Genome sequence research

Carmelo Santamaria (514926)
Federico Sciuto (517075)

Novembre 2023

1 Introduzione

Il genoma è l'insieme del patrimonio genetico che caratterizza ogni organismo vivente. Il genoma umano è costituito da 25 differenti molecole di DNA. Poiché la quantità di DNA nucleare è preponderante rispetto a quello mitocondriale, per genoma si intende spesso l'insieme delle molecole di DNA cromosomico, più correttamente chiamato genoma nucleare. Nell'uomo il genoma nucleare è costituito da circa 3.200 Mb (Mb = megabasi, o milioni di paia di basi), comprendenti circa 25.000 geni, localizzati sui cromosomi. Il genoma mitocondriale consiste di un DNA circolare a doppio filamento lungo 16,6 kb e contenente 37 geni, presente in molte copie nei mitocondri. Qualsiasi cellula di un organismo umano possiede lo stesso genoma (genoma equivalente): infatti è l'espressione differenziale dei geni, alla base dello sviluppo embrionale, che rende diversi i vari tipi cellulari.

2 Frameworks

2.1 MapReduce

MapReduce è un modello di programmazione che viene eseguito su Hadoop, un motore di analisi dati ampiamente utilizzato per i Big Data e scrive applicazioni che vengono eseguite in parallelo per elaborare grandi volumi di dati archiviati su cluster.

Il funzionamento di MapReduce può essere suddiviso in tre fasi, di cui 2 principali (Map e Reduce), con una quarta fase facoltativa (Combiner).

- **Mapper:** in questa prima fase, la logica condizionale filtra i dati attraverso tutti i nodi in coppie chiave-valore. La "chiave" si riferisce all'indirizzo

di offset per ogni record, mentre il "valore" contiene l'intero contenuto del record.

- **Shuffle:** durante la seconda fase, i valori di output dalla mappatura vengono ordinati e consolidati. I valori vengono raggruppati in base a chiavi simili e i quelli duplicati vengono scartati. Anche l'output della fase Shuffle è organizzato in coppie chiave-valore, ma questa volta i valori indicano un intervallo e non il contenuto di un record.
- **Reducer:** nella terza fase, l'output della fase Shuffle consolidato viene aggregato, con tutti i valori aggiunti alle chiavi corrispondenti. Tutto questo viene quindi combinato in una singola directory di output.
- **Combiner:** l'esecuzione di questa fase può ottimizzare le prestazioni dei processi MapReduce, accelerandone il flusso. Per effettuare questa operazione MapReduce recupera gli output della fase Mapper e li esamina a livello di nodo per individuare duplicati, che vengono combinati in una singola coppia chiave-valore, riducendo in questo modo il processo che la fase Shuffle deve completare.

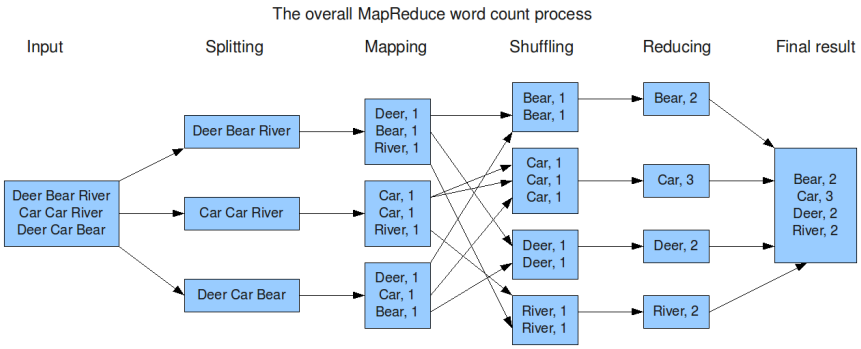


Figure 1: Schema di esempio del MapReduce

2.2 Ray

Ray è un framework open source versatile che offre una piattaforma unificata per la gestione e l'elaborazione di carichi di lavoro AI e Python su larga scala, con l'obiettivo principale di semplificare il processo di sviluppo e distribuzione di applicazioni di intelligenza artificiale. Fornisce agli sviluppatori gli strumenti necessari per sfruttare al meglio le risorse del sistema.

