```

```python
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
```

Codice

Ottimizzazione dei chunk

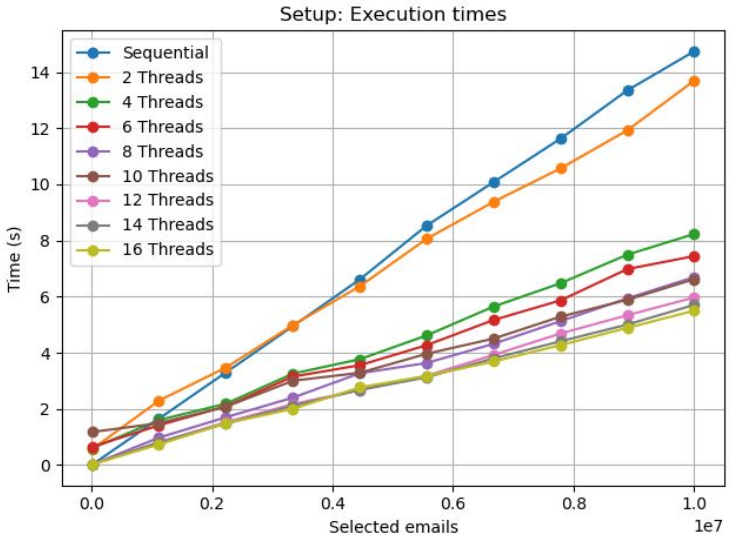
Nel contesto di questa parallelizzazione, all'interno della funzione *par_setup()*, forniamo il numero di "chunks" in cui l'array delle email deve essere suddiviso.

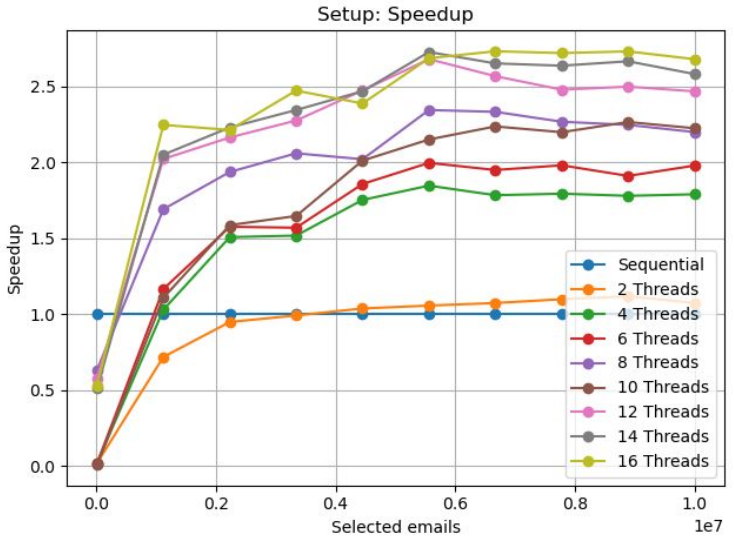
Utilizziamo un numero di "chunks" superiore al numero di thread disponibili sulla macchina per osservare il comportamento al crescere di tale numero.

