Relazione progetto Parallel Computing
Filtro di Bloom

Studente: Massimiliano Sirgiovanni
Matricola: 7077251
Email: massimiliano.sirgiovanni@stud.unifi.it

Come traccia per la realizzazione di questo progetto è stata scelta la realizzazione del **filtro di Bloom**, in maniera sequenziale e parallela. Il linguaggio di programmazione scelto è stato **Python**, prediligendo la libreria **Joblib** come metodo per la parallelizzazione del codice.
Prima di analizzare nel merito il codice scritto, è opportuno esaminare, brevemente, le principali caratteristiche dell'argomento scelto.

Il Filtro di Bloom

Il filtro di Bloom è un metodo di filtraggio dei dati, che si basa su un **metodo probabilistico**.
L'obiettivo è quello di, dati in input degli oggetti, **scartare il maggior numero di elementi** non contenuti in insieme predeterminato ed **accettare tutti gli elementi contenuti in questo insieme**.
Gli strumenti principali per la realizzazione del filtro sono tre:

1. Un **array di n bit** inizialmente tutti posti a zero;
2. Un insieme di **r funzioni hash** *(h0,...,hr)*, ciascuna delle quali trasforma un elemento in un intero compreso tra *0* ed *n-1*;
3. Un insieme *S*, di *m* elementi.

L'applicazione pratica dell'algoritmo, dunque, **valuta le r funzioni hash su tutti gli elementi in S**.
Per ogni applicazione delle r funzioni, **si utilizza il valore come indice per l'array di bit e si pone ad uno il bit corrispondente**.
Una volta inizializzato l'array di bit, per **verificare la presenza in *S*** di un elemento basterà **applicare** a quest'ultimo **le r funzioni hash e verificare se i bit**, *nelle posizioni relative ai valori restituiti dalle funzioni*, **sono 0 o 1**.
Se anche un solo **bit è posto a zero**, allora **l'elemento viene scartato**.
In questo modo, **vi è la certezza che gli elementi contenuti in S superino il filtraggio**, qualora vengano riproposti, mentre **vi è un'alta probabilità che i dati non contenuti in S vengano scartati** in fase di filtraggio.

Implementazione sequenziale

Una volta assodati il funzionamento e gli obiettivi del filtro di Bloom, si possiedono gli strumenti per poter analizzare il programma realizzato.
In precedenza, è stato introdotto il metodo teorico per creare il filtro, ora verranno descritte le scelte implementative per tradurre questo processo in codice.
Il punto di partenza è stato la **creazione dei tre strumenti** necessari per il giusto funzionamento del filtro. Per comodità, è stata definita una variabile globale *"n"* per indicare le dimensioni dell'array di bit.
Una volta assegnato un valore ad *n*, la realizzazione di **un array di *n* bit**, tutti posti a zero, è realizzabile, in python, con una sola istruzione:

*bitArray = [0] * n*

Quest'istruzione, per comodità, non è stata inserita nel main, ma all'interno di una funzione, che verrà esaminata più avanti.
Una volta ottenuto l'array di bit, il secondo strumento necessario sono le **funzioni hash**. Sono state implementate sei funzioni hash, per ognuna è stato realizzato un metodo distinto. Queste funzionano in maniera, più o meno, diversa l'una dalle altre, ma tutte **restituiscono un valore compreso tra *0* ed *n-1***. Per soddisfare quest'ultima condizione, qualsiasi sia l'operazione che venga eseguita prima sui dati, l'ultima sarà il **modulo di n**, *per ogni funzione hash*.
Come **tipo per gli elementi** a cui applicare il filtro si è optato per le **stringhe**, *in modo tale da poter gestire sia valori numerici, sotto forma di stringa, sia parole o simboli speciali*.
Dunque, le sei funzioni hash implementate lavorano principalmente con il **codice ascii** degli elementi. Questi codici poi, a seconda della funzione, vengono concatenati, o sommati, e vengono loro applicate varie funzioni numeriche.
Successivamente si è realizzato un **semplice array di parole**, chiamato *initialSet*, **da sfruttare per inizializzare l'array di bit** *(fungendo il ruolo di insieme S)*, la cui lunghezza può essere variabile a seconda delle necessità dell'utente.

