UNIVERSITÀ
DEGLI STUDI
FIRENZE

Bloom filter

Progetto di Parallel Computing

a cura di Jacopo Manetti

UNIVERSITÀ DEGLI STUDI FIRENZE

Il Progetto

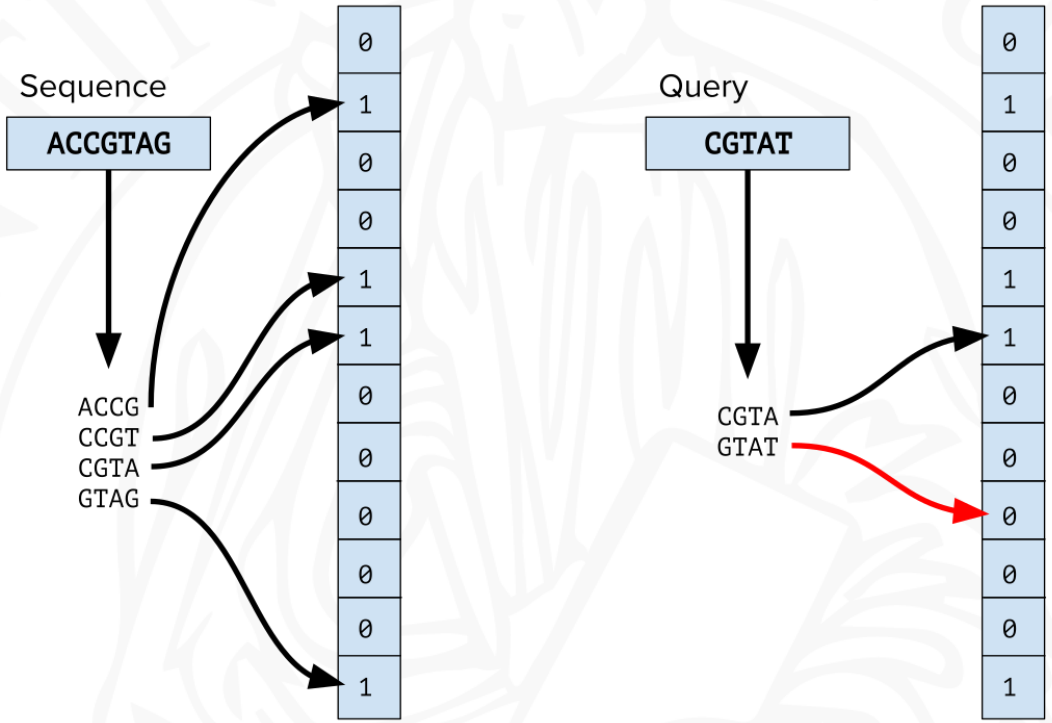
Il progetto ha confrontato le prestazioni dell'algoritmo Bloom Filter in versione parallela e sequenziale utilizzando **Python** e la libreria **Joblib**, su un computer dotato di un processore **Intel i7 11800h a 8 core** (16 logici). Sono state utilizzate tecniche di profilazione per valutare le prestazioni e i risultati hanno dimostrato che la versione parallela ha permesso uno speed up medio di circa 5 rispetto alla versione sequenziale.

Dato che uno degli utilizzi del bloom filter è l'individuazione di url maligni, i test utilizzano liste di url, tuttavia il codice scritto è corretto per qualsiasi altra applicazione generale.

L’algoritmo Bloom filter

L'algoritmo Bloom Filter è una struttura dati probabilistica utilizzata per verificare se un elemento è membro di un set.

- Quando un elemento viene aggiunto al filtro, le funzioni hash vengono applicate all'elemento e i bit corrispondenti alle posizioni indicate dalle funzioni hash vengono impostati a 1.
- Quando viene effettuata una verifica per vedere se un elemento è membro del set, le funzioni hash vengono applicate all'elemento e viene controllato se tutti i bit corrispondenti alle posizioni indicate dalle funzioni hash sono impostati a 1.
- Il filtro di Bloom può restituire falsi positivi, ovvero indicare che un elemento appartiene al set quando in realtà non è presente. La probabilità di falsi positivi dipende dalla dimensione del filtro e dal numero di elementi inseriti nel set.



