BloomFilter Openmp 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 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.

```
void BloomFilter::add(const std::string& items) {
    MultiHashes mh(this->size,  email: items);
    for(std::size_t i=0; i < nHashes; i++)
        this->bits[mh()] = true;
}
```

```
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;
}
```

```
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;
}
```

Codice

Setup parallelo

La fase di setup è stata parallelizzata attraverso l'utilizzo di un loop parallelo.

La parallelizzazione è stata fatta sugli elementi da elaborare e non sul calcolo delle diverse hash che avrebbe portato pochi vantaggi.

All'interno del ciclo la direttiva *critical* è stata utilizzata per accedere al vettore di bit in comune.

```
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;
}
```

Codice

Filtering parallelo

Sono state implementate due diverse parallelizzazioni della fase di filtering.

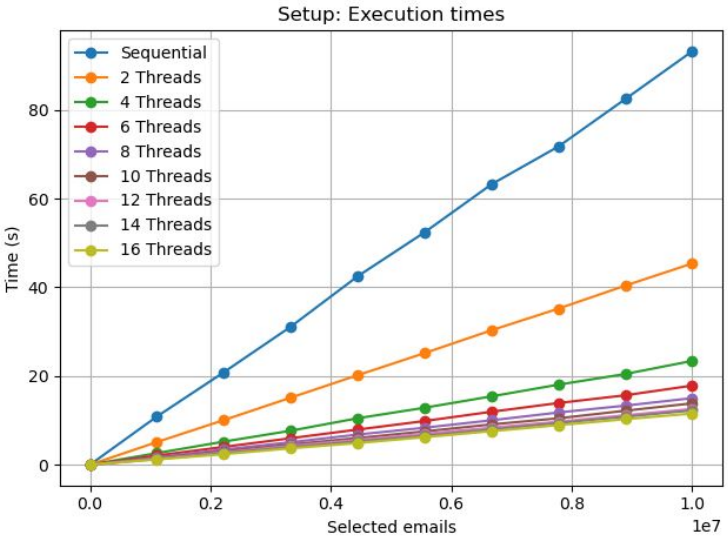
Nel primo caso è stato utilizzato un contatore di errori di tipo *shared*, aggiornato da ogni thread in una sezione *atomic* all'interno del loop parallelo.

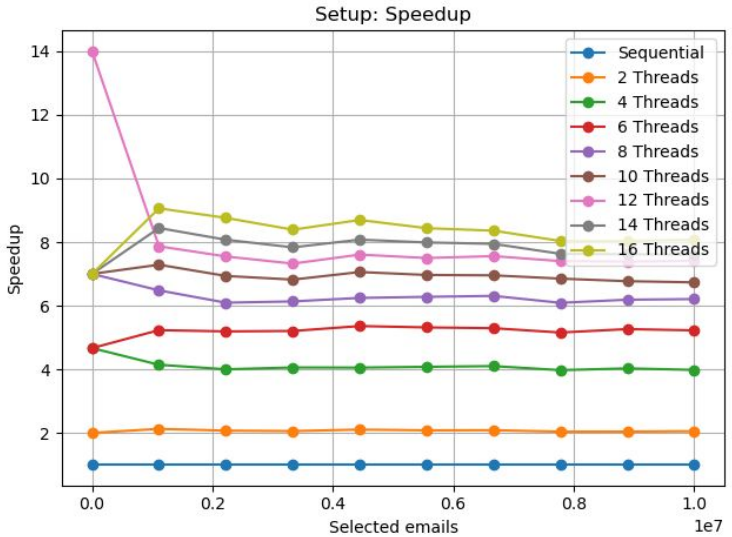
Nel secondo caso ad ogni thread è stata assegnato un contatore di errori *locale*; dopodiché, sempre all'interno di una sezione atomica, è stato calcolato il numero di errori complessivo

```
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;
}
```