Ottenuti tutti gli strumenti per la creazione di un filtro di Bloom funzionante, si può procedere all'**inizializzazione dell'array di bit**. Questo compito è delegato ad un metodo, ***initializeBitArray(array)***, che verrà poi richiamato nel codice.
Il parametro *array*, da passare alla funzione, **è il vettore *"initialSet"***, rappresentante l'insieme S. **Il metodo restituirà l'array di bit inizializzato**.
La prima istruzione eseguita permette di creare un array di n bit posti a zero *(vista sopra)*. Si procede poi a porre i bit ad uno quando necessario. Per farlo, si è creato un ciclo for, che effettua tante iterazioni quant'è la lunghezza dell'array passato in input *(len(array))*. Dunque, **per ogni elemento dell'array**, passato alla funzione, **viene eseguito un altro metodo**, ***addElement(string, bitArray)***.

Questo secondo metodo prende in input una stringa, che nel caso specifico sarà contenuta nell'*initialSet*, e l'array di bit. Dunque, **esegue la seguente istruzione per ognuna delle sei funzioni hash** definite:

bitArray[h0(string)] = 1

In questo modo, verrà modificato il bit con indice dato dalla funzione hash.
Tramite il ciclo for, sopracitato, **questa funzione viene applicata ad ogni elemento dell'array**, passato come argomento al metodo *initializeBitArray*.
Infine, verrà restituito l'array di bit inizializzato.
Effettuata questa prima operazione, non resta che **creare un metodo che possa applicare il filtraggio** vero e proprio a nuovi elementi.
Questo lavoro viene assegnato a due funzioni: ***CheckElement* e *CheckSet***.
La prima prende in input tre argomenti: ***string***, la stringa a cui applicare il filtro, ***bitArray***, un array di bit *(presumibilmente quello inizializzato in precedenza)*, ed ***onlyPass***, un flag che specifica se effettuare una stampa di tutte le operazioni o solo quando un elemento supera il filtro.
Il metodo consiste nell'**applicazione di una serie di istruzioni if**, una per ogni funzione hash. Si applicano, dunque, tali funzioni all'attributo string e **si verifica se è posto a zero il bit** corrispondente al risultato. Nel caso, il metodo termina la sua esecuzione restituendo il valore 0 *(qualora il flag onlyPass sia posto a zero verrebbe anche effettuata una stampa)*. In caso contrario, invece, verrà stampato un messaggio a schermo e verrà restituito il valore uno.
La funzione ***CheckSet***, d'altra parte, **applica il metodo appena descritto ad un insieme di stringhe**. Di conseguenza, ha come parametri l'**insieme di stringhe** citato, l'**array di bit** ed il flag **onlyPass**. All'interno del metodo viene eseguito un ciclo for che applica la funzione *CheckElement* ad ogni elemento dell'array.
Inoltre, per mezzo di una variabile, **tiene traccia del numero di elementi che riescono a superare il filtro**. Questo numero verrà poi restituito come output della funzione.
Tramite l'applicazione delle funzioni descritte, è possibile applicare il filtro di Bloom, costruito su un insieme generico di elementi, a qualsiasi stringa.
Il programma descritto, tuttavia, **esegue tutte le operazioni in maniera sequenziale.**
Per migliorare il tempo di esecuzione di questo programma, che su dati molto grandi potrebbe essere troppo oneroso, si possono parallelizzare parti del codice.

Implementazione parallela

Per poter gestire al meglio l'utilizzo della programmazione parallela, può essere utile dividere l'esecuzione in due parti: **l'inizializzazione del vettore di bit e l'applicazione del filtro sui nuovi elementi**.
Durante la procedura di inizializzazione, come visto, va creato l'array di bit e bisogna applicare le funzioni hash ad un insieme di elementi prestabiliti, col fine di modificare