UNIVERSITÀ DEGLI STUDI FIRENZE

Il codice sequenziale

Il codice utilizza la libreria hashlib per creare hash sha256 e utilizza la funzione hash_calculate per calcolare l'indice in bit array per ogni URL, questa funzione riceve in input oltre alla stringa a cui applicare l'hash anche 2 valori interi, il primo n serve per applicare il modulo al valore hash restituito in modo che il valore rientri nella lunghezza dell'array di bit, il secondo j serve a cambiare indice della funzione hash in modo che a ogni iterazione su j la funzione hash applicata restituisca un valore diverso, in questo modo per ogni stringa si possono applicare più funzioni hash diverse.

```python
def hash_calculate(string, n, j):
    sha256 = hashlib.sha256()
    # Aggiorna l'oggetto sha256 con la stringa e l'indice i
    sha256.update((string + str(j)).encode('utf-8'))
    return int(sha256.hexdigest(), 16) % n
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Il codice sequenziale

```python
class BloomFilter:



    def __init__(self, size, hash_count):
        # Crea un array di bit di dimensione size, inizialmente tutti impostati su zero
        self.bit_array = [0] * size
        self.size = size
        self.hash_count = hash_count


    def initialize(self, urls):
        # Itera per il numero di volte specificato nella proprietà hash_count
        for j in range(self.hash_count):
            # Utilizza la funzione hash_calculate per ogni url
            # e aggiorna il filtro
            for i in range(len(urls)):
                self.bit_array[hash_calculate(urls[i], self.size, j)] = 1


    def check(self, url):
        for i in range(self.hash_count):
            if self.bit_array[hash_calculate(url, self.size, i)] == 0:
                return False
        return True
```

__init__ : inizializza a 0 una struttura di bit array di dimensione ”size” e imposta il numero di funzioni hash da utilizzare.

initialize: riceve una lista di URL e utilizza la funzione ”hash calculate” su ogni elemento per impostare a 1 i relativi indici.

check: riceve un URL e utilizza la funzione ”hash calculate” per ottenere l’indice in bit array. Se anche 1 tra i valori in bit array corrispondenti a questi indici sono a 0, l’URL non fa parte della lista iniziale e viene restituito False. In caso contrario, viene restituito True.

UNIVERSITÀ DEGLI STUDI FIRENZE

Il codice parallelo

```python
class ParallelBloomFilter:



    def __init__(self, size, hash_count, n_thread):
        # Crea un array di bit di dimensione size, inizialmente tutti impostati su zero
        self.bit_array = [0] * size
        self.size = size
        self.hash_count = hash_count
        self.n_thread = n_thread

    def initialize(self, urls):
        # Crea una lista vuota per contenere i risultati
        result = []
        with Parallel(n_jobs=self.n_thread) as parallel: #per riutilizzare lo stesso pool di thread
            # Itera per il numero di volte specificato nella proprietà hash_count
            for j in range(self.hash_count):
                result.extend( #parallelizzo applicando l'hash a più elementi della lista di url
                               #con delayed aspetto che hash_calculate venga applicato a ogni elemento della
                               #lista prima di passare all'hash successivo
                    parallel(delayed(hash_calculate)(urls[i], self.size, j)
                    for i in range(len(urls)))
                    )

        # Itera per ogni elemento nella lista result
        for i in range(len(result)):
            # Imposta l'elemento corrispondente nella bit_array a 1
            self.bit_array[result[i]] = 1
```

Al costruttore **__init__** è stato aggiunto un nuovo parametro n_thread per modificare il numero di thread da usare durante la parallelizzazione.

Dato che è impossibile parallelizzare direttamente il metodo **initialize()** a causa della zona condivisa nell'array di bit.

È necessario dividere il metodo in 2 parti usando un array d'appoggio. Viene eseguito il calcolo della funzione hash in parallelo su tutti gli URL e vengono salvati i risultati nell'array result, che poi viene usato per inizializzare il filtro in modo sequenziale.

UNIVERSITÀ DEGLI STUDI FIRENZE

Il codice parallelo

```python
class ParallelBloomFilter:



    def __init__(self, size, hash_count, n_thread):
        # Crea un array di bit di dimensione size,
        self.bit_array = [0] * size
        self.size = size
        self.hash_count = hash_count
        self.n_thread = n_thread

    def initialize(self, urls):
        # Crea una lista vuota per contenere i risultati
        result = []
        with Parallel(n_jobs=self.n_thread) as parallel: #per riutilizzare lo stesso pool di thread
            # Itera per il numero di volte specificato nella proprietà hash_count
            for j in range(self.hash_count):
                result.extend( #parallelizzo applicando l'hash a più elementi della lista di url
                               #con delayed aspetto che hash_calculate venga applicato a ogni elemento della
                               #lista prima di passare all'hash successivo
                    parallel(delayed(hash_calculate)(urls[i], self.size, j)
                    for i in range(len(urls)))
                    )