Il cuore di Ray è costituito dai cluster Ray, che consentono di distribuire e gestire in modo efficiente i carichi di lavoro. Un cluster Ray è composto da: un nodo principale (head) responsabile della coordinazione generale e dell'allocazione delle risorse, e un insieme di nodi di lavoro (worker) che eseguono i task specifici.

Gli sviluppatori possono definire i loro carichi di lavoro come task paralleli o processi, che vengono quindi distribuiti tra i nodi di lavoro all'interno del cluster Ray. Questo modello parallelo consente di sfruttare al meglio le risorse disponibili, accelerando notevolmente l'elaborazione di compiti intensivi, come l'addestramento di modelli di machine learning o la simulazione nell'apprendimento per rinforzo.

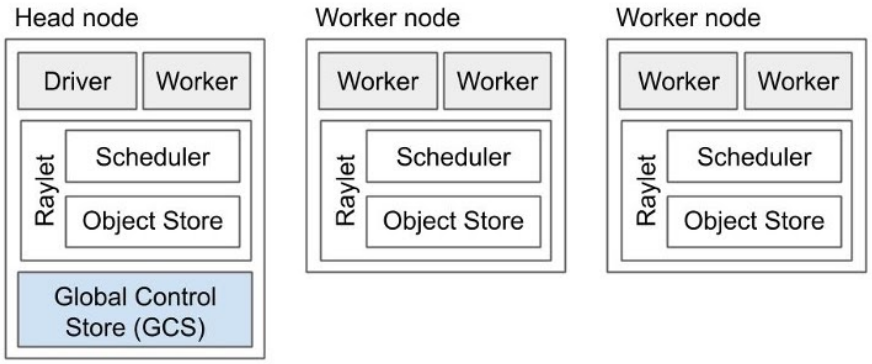


Figure 2: Schema di esempio del funzionamento di un cluster Ray

3 Codice

Il codice è stato scritto in Python, un linguaggio di programmazione noto per la sua semplicità e vantaggioso per la sua ampia libreria standard. L'ambiente di sviluppo in cui il codice è stato creato e testato è Ubuntu 22.04 LTS, un sistema operativo stabile e leggero, che offre un ottimo supporto per Python e la libreria Ray. Per l'implementazione di alcune funzionalità specifiche, sono state utilizzate le seguenti librerie e moduli:

```
1 from Bio import SeqIO
      # Modulo della libreria Biopython, utilizzato per lavorare con
      sequenze biologiche (sequenze di DNA, RNA e proteine). SeqIO
      permette la lettura e la scrittura di file contenenti queste
      sequenze
2 from time import perf_counter
      # Funzione del modulo time che restituisce un tempo di clock per
      misurare il tempo trascorso tra due punti in un programma
3 from functools import reduce
      # Funzione del modulo functools che consente di applicare una
      funzione cumulativa a elementi di un iterabile, riducendo la
      sequenza a un singolo valore. Usato per eseguire operazioni di
      aggregazione su una sequenza di dati.
4 import multiprocessing
      # Libreria che fornisce supporto per la programmazione concorrente,
       consentendo di eseguire processi multipli contemporaneamente per
      sfruttare i multi-core delle CPU
5 from multiprocessing import cpu_count
      # Funzione che restituisce il numero di core della CPU disponibili
      sul sistema in uso
6 import ray
      # Libreria per la programmazione parallela e distribuita in Python
```

Per i test del codice multithread è stata utilizzata una macchina con 12 thread, che è stata il nodo "head" nel cluster utilizzato con la versione Ray, affiancato da altri 9 nodi "worker" con 4 thread ciascuno; per un totale di 10 nodi e 48 thread.

3.1 Monothread

Nel codice Monothread il lavoro è stato suddiviso tra le funzioni:

1. **"Mapping"**, la funzione di mappatura, che
   - Prende in ingresso:
     - sequenza (una sequenza genetica);
     - stringa_target (la stringa da cercare in sequenza);
   - Calcola la "*lunghezza_target*", cioè la lunghezza della "*stringa_target*".
   - Crea una lista di sottosequenze attraverso una list comprehension contenente tutte le sottosequenze di sequenza che hanno la stessa lunghezza di "*stringa_target*".

- Restituisce la lista *"sottosequenze"*.