l'array. Questa procedura viene eseguita nel metodo *initializeBitArray(array)*, visto in precedenza. Bisogna analizzare il metodo per capire quali parti possano essere parallelizzate.
Ovviamente, **l'array di bit posti a zero viene creato con un'unica istruzione**, impossibile da parallelizzare.
Il miglior candidato per la parallelizzazione è il ciclo for che esegue la funzione *addElement* su tutti gli elementi dell'array passato in input. Tuttavia, vi è una criticità. All'interno della funzione descritta vengono effettuate operazioni di scrittura sull'array di bit.
In un sistema multiprocessing, **ogni processo ha una sua memoria distinta**, dunque per modificare una variabile globale bisognerebbe applicare un sistema di comunicazione e sincronizzazione tra i vari processi. Questo potrebbe risultare più costoso rispetto ad una semplice esecuzione sequenziale, *specie se si ha un numero di dati ridotto*.
Tuttavia, si è trovato un metodo per sfruttare il multiprocessing, implementato dalla libreria **Joblib**, per l'inizializzazione dell'array.
Infatti, **il metodo addElement consta di due fasi**, *che nel codice sequenziale venivano eseguite in un'unica istruzione*. Prima, **viene applicata la funzione hash ad una stringa** in input e poi, dato il risultato della funzione, **viene modificato l'array di bit**. **Si possono separare queste due operazioni per poter parallelizzare parte del codice**.
Infatti, sono stati realizzati **sei cicli paralleli**, che **applicano, ciascuno, una delle funzioni hash ad ogni elemento dell'array in input**. Successivamente, **ogni ciclo restituirà un vettore**, *con stessa lunghezza di quello in input*, **contenente i risultati delle funzioni hash**.

```
ones0 = Parallel(n_jobs=4)(delayed(h0)(array[i]) for i in range(0, len(array)))
ones1 = Parallel(n_jobs=4)(delayed(h1)(array[i]) for i in range(0, len(array)))
ones2 = Parallel(n_jobs=4)(delayed(h2)(array[i]) for i in range(0, len(array)))
ones3 = Parallel(n_jobs=4)(delayed(h3)(array[i]) for i in range(0, len(array)))
ones4 = Parallel(n_jobs=4)(delayed(h4)(array[i]) for i in range(0, len(array)))
ones5 = Parallel(n_jobs=4)(delayed(h5)(array[i]) for i in range(0, len(array)))
```

Una volta ottenuti questi valori, è sufficiente realizzare un **ciclo for per modificare i valori nell'array di bit**. Non è necessario realizzare un ciclo per ogni array, dato che hanno tutti e sei la medesima dimensione.

```
for i in range(0, len(array)):
    bitArray[ones0[i]] = 1
    bitArray[ones1[i]] = 1
    bitArray[ones2[i]] = 1
    bitArray[ones3[i]] = 1
    bitArray[ones4[i]] = 1
    bitArray[ones5[i]] = 1
```

Volendo aggiungere una nuova funzione hash per il filtro, basterebbe inserire un ciclo parallelo identico a quelli visti in precedenza, modificando ovviamente la funzione presa in input, e poi inserire una nuova istruzione per aggiornare l'array di bit, nel ciclo for appena visto.
Una volta inizializzato il vettore di bit, la seconda parte da gestire è il **filtraggio delle stringhe**. Il metodo realizzato per il coordinamento di questa operazione è *checkSet*, visto in precedenza, il quale restituisce il numero di elementi che superano la fase di filtraggio.
Il metodo, nel codice sequenziale, applica ad ogni elemento, del vettore in input, **una funzione**, ***checkElement***. Questa, verifica la presenza di un singolo elemento, *effettuando anche una stampa*, e **restituisce il valore uno quando l'elemento in ingresso passa il filtraggio**, zero altrimenti. Dunque, **per tenere traccia del numero di stringhe accettate dal filtro**, **il metodo viene eseguito all'interno di una somma**:

passed = passed + checkElement(array[i], bitArray, onlyPass)

Come visto nel caso dell'inizializzazione dell'array, è opportuno trovare un modo per eseguire il codice, in parallelo, senza modificare una variabile comune.
Dunque, con questo scopo, **è stato realizzato un ciclo for parallelo che restituisce un array di bit**, di lunghezza pari all'array passato in input. In questo ciclo, viene applicata, ad ogni elemento del vettore in input, la funzione *checkElement*.
Il vettore risultante contiene il valore uno in una posizione solo ed esclusivamente se l'elemento corrispondente è stato accettato dal filtro.

passed = Parallel(n_jobs=4)(delayed(checkElement)(array[i], bitArray) for i in range(0, len(array)))