        # Itera per ogni elemento nella lista result
        for i in range(len(result)):
            # Imposta l'elemento corrispondente nella bit_array a 1
            self.bit_array[result[i]] = 1
```

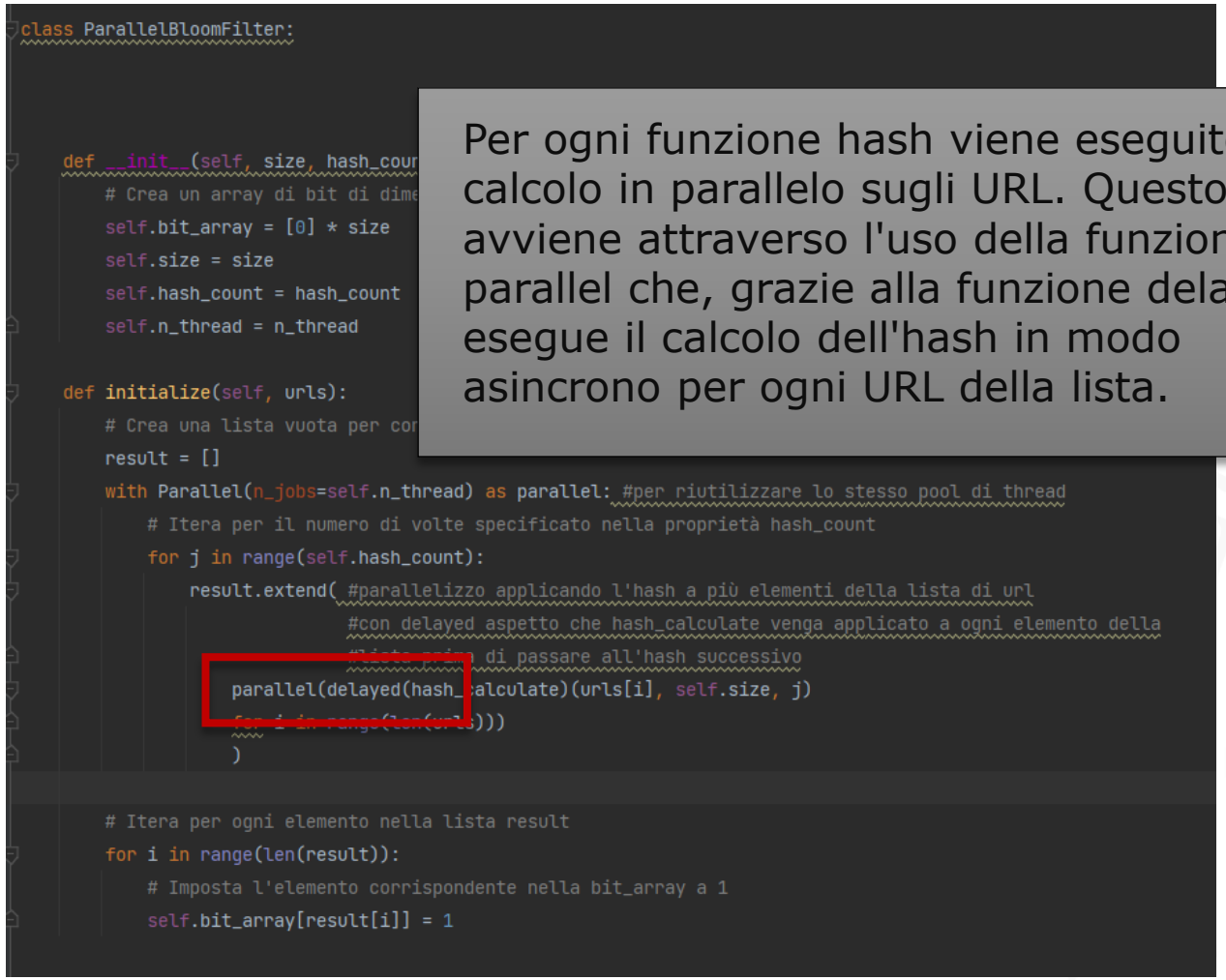
L'intero calcolo parallelo si trova all'interno del context manager API della classe Parallel per evitare di generare e distruggere i thread ad ogni nuova funzione di hash applicata.

Al costruttore **__init__** è stato aggiunto un nuovo parametro per modificare il numero a usare durante la zione.

impossibile parallelizzare te il metodo **initialize()** a causa della zona condivisa nell'array di bit.

È necessario dividere il metodo in 2 parti usando un array d'appoggio. Viene eseguito il calcolo della funzione hash in parallelo su tutti gli URL e vengono salvati i risultati nell'array result, che poi viene usato per inizializzare il filtro in modo sequenziale.

Il codice parallelo