2. **"Reducing"**, la funzione di riduzione, che
   - Prende in ingresso:
     - sottosequenze (la lista di sottosequenze create dalla funzione *mapping*);
     - stringa_target;
   - Calcola il *"conteggio"*, ovvero quante volte *"stringa_target"* appare in *"sottosequenze"* utilizzando il metodo ***count()***.
   - Restituisce il *"conteggio"*.
3. **"Conteggio_finale"**, che
   - Prende in ingresso i conteggi intermedi:
     - count1;
     - count2;
   - Restituisce la loro somma.
4. **"MapReduce"**, la funzione principale che esegue l'analisi MapReduce attraverso le funzioni definite precedentemente.
   Essa:
   - Registra il tempo di inizio dell'esecuzione
   - Utilizza ***map()*** per applicare la funzione ***lambda*** a ciascuna sequenza in *"data"*. La funzione ***lambda*** prende una sequenza, esegue la funzione ***reducing*** sulla sequenza utilizzando la *"stringa_target"* e restituisce il *"conteggio"*.
   - Utilizza ***reduce()*** per sommare i conteggi intermedi e calcolare il *"conteggio_totale"*.
   - Registra il tempo di fine esecuzione.
   - Stampa il risultato, indicando quante volte la *"stringa_target"* è presente nel file e quanto tempo è stato necessario per l'analisi.

```python
from Bio import SeqIO
from time import perf_counter
from functools import reduce

# Funzione di mapping che conta le occorrenze della stringa target in una sottosequenza
def mapping(sequenza, stringa_target):
    lunghezza_target = len(stringa_target)
    sottosequenze = [sequenza[i:i + lunghezza_target] for i in range(len(sequenza) - lunghezza_target + 1)]
    return sottosequenze

```

```python
# Funzione di reduce che somma i conteggi intermedi per sequenza
def reducing(sottosequenze, stringa_target):
    conteggio = sottosequenze.count(stringa_target)
    return conteggio

def conteggio_finale(count1, count2):
    return count1 + count2

# Funzione principale che esegue le funzioni di mapping, reducing e conteggio_finale
def MapReduce(data, stringa_target):
    start_time = perf_counter()

    # Utilizza map() per ottenere i conteggi intermedi per sequenza
    conteggi_intermedi = map(lambda sequenza: reducing(mapping(sequenza.seq, stringa_target), stringa_target), data)

    # Utilizza reduce() per sommare i conteggi intermedi
    conteggio_totale = reduce(conteggio_finale, conteggi_intermedi)

    end_time = perf_counter()

    print(f"\nLa stringa '{stringa_target}'    presente {conteggio_totale} volte nel file.")
    print(f"Tempo di esecuzione: {end_time - start_time:0.3f} secondi.\n")

def main():
    global filename

    # Scelta del file da analizzare
    genome_files = input("\nSeleziona il file da analizzare:\n1 -> GRCh37_42MB_protein.faa\n2 -> GRCh38_105MB_protein.faa\n3 -> GRCh37_302MB_rna.fna\n4 -> GRCh38_742MB_rna.fna\n")

    file_mapping = {
        "1": "GRCh37_42MB_protein.faa",
        "2": "GRCh38_105MB_protein.faa",
        "3": "GRCh37_302MB_rna.fna",
        "4": "GRCh38_742MB_rna.fna"
    }

    filename = file_mapping.get(genome_files)

    if not filename:
        print("\nOpzione non valida\n")
    else:
        stringa_target = input("\nStringa da ricercare: ")

        # Fase di Splitting (Splitting Phase)
        sequenze = list(SeqIO.parse(filename, 'fasta'))

        # Chiamata alla funzione MapReduce
        MapReduce(sequenze, stringa_target)

if __name__ == "__main__":
    main()
```

3.2 Multithread

L'implementazione del codice Multithread è molto simile a quella vista nel Monothread, con la differenza che vengono utilizzati la libreria ***"Multiprocessing"*** e il modulo ***"cpu_count"*** per supportare il parallelismo; è stata, inoltre, modificata la funzione ***"MapReduce"*** e inserita una nuova funzione:

1. **"Process_chunk"**, definita per suddividere il lavoro fra i vari core, che
   - Prende in ingresso:
     - chunk di dati (un sottoinsieme delle sequenze genetiche);
     - stringa_target;
   - Esegue la fase di ***mapping*** e ***reducing*** sulle sequenze all'interno del *"chunk"*
   - Restituisce il *"conteggio"*