Per ottenere il numero di elementi che ha superato il filtro è sufficiente sommare tra loro tutti gli elementi dell'array calcolato. La somma viene effettuata in un semplice ciclo for.
Tramite l'utilizzo di questo ciclo parallelo, anche **la stampa nel metodo checkElement viene fatta in parallelo**.
Infatti, nell'esecuzione potrebbe capitare di trovare le stampe in ordine diverso rispetto a quello ottenuto con il programma sequenziale. Tuttavia, nel caso specifico, l'ordine delle stampe non ha alcuna rilevanza.
Una volta parallelizzate queste due parti del codice, si è già ottenuto un programma che permette di eseguire il filtro di Bloom molto più velocemente rispetto al caso sequenziale. Un ultimo aggiustamento può, però, essere attuato all'interno delle funzioni hash. Più nello specifico nella funzione h1. Infatti, le altre cinque funzioni eseguono operazioni molto semplici, in cui la parallelizzazione risulterebbe poco utile o impossibile da applicare. Tuttavia, la funzione h1 contiene un ciclo che restituisce un array di elementi, ottenibili tramite una funzione. Un caso del genere è ottimale per l'applicazione di un ciclo for parallelizzato.

L’obiettivo di tale ciclo è quello di estrarre ogni carattere dalla stringa passata in input e di trasformarlo in codice ascii. Dopodiché, il codice verrà inserito all’interno di un array. Per parallelizzarlo basta applicare un ciclo for parallelo sulla funzione *ord(string)*, la quale restituisce il codice ascii della stringa:

```
result = Parallel(n_jobs=4)(delayed(ord)(string[i]) for i in range(0, len(string)))
```

L’array risultante verrà poi utilizzato esattamente come nel caso sequenziale.

Testing e valutazione delle performance

Il programma sviluppato è molto generico per quanto concerne i dati in ingresso. Infatti, con poca fatica, è possibile modificare la grandezza dell’array di bit, gli elementi contenuti nell’insieme S o gli oggetti a cui applicare il filtro.
Per la procedura di testing che si andrà a descrivere in questa sede, è stato impostato un **valore di n pari a 20000**.
Si passa poi alla creazione dei **due insiemi di stringhe**, uno per **inizializzare l’array di bit** e l’altro **contenente gli oggetti a cui applicare la procedura di filtraggio**.
I due insiemi sono stati realizzati tramite **vettori di stringhe**, **il primo composto da settanta parole ed il secondo da centosessante**. All’interno di quest’ultimo insieme sono state inserite **sei stringhe contenute anche nel primo vettore**, che, di conseguenza, dovrebbero superare senza problemi il filtro.
Vengono poi eseguite le seguenti istruzioni:

```
bitArray = initializeBitArray(initialSet)

    for i in range(0, 10):
        print(f'\n////////////////////// Execution number {i+1}//////////////////////////////////\n')
            print(f'\nOnly {checkSet(newSet, bitArray, 0)} elements CAN PASS of
which 6
            elements were present in the set used for initialization \n')
```

Ovvero, viene prima **inizializzato il vettore di bit** e poi **viene eseguita**, dieci volte, **la procedura di filtraggio** sull’array precedentemente definito.
All’interno del metodo *checkSet*, lo zero passato come argomento indica il flag *onlyPass*, *in modo da ottenere una stampa più semplice e di facile lettura.*

```
//////////////////// Execution number 10/////////////////////////////////////

apple CAN pass!
angry CAN pass!
horse CAN pass!
arrow CAN pass!
dragon CAN pass!
sword CAN pass!
Edward CAN pass!
elf CAN pass!
Checking time: 3.1586170196533203 seconds

Only 8 elements CAN PASS of which 6 elements were present in the set used for initialization
```

Figura 1: Output dell'esecuzione sequenziale

Si può osservare *(Figura 1)* che su centosessanta nuovi elementi, a cui è stato applicato il filtro, solo otto elementi sono stati accettati, di cui sei effettivamente presenti nell'insieme di partenza.
Dunque, **tutti gli elementi contenuti nell'insieme iniziale** (*S*) **hanno superato il filtro**, come richiesto dalla definizione stessa del filtro di Bloom, e **solo due dei rimanenti centocinquantaquattro sono riusciti ad eluderlo**. Indubbiamente, si tratta di un risultato ottimale.
Le istruzioni descritte in precedenza, per la procedura di testing, vengono sì utilizzate nel caso sequenziale, ma **possono essere usate anche per quello parallelo**. Si osservi ora la figura sottostante *(Figura 2)*, la quale riporta il risultato della decima esecuzione della funzione *checkSet(newSet, bitArray, 0),* per il codice parallelizzato, e si confronti con l'esecuzione effettuata sequenzialmente *(Figura 1)*.

```
//////////////////// Execution number 10/////////////////////////////////////

angry CAN pass!
apple CAN pass!
horse CAN pass!
arrow CAN pass!
sword CAN pass!
elf CAN pass!
dragon CAN pass!
Edward CAN pass!
Checking time: 1.039581298828125 seconds

Only 8 elements CAN PASS of which 6 elements were present in the set used for initialization
```