```
class ParallelBloomFilter:



    def __init__(self, size, hash_cour
        # Crea un array di bit di dime
        self.bit_array = [0] * size
        self.size = size
        self.hash_count = hash_count
        self.n_thread = n_thread

    def initialize(self, urls):
        # Crea una lista vuota per cor
        result = []
        with Parallel(n_jobs=self.n_thread) as parallel: #per riutilizzare lo stesso pool di thread
            # Itera per il numero di volte specificato nella proprietà hash_count
            for j in range(self.hash_count):
                result.extend( #parallelizzo applicando l'hash a più elementi della lista di url
                               #con delayed aspetto che hash_calculate venga applicato a ogni elemento della
                               #lista prima di passare all'hash successivo
                    parallel(delayed(hash_calculate)(urls[i], self.size, j)
                    for i in range(len(urls)))
                    )

        # Itera per ogni elemento nella lista result
        for i in range(len(result)):
            # Imposta l'elemento corrispondente nella bit_array a 1
            self.bit_array[result[i]] = 1
```

Per ogni funzione hash viene eseguito il calcolo in parallelo sugli URL. Questo avviene attraverso l'uso della funzione parallel che, grazie alla funzione delayed, esegue il calcolo dell'hash in modo asincrono per ogni URL della lista.

Al costruttore **__init__** è stato aggiunto un nuovo parametro hread per modificare il numero hread da usare durante la allelizzazione.

o che è impossibile parallelizzare ttamente il metodo **initialize()** a sa della zona condivisa nell'array di bit.

È necessario dividere il metodo in 2 parti usando un array d'appoggio. Viene eseguito il calcolo della funzione hash in parallelo su tutti gli URL e vengono salvati i risultati nell'array result, che poi viene usato per inizializzare il filtro in modo sequenziale.

Il codice parallelo

```
def check(self, url):
    for i in range(self.hash_count):
        if self.bit_array[hash_calculate(url, self.size, i)] == 0:
            return False
    return True


def parallelCheck(self, urls):
    result = []
    result.extend(Parallel(n_jobs=self.n_thread)(delayed(self.check)(urls[i])
                                                 for i in range(len(urls))))
    # parallelizzo applicando contemporaneamente la stessa funzione per il controllo della stringa a più parole
    # in contemporanea
    return result
```

La seconda parte che può essere parallelizzata è il check degli elementi, per farlo viene creato un nuovo metodo **ParallelCheck()** che semplicemente applica il metodo check a tutti gli elementi della lista di URL in parallelo per ogni funzione hash.

UNIVERSITÀ DEGLI STUDI FIRENZE

Il codice – test di funzionamento

Versione sequenziale

```
bf = BloomFilter(1000, 10)

start_time = time.time()
bf.initialize(url_maligni)
end_time = time.time()
print("Tempo di esecuzione per l'inserimento degli URL maligni:", end_time - start_time)

start_time = time.time()
for url in url_test:
    if bf.check(url):
        print(url, "è un URL maligno")
end_time = time.time()
print("Tempo di esecuzione per la verifica degli URL:", end_time - start_time)
```

Versione parallela

```
pbf = ParallelBloomFilter(1000, 10, 16)

start_time = time.time()
pbf.initialize(url_maligni)
end_time = time.time()
print("Tempo di esecuzione per l'inserimento degli URL maligni:", end_time - start_time)

start_time = time.time()
print(pbf.parallelCheck(url_test))
end_time = time.time()
print("Tempo di esecuzione per la verifica degli URL:", end_time - start_time)
```

Per controllare il corretto funzionamento dell'algoritmo in entrambe le versioni sono state create 2 liste di url, url maligni con cui inizializzare il filtro e url test che contiene degli url maligni della prima lista (nelle prime 7 posizioni) e altri url generici, viene quindi creato un filtro inizializzato con la lista degli url maligni e viene poi stampato il check usando la seconda lista, il risultato ottenuto segnalerà quindi come maligni (true) solo i primi 7 elementi.

UNIVERSITÀ DEGLI STUDI FIRENZE

Il codice – Test delle prestazioni

