Valutazione effetti della parallelizzazione sull'algoritmo K-Means

Jacopo Sabatino
jacopo.sabatino@edu.unifi.it

Abstract

In questo articolo sono illustrati gli effetti della parallelizzazione sull'algoritmo K-Means. L'algoritmo K-Means implementato è nella sua forma base e parte con un set di centroidi predefinito. È stato valutato lo speedup per diversi valori di quantità di punti, numero di cluster e numero di thread.

1. Introduzione

Per questo elaborato è stata implementata una versione sequenziale e una versione parallela dell'algoritmo K-Means con l'obiettivo di valutarne lo speedup.

La logica delle due versioni è uguale, anche se per quella parallela è stato necessario apportare alcune modifiche che ne permettono la corretta parallelizzazione. Il codice è stato parallelizzato utilizzando le direttive **OpenMP**.

Prima di valutare lo speedup, è stata effettuata un'analisi per sapere quale modello di architettura fra AoS e SoA si adattasse meglio al problema.

2. Implementazione

L'algoritmo K-Means implementato è nella sua versione base. I punti su cui è eseguito l'algoritmo sono tridimensionali. L'esecuzione parte da un numero K di centroidi predefiniti e la condizione di interruzione imposta si basa su un numero finito di esecuzioni del ciclo. Le ultime due condizioni permettono di avere riproducibilità dei test.

La versione sequenziale ha due implementazioni, una i cui punti sono memorizzati come un array di strutture (AoS) e una i cui punti sono memorizzati come una struttura di array (SoA). La versione parallela invece è stata implementata solo con il modello SoA.

2.1. Parallelizzazione

La prima scelta di parallelizzazione presa è stata dove generare i thread. I thread sono generati all'esterno del ciclo che si occupa di eseguire le N iterazioni del K-Means. Ciò permette di riutilizzare gli stessi thread fra le varie iterazioni e avere un solo fork-join.

Durante la generazione dei thread sono dichiarate shared le seguenti variabili:

- *dataPoints*: struttura SoA che memorizza tutti i punti. Non è a rischio data race perché utilizzata dai vari thread solo in lettura;
- *centroids*: struttura SoA che memorizza i centroidi. É utilizzata in lettura per il calcolo delle distanze e in scrittura per l'aggiornamento alla fine di ognuna delle N iterazioni;
- *totalClustersSize*: vettore di interi che rappresentano il numero di elementi appartenenti ad ogni cluster. É aggiornato ad ogni iterazione del ciclo.

Ad ogni iterazione sono dichiarate la struttura SoA *newCentroids* e il vettore di interi *clustersSize*, in modo che ogni thread ne abbia una versione locale. Tali strutture sono poi utilizzate per aggiornare le strutture shared *centroids* e *totalClustersSize*.

Dichiarate le due strutture, l'algoritmo entra nella sua regione parallela, un ciclo **for** che serve ad assegnare ogni punto ad un cluster. Per aumentare il lavoro da effettuare nella regione parallela, è stato aggiunto al suo interno l'aggiornamento delle strutture locali *newCentroids* e *clustersSize*. Lo scheduling scelto per il ciclo è di tipo statico, cioè ad ogni thread è assegnata a priori l'esecuzione di un numero fisso di iterazioni del ciclo. La dimensione dei blocchi di iterazioni da assegnare è definita implicitamente e corrisponde al rapporto fra numero di iterazioni totali e numero di thread. La barriera implicita della direttiva **omp for** è stata mantenuta in quanto necessaria per far sì che tutti i punti siano assegnati prima di aggiornare la struttura shared *centroids*. Se la barriera fosse rimossa, la struttura potrebbe essere aggiornata quando alcuni thread necessitano ancora di effettuare la lettura dei suoi valori.

Prima di aggiornare *centroids*, è necessario che tutte le coordinate dei centroidi contenuti siano poste a zero. Ciò è effettuato da un unico thread attraverso la direttiva **omp single**. Tale direttiva ha una barriera implicita che permette l'aggiornamento dei centroidi solo una volta che questi hanno effettivamente tutte le variabili a zero.

L'aggiornamento dei centroidi è eseguito accumulando per ogni cluster le coordinate del centroide corrispondente

trovato da ogni thread. Questa operazione è a rischio data race e quindi è protetta da una direttiva **omp atomic** per ogni singola coordinata del centroide. Inoltre, per ogni cluster è stato accumulato in *totalClustersSize* il numero di punti a lui assegnati da ogni thread. Tale operazione è protetta sempre da un **omp atomic**.

Accumulate le singole coordinate per ogni centroide, bisogna calcolarne il vero valore dividendole per il numero di punti assegnati al cluster di riferimento. Per fare ciò, bisogna assicurarsi che l'operazione di accumulo sia effettivamente finita e quindi è stata inserita una **omp barrier**.

Il calcolo delle reali coordinate è eseguito da un solo thread grazie alla direttiva **omp single**. La barriera implicita della direttiva fa sì che gli altri thread attendano l'effettivo aggiornamento dei centroidi prima di iniziare la nuova iterazione dell'algoritmo K-Means.

3. Test e confronti

Per valutare lo speedup fra versione sequenziale e versione parallela sono stati eseguiti vari test. La logica seguita per confrontare le due versioni è variare uno dei parametri che caratterizzano il problema e mantenere inalterati i restanti.
I confronti eseguiti sono:

- al variare del numero di punti;
- al variare del numero di cluster;
- al variare del numero dei thread.

Prima di eseguire questi test sono state paragonate due implementazioni sequenziali del problema per capire se fosse più consono memorizzare i punti come array di strutture o come struttura di array.