Figura 2: Output dell'esecuzione in parallelo

Come si può notare, **l'unica differenza tra le due esecuzioni**, *eccetto il tempo impiegato di cui si discuterà a breve*, **è l'ordine delle stampe**. Ovviamente, nel caso sequenziale, per ognuna delle dieci esecuzioni, l'ordine di stampa sarà sempre

quello visto in *Figura 1*. Per quanto riguarda il codice parallelo, **l'ordine di stampa varia tra le diverse esecuzioni**, anche a parità di dati.
Naturalmente, sia per l'esecuzione sequenziale che per quella parallela, gli elementi che superano il filtro sono sempre gli stessi, con le medesime caratteristiche.

Studiato l'output delle due versioni del programma, **fondamentale è lo studio delle loro performance**. Infatti, sono in questo modo si potrà capire se le procedure di parallelizzazione sono state ben realizzate e se sono utili nell'esecuzione del programma. Col fine di valutare le performance, **sono state inserite le seguenti istruzioni a circondare i punti salienti del codice**.

```
sync_start = time.time()
# Code to evaluate
sync_end = time.time()
print(f'Running time: {sync_end - sync_start} seconds')
```

In questo modo, **è possibile ottenere il tempo impiegato per eseguire una determinata parte di codice**. Nel caso studiato, **queste istruzioni sono state inserite all'interno del metodo *initializeBitArray(array)*,** *per ottenere il tempo impiegato per l'inizializzazione dell'array di bit,* **nel metodo *checkSet(array, bitArray, onlyPass)*,** *per verificare quanto tempo sia necessario per applicare il filtro ad un insieme,* **ed infine viene posto nel "main"***, per calcolare il tempo di esecuzione dell'intero programma*.
Naturalmente, questi tre valori vengono stampati a schermo come output del programma.
Si può osservare, in *Figura 3*, il tempo impiegato per l'inizializzazione nel caso sequenziale. In questo caso si superano gli otto secondi per inizializzare l'array di bit.

```
////////////////////START////////////////////////////

Initialization Set contain 70 elements
The new set contain 160 elements

Initialization Time: 8.116327047348022 seconds
```

Figura 3: Tempo di inizializzazione sequenziale

Passando all'esecuzione parallela, si noterà *(Figura 4)* **un tempo di esecuzione pari quasi alla metà di quello ritrovato nel caso sequenziale**.

```
////////////////////START////////////////////////////

Initialization Set contain 70 elements
The new set contain 160 elements

Initialization Time: 4.333853483200073 seconds
```

Figura 4: Tempo di inizializzazione parallela

Utilizzando quattro processi ci si aspetterebbe un tempo minore. Tuttavia, vi è da ricordare che **non tutte le operazioni** eseguite per inizializzare l'array **sono state parallelizzate**.

Vi è ancora un ciclo for, *nonché alcune parti nell'esecuzione delle funzioni hash*, che vengono eseguiti sequenzialmente. Dunque, è giustificato il valore del tempo di esecuzione. Stesso discorso vale per le operazioni successive.

Possiamo valutare lo **speed up** per l'esecuzione della sola fase di inizializzazione dell'array di bit in questo modo:

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{8.11}{4.33} \simeq 1.88$$

Come anticipato, in questo caso lo speedup è circa pari a due, ovvero **il programma eseguito in parallelo impiega circa la metà del tempo**, per inizializzare l'array di bit, **rispetto a quello sequenziale**.
Passando alla fase successiva, in cui si ha un codice in parte parallelizzato, si ottiene, tendenzialmente, uno speedup di valore, circa, pari a tre *(come è visibile anche nell'esempio mostrato nelle Figure 1 e 2).*
Si osservino le due seguenti figure *(5 e 6)*, nelle quali si possono osservare, rispettivamente, un output di un'esecuzione sequenziale ed una parallela.