```python
# test prestazioni

def generate_random_word(length):
    # Genera una stringa casuale di lettere minuscole di lunghezza length
    return ''.join(random.choices(string.ascii_lowercase, k=length))

element = 500 #numero di elementi di prova
testing=[]
testing2=[]

random.seed(30)
for i in range(element):
    testing.append(generate_random_word(10))

for i in range(element):
    testing2.append(generate_random_word(10))


pbf = ParallelBloomFilter(1000, 7, 16)

start_time1 = time.time()
pbf.initialize(testing)
end_time1 = time.time()
print("Tempo di esecuzione per l'inserimento degli URL maligni:", end_time1 - start_time1)

start_time2 = time.time()
pbf.parallelCheck(testing2)
end_time2 = time.time()
print("Tempo di esecuzione per la verifica degli URL:", end_time2 - start_time2)
print("tempo di esecuzione totale:", end_time2 - start_time1)
```

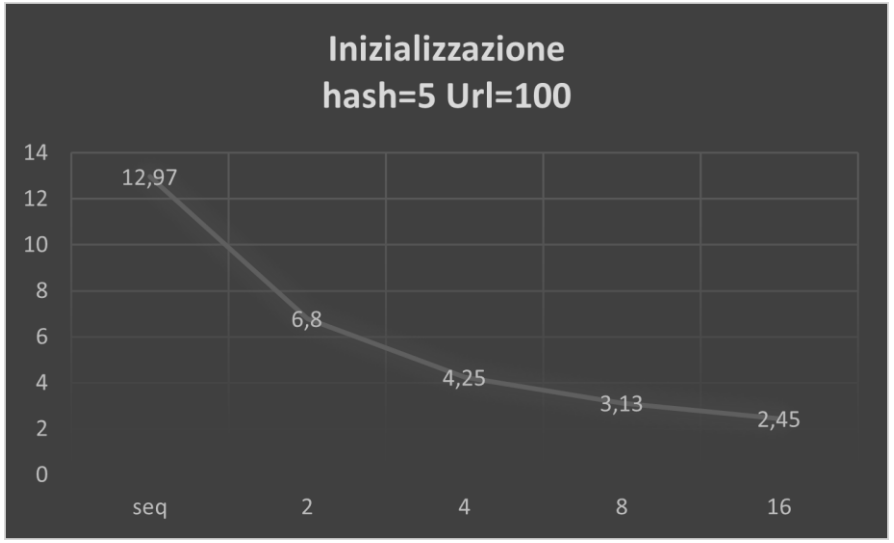
Data la natura del problema i parametri di confronto rilevanti sono il variare del numero delle funzioni hash e del numero di url.

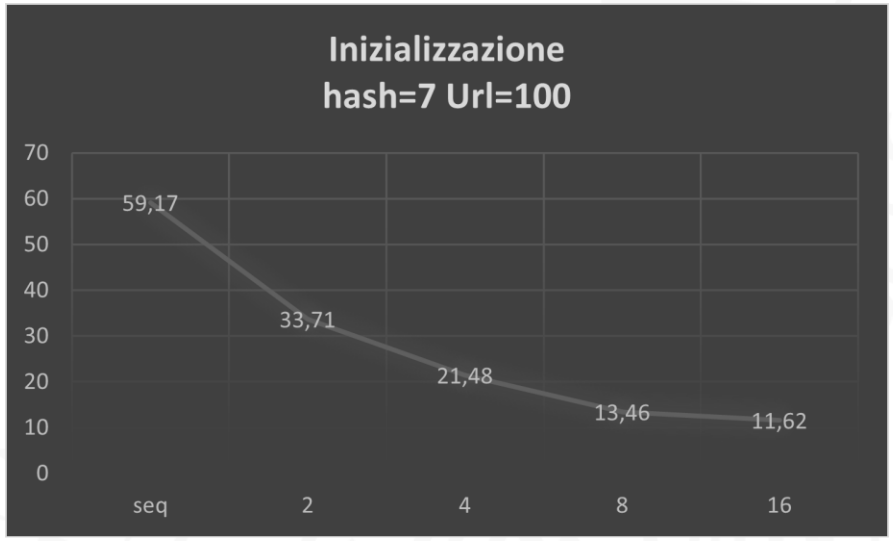
Per confrontare quest'ultimo cambiamento è stata definita una funzione **generate_random_word()** per generare casualmente una lista di stringhe, dato che viene usato sempre lo stesso seed le parole generate sono sempre le stesse.

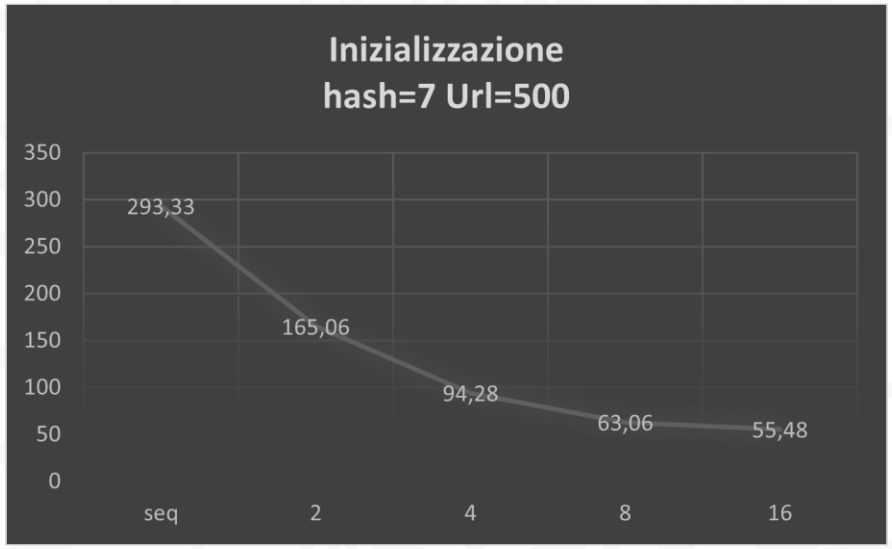