```
test_chunks = [(n_threads * 2 ** i) for i in range(8)]
par_setup_time = bloom_filter.par_setup(test_emails, n_threads, chunks)
```

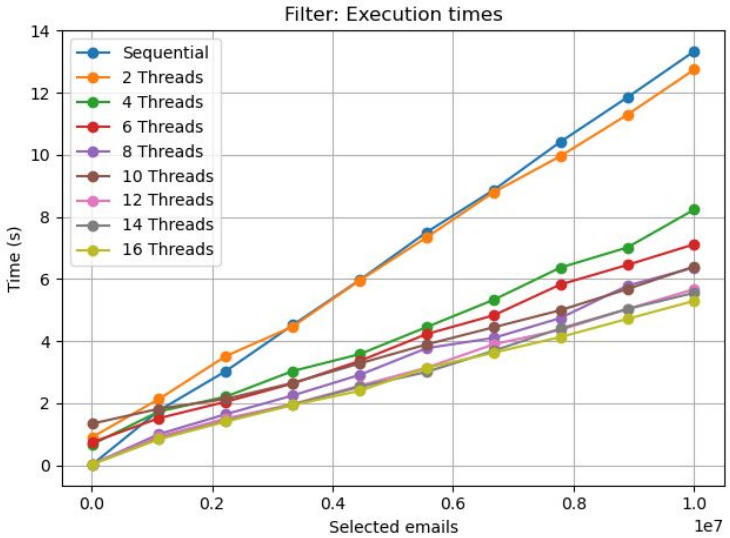
```
def par_setup(self, items, n_threads, chunks=None):
    self.initialize(items)

    # 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
```

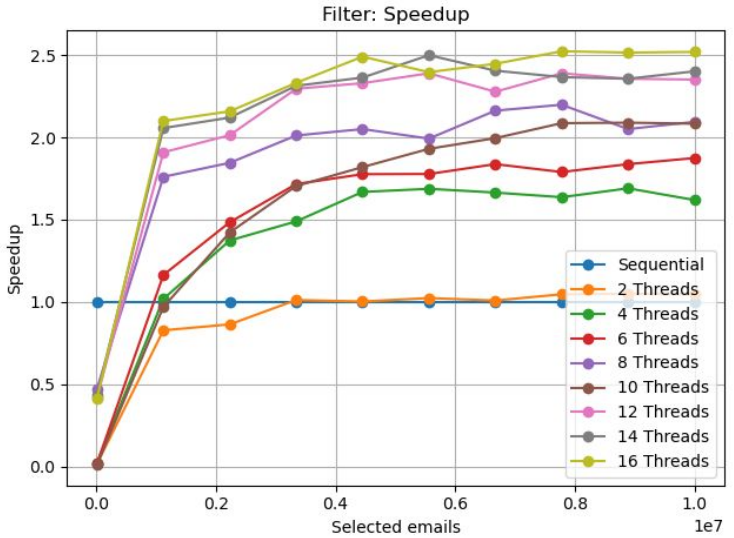
Risultati - Setup FPR 0.05



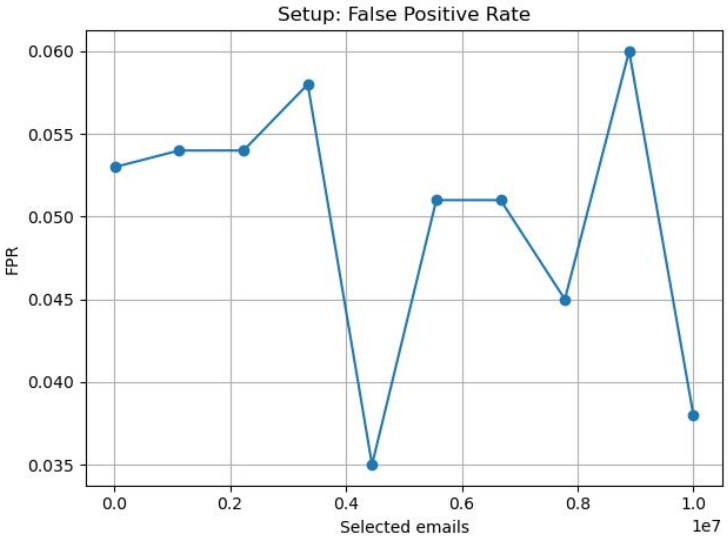


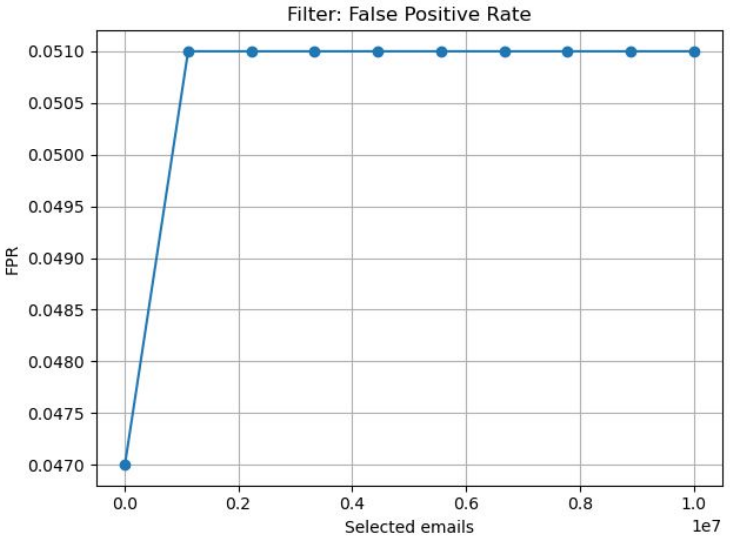
Risultati - Filter FPR 0.05



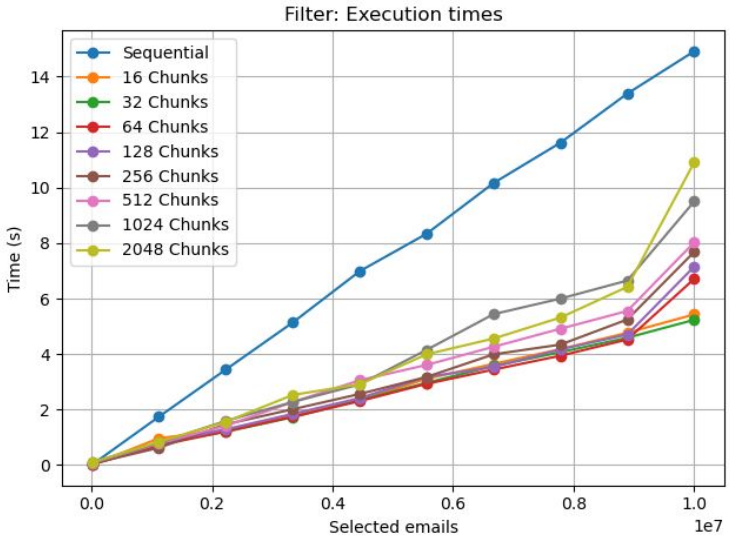


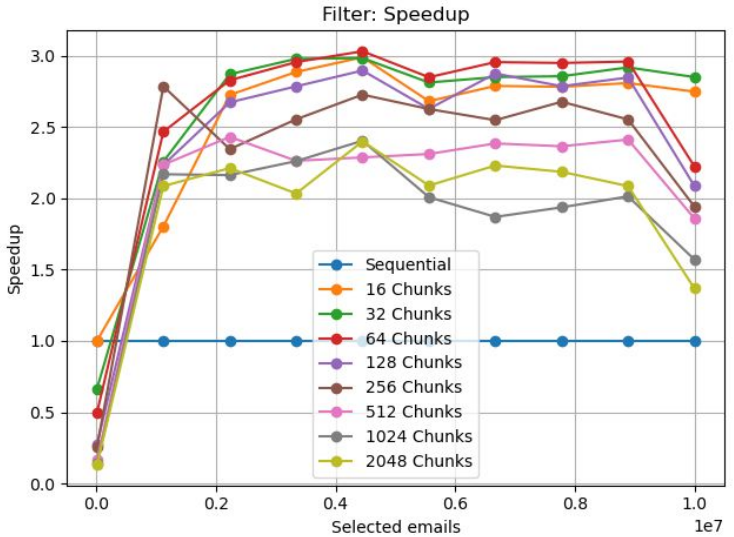
Risultati - FPR Setup & Filter





Risultati - Chunks (16 threads)





Conclusioni

I risultati ottenuti evidenziano che la parallelizzazione delle operazioni di setup e filtraggio del Bloom Filter porta a uno speedup significativo, il quale varia in base al numero di processori impiegati.

A parità di dimensione del vettore, un numero maggiore di thread porta a un maggiore vantaggio computazionale.

L'ottimizzazione della dimensione dei chunk ha contribuito a un lieve incremento dello speedup, soprattutto per determinati valori.

Risulta evidente che a parità di dati processati, OpenMP offre un miglioramente molto maggiore in termini di speedup per questo tipo di problemi.