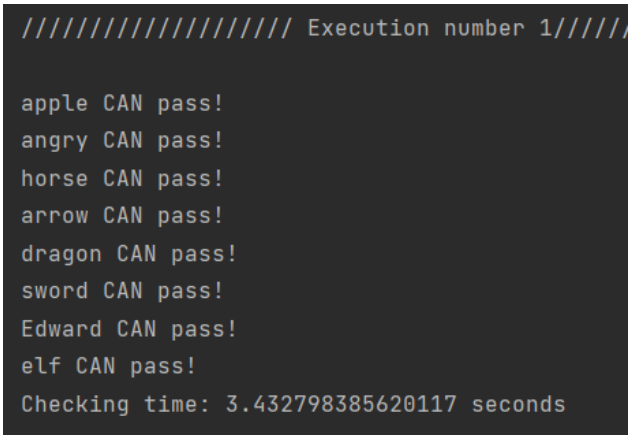

Figura 5: Output verifica degli elementi (sequenziale)

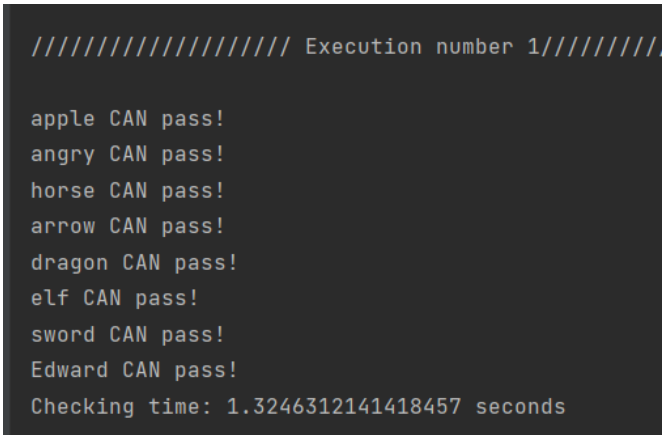

Figura 6: Output verifica degli elementi (parallela)

Applicando la formula dello **speed up**, vista in precedenza, si ottiene, come previsto:

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{3.43}{1.32} \simeq 2.60$$

L'esecuzione del metodo in parallelo è, quasi, **tre volte più veloce rispetto all'esecuzione sequenziale**.

Nonostante, per osservare l'output dell'algoritmo parallelo, fosse più utile effettuare diverse volte la fase di checking, per valutare la velocità di esecuzione è preferibile eseguire una sola volta il metodo *checkSet*. Per farlo è sufficiente modificare il limite massimo del ciclo for.

I risultati ottenuti sono visibili nelle seguenti immagini *(Figura 7 e Figura 8)*.

```
Initialization Set contain 70 elements
The new set contain 160 elements

Initialization Time: 8.275121450424194 seconds

////////////////////// Execution number 1/////////////////////////////////////

apple CAN pass!
angry CAN pass!
horse CAN pass!
arrow CAN pass!
dragon CAN pass!
sword CAN pass!
Edward CAN pass!
elf CAN pass!
Checking time: 3.432798385620117 seconds

Only 8 elements CAN PASS of which 6 elements were present in the set used for initialization

Running time: 11.707919836044312 seconds

Process finished with exit code 0
```

Figura 7: Esecuzione sequenziale dell'intero programma

```
//////////////////////START////////////////////////////////

Initialization Set contain 70 elements
The new set contain 160 elements

Initialization Time: 4.332472085952759 seconds

////////////////////// Execution number 1/////////////////////////////////////

apple CAN pass!
angry CAN pass!
horse CAN pass!
arrow CAN pass!
dragon CAN pass!
elf CAN pass!
sword CAN pass!
Edward CAN pass!
Checking time: 1.3246312141418457 seconds

Only 8 elements CAN PASS of which 6 elements were present in the set used for initialization

Running time: 5.6571033000946045 seconds

Process finished with exit code 0
```

Figura 8: Esecuzione Parallela dell'intero programma

I valori restituiti possono variare leggermente effettuando altre esecuzioni o eseguendo il programma su un'altra macchina.
Si può calcolare lo **speed up** come visto in precedenza:

$$SpeedUp = \frac{Ts}{Tp} \simeq \frac{11.71}{5.66} \simeq 2.07$$

Eseguendo il programma in parallelo, dunque, **si riescono ad ottenere i medesimi risultati nella metà del tempo**, rispetto al programma sequenziale.
Sicuramente, per casi reali, dunque molto più grandi di questo semplice test, l'utilizzo di tecniche di programmazione parallela potrebbe portare ad un risparmio di tempo estremamente significativo.