Notare che la dimensione del filtro influisce solo il numero di falsi positivi e non sulle prestazioni, viene quindi inizializzato a 1000.

Vengono presi i tempi di esecuzione sia delle 2 fasi distinte (inizializzazione e check) sia dell'esecuzione totale.

Analisi delle prestazioni - inizializzazione



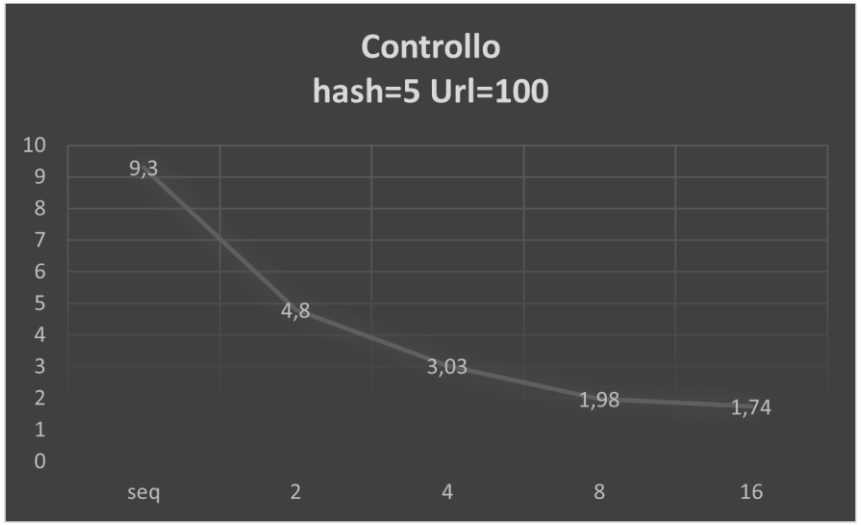


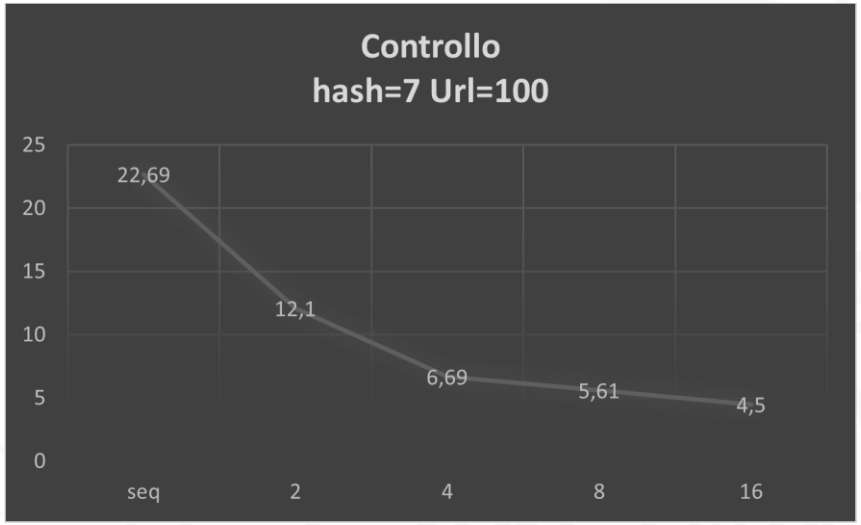


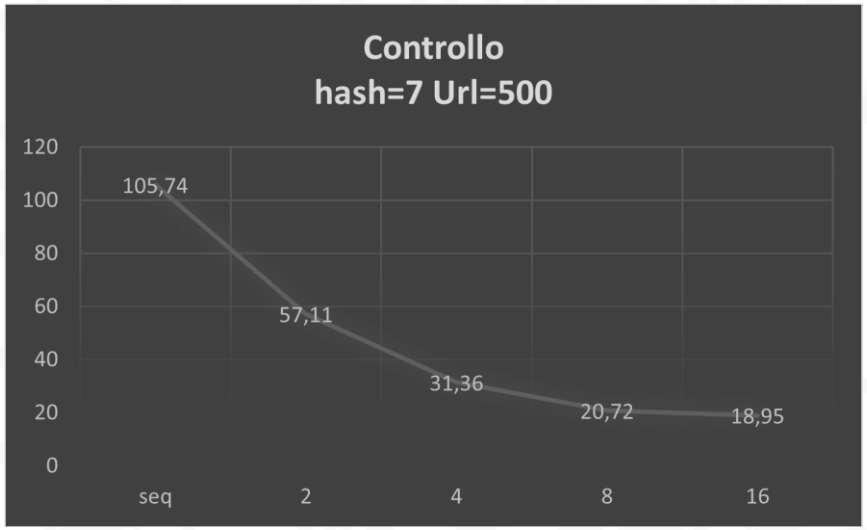
Qui vengono mostrate le diverse esecuzioni dell'inizializzazione al variare del numero di thread.

Lo speed-up medio ottenuto è di circa 5.

Analisi delle prestazioni - controllo





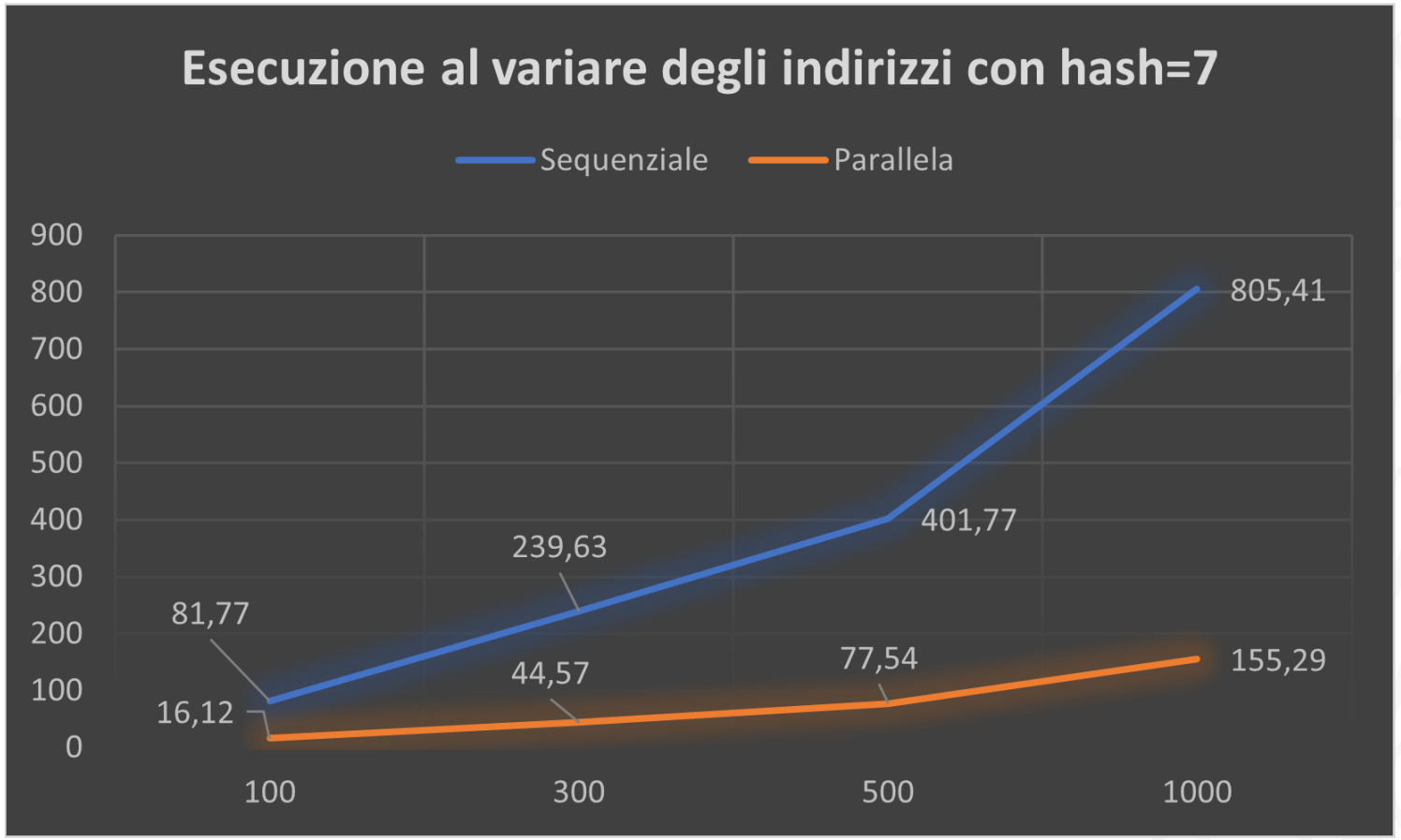


Qui vengono mostrate le diverse esecuzioni del controllo al variare del numero di thread.