Le modifiche nella funzione **"MapReduce"** consistono in:

- Una divisione dei dati in *"chunk"* per la parallelizzazione, calcolando il *"chunk_size"* in base al numero di processi specificato.
- Uso di un pool di processi (*multiprocessing.Pool*) per distribuire il lavoro sui *"chunk"* ai processi paralleli utilizzando *"pool.starmap"*.
- La somma dei conteggi intermedi prodotti dai processi paralleli per ottenere il *"conteggio totale"*.
- La possibilità di scegliere il numero di processi da utilizzare per il parallelismo utilizzando *"multiprocessing.cpu_count()"* per ottenere il numero massimo di core disponibili sulla macchina.

```
from Bio import SeqIO
from time import perf_counter
from functools import reduce
import multiprocessing
from multiprocessing import cpu_count

# Funzione di mapping che conta le occorrenze della stringa target in una sottosequenza
def mapping(sequenza, stringa_target):
    lunghezza_target = len(stringa_target)
    sottosequenze = [sequenza[i:i + lunghezza_target] for i in range(len(sequenza) - lunghezza_target + 1)]
    return sottosequenze

# Funzione di reduce che somma i conteggi intermedi per sequenza
def reducing(sottosequenze, stringa_target):
    conteggio = sottosequenze.count(stringa_target)
    return conteggio

```

```python
def conteggio_finale(count1, count2):
    return count1 + count2

# Funzione che elabora una porzione dei dati e restituisce il conteggio
def process_chunk(chunk, stringa_target):
    conteggi_intermedi = map(lambda sequenza: reducing(mapping(sequenza.seq, stringa_target), stringa_target), chunk)
    conteggio = reduce(conteggio_finale, conteggi_intermedi)
    return conteggio

# Funzione principale che esegue le funzioni di mapping, reducing, conteggio_finale e process_chunk
def MapReduce(data, stringa_target, num_process):
    start_time = perf_counter()

    # Dividi i dati in porzioni per la parallelizzazione
    chunk_size = len(data) // num_process
    chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

    # Creazione di un pool di processi
    with multiprocessing.Pool(processes=num_process) as pool:
        conteggi_intermedi = pool.starmap(process_chunk, [(chunk, stringa_target) for chunk in chunks])

    # Utilizza reduce() per sommare i conteggi intermedi
    conteggio_totale = reduce(conteggio_finale, conteggi_intermedi)

    end_time = perf_counter()

    print(f"\nLa stringa '{stringa_target}' appare {conteggio_totale} volte nel file.")
    print(f"Tempo di esecuzione: {end_time - start_time:0.3f} secondi.\n")

def main():
    global filename

    # Scelta del file da analizzare
    genome_files = input("\nSeleziona il file da analizzare:\n1 -> GRCh37_42MB_protein.faa\n2 -> GRCh38_105MB_protein.faa\n3 -> GRCh37_302MB_rna.fna\n4 -> GRCh38_742MB_rna.fna\n")

    file_mapping = {
        "1": "GRCh37_42MB_protein.faa",
        "2": "GRCh38_105MB_protein.faa",
        "3": "GRCh37_302MB_rna.fna",
        "4": "GRCh38_742MB_rna.fna"
    }

    filename = file_mapping.get(genome_files)

    if not filename:
        print("\nOpzione non valida\n")
    else:
        stringa_target = input("\nStringa da ricercare: ")
        num_process_max = multiprocessing.cpu_count()  # Numero di core
```

```
    disponibili
        num_process = int(input(f"\nNumero di processi da utilizzare (max = {num_process_max}): "))

        if num_process < 1 or num_process > num_process_max:
            print(f"\nNumero di processi inserito non valido (deve essere compreso tra 1 e {num_process_max})\n")
        else:
            # Fase di Splitting (Splitting Phase)
            sequenze = list(SeqIO.parse(filename, 'fasta'))

            # Chiamata alla funzione MapReduce con parallelismo
            MapReduce(sequenze, stringa_target, num_process)

if __name__ == "__main__":
    main()
```

3.3 Ray

Anche nel codice Ray l'implementazione è molto simile a quelle viste in precedenza; le differenze, oltre all'utilizzo della libreria *"ray"*, sono:

- L'esecuzione della libreria ***"ray.init()"*** per inizializzare il framework **Ray**.
- L'annotazione della funzione ***process_chunk*** con ***"@ray.remote"***, che permette a questa funzione di essere eseguita in modo parallelo da **Ray**.
- Anche in questo caso, così come per il Multithreading si ha la possibilità di scegliere il numero di processi da utilizzare, ma per ottenere il numero massimo di core disponibili sul cluster stavolta sono stati utilizzati:
  - cluster_resources = ray.cluster_resources()
  - num_process_max = int(cluster_resources.get("CPU", 1))

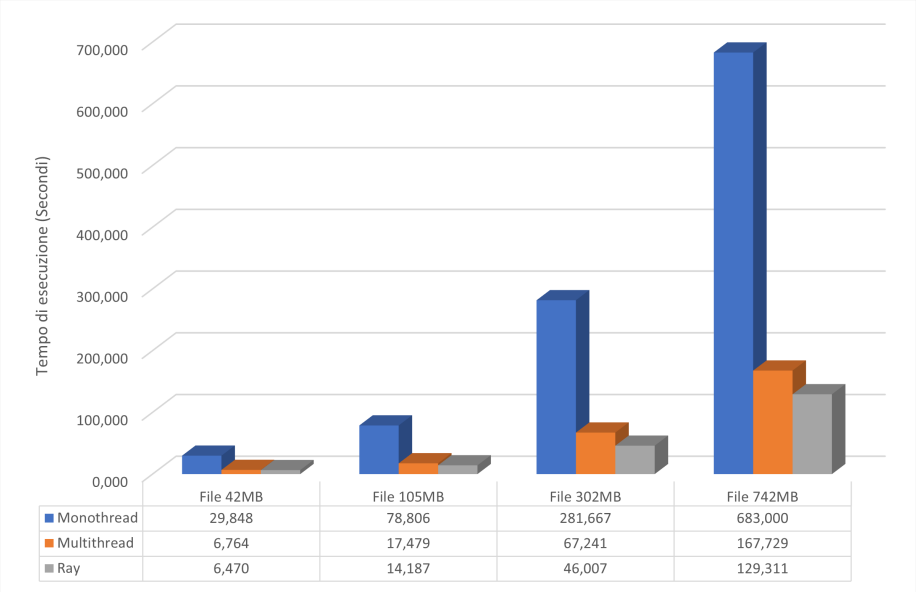
```
from Bio import SeqIO
from time import perf_counter
from functools import reduce
import multiprocessing
from multiprocessing import cpu_count
import ray

# Funzione di mapping che conta le occorrenze della stringa target in una sottosequenza
def mapping(sequenza, stringa_target):
    lunghezza_target = len(stringa_target)
    sottosequenze = [sequenza[i:i + lunghezza_target] for i in range(len(sequenza) - lunghezza_target + 1)]
    return sottosequenze

# Funzione di reduce che somma i conteggi intermedi per sequenza
def reducing(sottosequenze, stringa_target):
```

```
    conteggio = sottosequenze.count(stringa_target)
    return conteggio

def conteggio_finale(count1, count2):
    return count1 + count2

# Funzione che elabora una porzione dei dati e restituisce il conteggio
@ray.remote  # Indica che questa funzione    gestita da Ray
def process_chunk(chunk, stringa_target):
    conteggi_intermedi = map(lambda sequenza: reducing(mapping(sequenza
    .seq, stringa_target), stringa_target), chunk)
    conteggio = reduce(conteggio_finale, conteggi_intermedi)
    return conteggio

# Funzione principale che esegue le funzioni di mapping, reducing,
    conteggio_finale e process_chunk
def MapReduce(data, stringa_target, num_process):
    start_time = perf_counter()

    # Dividi i dati in porzioni per la parallelizzazione
    chunk_size = len(data) // num_process
    chunks = [data[i:i+chunk_size] for i in range(0, len(data),
    chunk_size)]

    # Creazione di un elenco di riferimenti a risultati futuri
    future_results = [process_chunk.remote(chunk, stringa_target) for
    chunk in chunks]

    # Attendere i risultati dai processi paralleli
    conteggi_intermedi = ray.get(future_results)

    # Utilizza reduce() per sommare i conteggi intermedi
    conteggio_totale = reduce(conteggio_finale, conteggi_intermedi)

    end_time = perf_counter()

    print(f"\nLa stringa '{stringa_target}' appare {conteggio_totale}
    volte nel file.")
    print(f"Tempo di esecuzione: {end_time - start_time:0.3f} secondi.\
    n")

def main():
    ray.init()  # Inizializza Ray

    global filename

    # Scelta del file da analizzare
    genome_files = input("\nSeleziona il file da analizzare:\n1 ->
    GRCh37_42MB_protein.faa\n2 -> GRCh38_105MB_protein.faa\n3 ->
    GRCh37_302MB_rna.fna\n4 -> GRCh38_742MB_rna.fna\n")

    file_mapping = {
        "1": "GRCh37_42MB_protein.faa",
        "2": "GRCh38_105MB_protein.faa",
        "3": "GRCh37_302MB_rna.fna",
        "4": "GRCh38_742MB_rna.fna"
    }
```

```
65
66      filename = file_mapping.get(genome_files)
67
68      if not filename:
69          print("\nOpzione non valida\n")
70      else:
71          stringa_target = input("\nStringa da ricercare: ")
72
73          # Ottiene il numero totale di core disponibili sul cluster
74          cluster_resources = ray.cluster_resources()
75          num_process_max = int(cluster_resources.get("CPU", 1)) # Numero
        di core disponibili
76          num_process = int(input(f"\nNumero di processi da utilizzare (
    max = {num_process_max}): "))
77
78          if num_process < 1 or num_process > num_process_max:
79            print(f"\nNumero di processi inserito non valido (deve essere
        compreso tra 1 e {num_process_max})\n")
80          else:
81            # Fase di Splitting (Splitting Phase)
82            sequenze = list(SeqIO.parse(filename, 'fasta'))
83
84            # Chiamata alla funzione MapReduce con parallelismo
85            MapReduce(sequenze, stringa_target, num_process)
86
87  if __name__ == "__main__":
88      main()
```