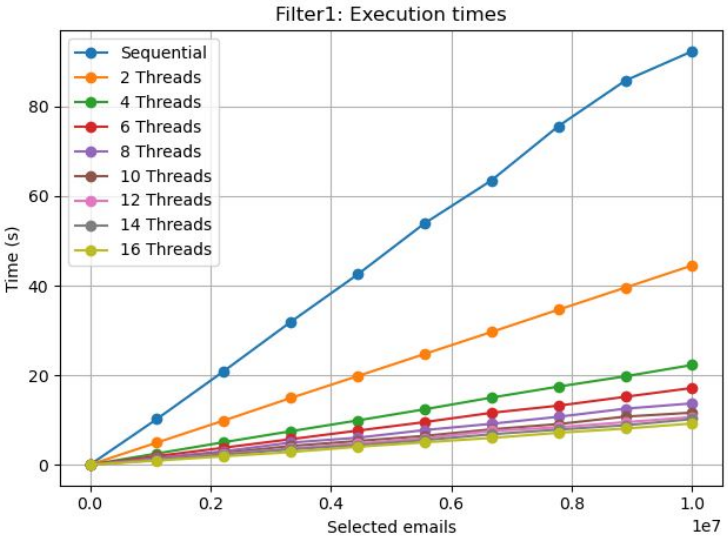
```
int BloomFilter::parallelFilterAll2(std::string items[], size_t nItems) {
    int error = 0;
    int threadError = 0;
#pragma omp parallel default(none) shared(items, error) firstprivate(nItems, threadError)
    {
#pragma omp for nowait
        for (std::size_t i = 0; i < nItems; i++) {
            if (filter(items[i]))
                threadError++;
        }
#pragma omp atomic
        error += threadError;
    }
    return error;
}
```