Lo speed-up medio ottenuto è di circa 5.

UNIVERSITÀ DEGLI STUDI FIRENZE

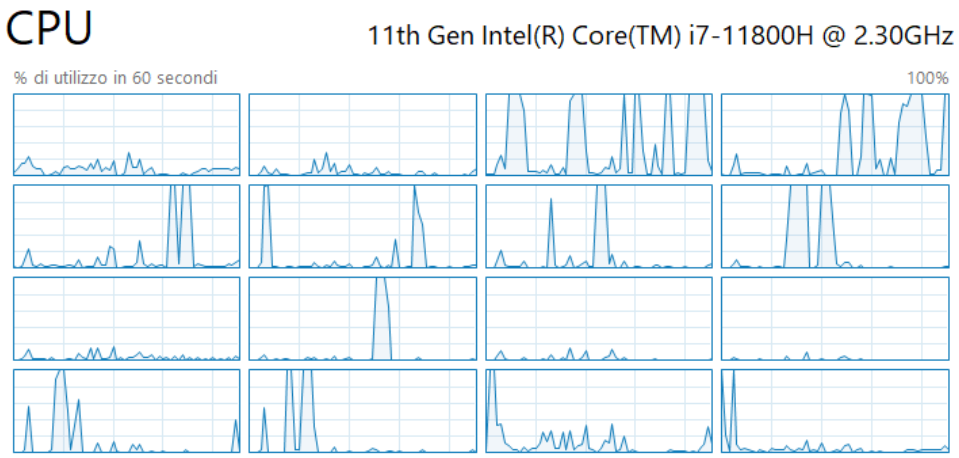
Analisi delle prestazioni



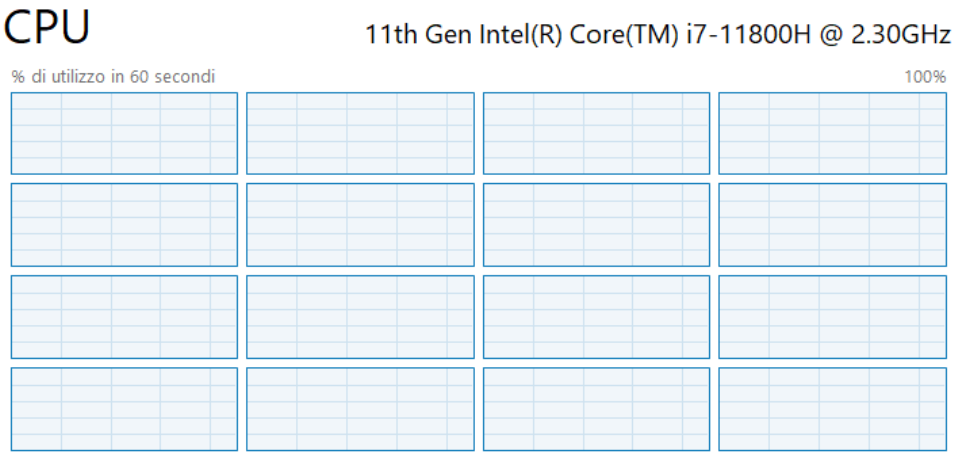
Qui è riportata l'esecuzione complessiva al variare degli indirizzi, dato che entrambi le fasi hanno uno speed up simile (circa 5) anche qui si può notare la stessa tendenza.

Conclusione

In conclusione l'implementazione parallela presenta un netto vantaggio in termini di tempo di esecuzione rispetto alla versione sequenziale, con una media di aumento della velocità di circa 5 volte.



CPU durante l'esecuzione sequenziale



CPU durante l'esecuzione parallela

UNIVERSITÀ DEGLI STUDI FIRENZE

Fine