4 Risultati

Di seguito sono riportate le medie dei tempi d'esecuzione delle prove effettuate. Rappresentate sia con un grafico, che con una tabella.



	File 42MB	File 105MB	File 302MB	File 742MB
Monothread	29,848	78,806	281,667	683,000
Multithread	6,764	17,479	67,241	167,729
Ray	6,470	14,187	46,007	129,311

Figure 3: Grafico dei tempi d'esecuzione, con tabella contenente i valori espressi in secondi

5 Considerazioni

I test sulla ricerca di sottostringhe specifiche all'interno di sequenze genomiche hanno fornito risultati significativi. Attraverso l'analisi di diverse architetture di calcolo, tra cui una versione monothread, una multithread e un'implementazione con Ray, è possibile osservare le differenze prestazionali delle diverse strategie di elaborazione dati in questa attività di bioinformatica.

La versione monothread si è dimostrata la più lenta tra quelle analizzate, lentezza che cresce in maniera proporzionale alla grandezza del file utilizzato. Questo è dovuto al limite di utilizzo di un solo thread, che ha comportato un'esecuzione sequenziale, rendendo difficile l'analisi di sequenze genomiche di grandi dimensioni.

Il multithread è emerso come un significativo passo avanti, consentendo l'elaborazione parallela su più core. Si può osservare un notevole miglioramento delle prestazioni rispetto alla versione monothread con l'aumentare del numero di core utilizzati.

Nonostante siano state riscontrate inefficienze nell'implementazione di Ray, in quanto non tutti i core a disposizione sono stati utilizzati contemporaneamente al 100% (inefficienza che potrebbe essere attribuita a diversi fattori, tra cui la distribuzione dei dati, l'efficienza dei nodi utilizzati e la possibilità che gli stessi fossero destinati a compiti esterni al test), un ulteriore passo in avanti è stato compiuto con l'implementazione di Ray. L'uso di questa architettura ci ha permesso di sfruttare più macchine e, di conseguenza, più core contemporaneamente rispetto a quanto consentisse di fare il multithreading. L'efficienza e la scalabilità offerte da Ray sono risultate utili per affrontare grandi volumi di dati genetici.

Questo progetto ha fornito informazioni sull'ottimizzazione dell'analisi delle sequenze genomiche, dimostrando come l'adozione delle giuste strategie di parallelismo possa portare a risultati notevolmente più veloci e precisi.