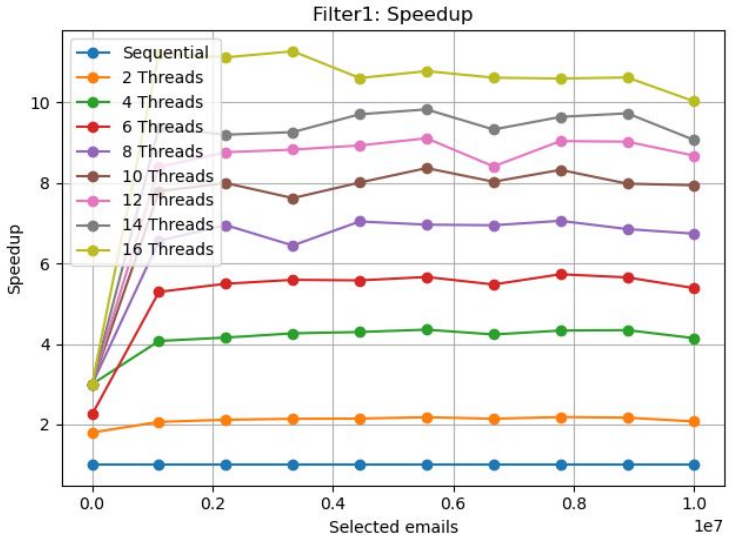
Risultati - Setup FPR 0.05



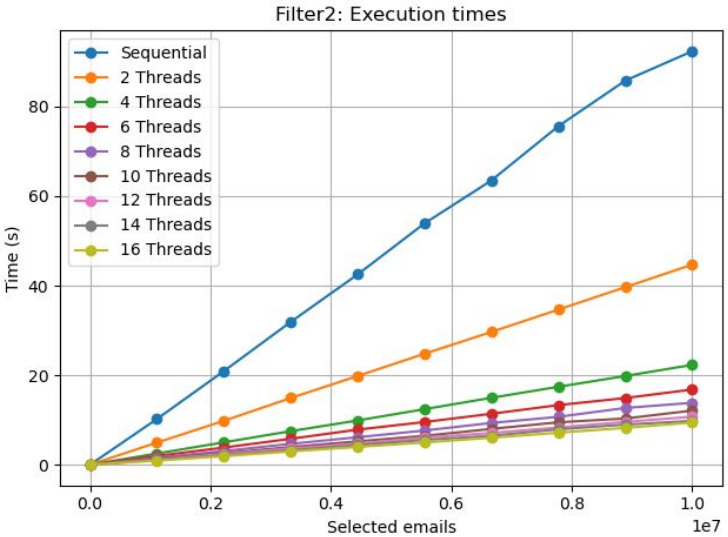


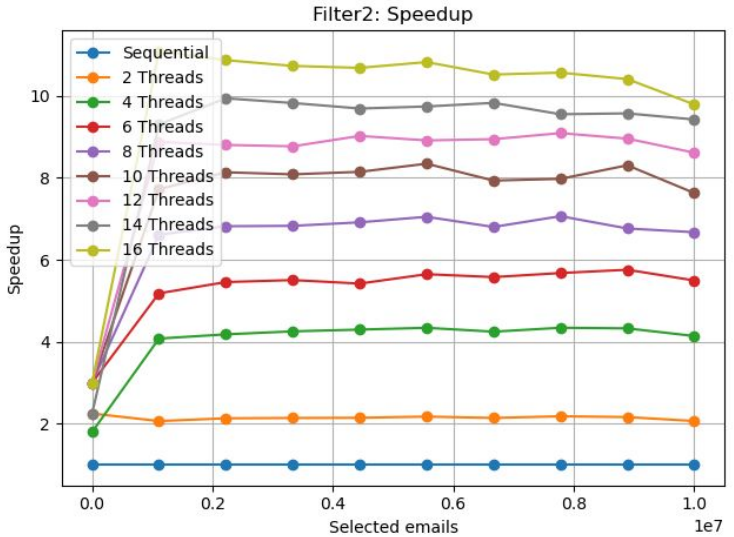
Risultati - Filter1 FPR 0.05



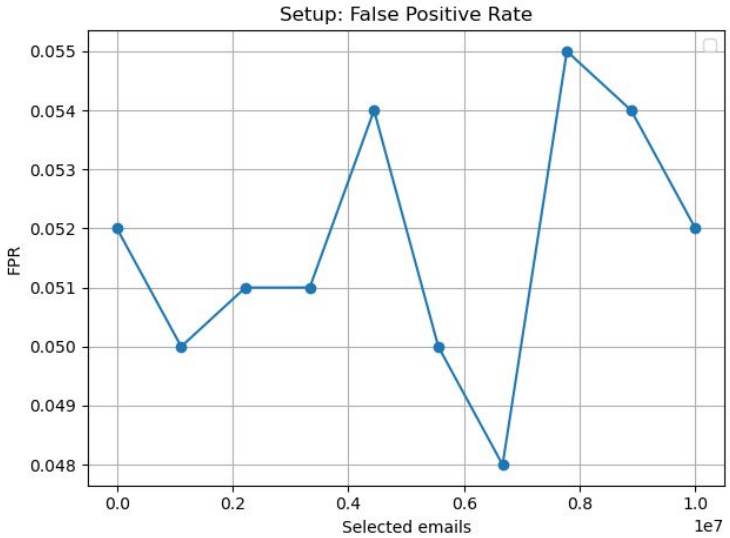


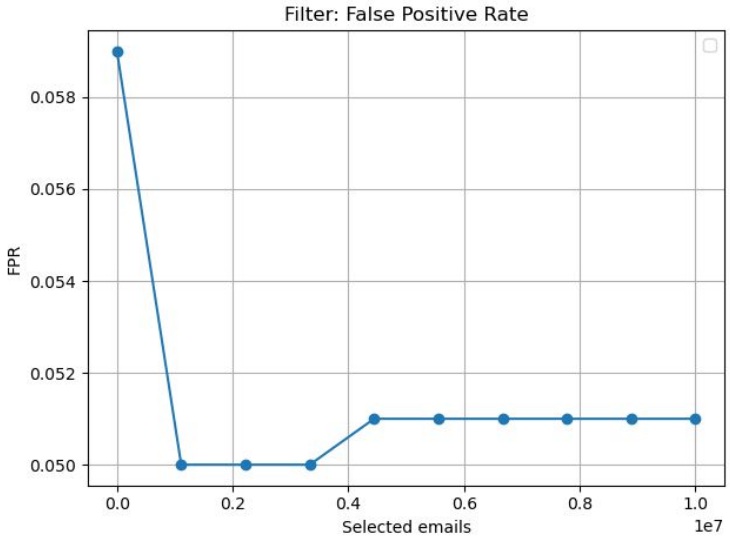
Risultati - Filter2 FPR 0.05





Risultati - FPR Setup & Filter





Conclusioni

- I risultati ottenuti evidenziano che la parallelizzazione con OpenMP delle operazioni di setup e filtraggio del Bloom Filter porta a uno speedup significativo.

- Lo speedup sembra sempre aumentare al crescere del numero di processori utilizzati, notiamo infatti che con 16 thread si raggiunge un miglioramento di un fattore maggiore di 10 nel caso del Filtering in termini di tempo computazionale.

- Le due diverse parallelizzazioni del filtraggio ottengono risultati quasi identici in termini di speedup