Ogni configurazione testata è stata eseguita dieci volte cosicché il valore utilizzato nel confronto sia più vicino al caso medio.

Il vincolo di arresto dei test è uguale a dieci iterazioni. Questo valore è adeguato per ottenere una corretta suddivisione in cluster per tutte le configurazioni testate.

Tutti i dataset utilizzati nei test sono composti da quattro blob con distribuzione gaussiana e centri diversi. Per questo motivo il numero K di cluster è sempre posto a quattro ad eccezione dei casi in cui si testa l'algoritmo al suo variare.

3.1. Confronto AoS-SoA

In questa implementazione del K-Means la lettura in memoria serve ad ottenere le coordinate di un punto in modo che se ne possa calcolare la distanza rispetto ai centroidi. Durante il calcolo della distanza, tutte le coordinate del punto sono utilizzate, e ciò è un punto a vantaggio per la scelta del modello AoS. È stato comunque scelto di verificare le prestazioni dell'algoritmo rispetto al modello SoA per scegliere quale architettura sia più valida.

Il paragone fra le due versioni è stato effettuato fissando il numero K di cluster a quattro e variando il numero di punti.

Come visibile dalla **Figura 1**, AoS ha prestazioni migliori rispetto a SoA per quantità di dati contenute. Per grandi quantità di dati invece AoS risulta meno efficace. Ciò è probabilmente dovuto all'elevato numero di accessi in memoria effettuati dal modello AoS al crescere dei punti. SoA invece, vettorizzando le coordinate dei punti, effettua un minor numero di accessi in memoria.

Visto che solitamente gli algoritmi di clustering sono utilizzati su grandi quantità di dati, è stato scelto di utilizzare il modello SoA per l'implementazione della versione parallela. Quindi per i confronti che vedremo nei prossimi paragrafi, la versione sequenziale utilizzata nei confronti è quella con il modello SoA.

3.2. Confronto al variare del numero di punti

Il confronto fra versione sequenziale e parallela è stato eseguito fissando a quattro sia il numero K di cluster che il numero di thread. Il guadagno in prestazione fra le due versioni è valutato utilizzando lo speedup, cioè il rapporto fra tempo di esecuzione dell'algoritmo sequenziale e il corrispondente parallelo. Il valore ideale dello speedup è pari al numero di thread utilizzati per parallelizzare, quindi in questo caso quattro.

Come si può vedere dalla **Figura 2**, lo speedup cresce all'aumentare del numero di punti. Il grafico tende asintoticamente al valore ideale, quindi per un elevato numero di punti lo speedup è lineare. Ciò ha senso perché dare un alto carico di lavoro ad ogni thread motiva la partizione del lavoro totale e ammortizza il costo di gestione dei thread stessi. Se invece il carico di lavoro assegnato ad ogni thread fosse piccolo, il guadagno in prestazioni non sarebbe evidente, perché il lavoro totale è già ben gestito da un numero inferiore di thread. Quindi, per un numero contenuto di punti, il lavoro totale è ben gestito anche dalla versione sequenziale, ed è per questo che lo speedup è più basso. Questo concetto si nota bene dal confronto al variare del numero dei thread.

3.3. Confronto al variare del numero di cluster

Per questo confronto è stato variato il numero K di cluster e fissato il numero di punti a quattro milioni e il numero di thread a quattro. È importante ricordare che i dati generati per i test provengono da quattro blob con centri diversi, per cui al variare del numero di cluster ci sarà un numero di gruppi che può essere diverso da quello effettivo.

Dal grafico in **Figura 3** si può notare che all'aumentare del numero di cluster aumenta lo speedup. Questo perché,

aumentando il numero di cluster, aumenta il numero di distanze da calcolare per assegnare un punto ad un cluster. Quindi è stato incrementato il carico di lavoro assegnato ad ogni thread e di conseguenza l'efficienza della parallelizzazione.

A differenza del confronto al variare dei punti, in questo caso gli speedup calcolati hanno valori più alti. Ciò è causato proprio dal valore scelto per il numero di punti, che già di per sé assegna un buon carico di lavoro ad ogni thread.

3.4. Confronto al variare del numero di thread

In questo caso il problema è fissato e ciò che varia è il grado di parallelizzazione, cioè il numero di thread. Il numero di cluster scelto è pari a quattro, mentre i numeri di punti scelti sono 400k, 4M e 40M. Così facendo sono stati fissati tre problemi distinti. Il motivo di questa scelta è verificare se quantità di dati diverse rispondono in maniera differente al numero di thread.

Osservando **Figura 4** si nota che al crescere del numero di thread aumenta lo speedup, ma non in modo lineare. Quindi, continuare ad aumentare il numero di thread non è utile ai fini di un miglioramento delle prestazioni perché il guadagno in speedup sarà sempre minore fino ad un possibile peggioramento delle prestazioni. Ciò è causato dagli alti costi di gestione dei thread e dai ridotti carichi di lavoro assegnati. Per rendere ciò più chiaro, è stata calcolata l'efficienza, cioè il rapporto fra speedup e numero di thread utilizzati. L'efficienza rappresenta quanto i thread sono ben utilizzati rispetto allo sforzo fatto per gestire la loro sincronizzazione e comunicazione.

N° punti	Efficienza			
	2 thread	4 thread	8 thread	16 thread
400k	86,4%	74,4%	44,1%	25,8%
4M	90,1%	83,6%	58,4%	38,4%
40M	88,5%	85,3%	52,0%	37,2%

Tabella 1: Efficienza in percentuale al variare del numero di thread.

Guardando la **Tabella 1** si nota che per questo algoritmo un numero di thread pari a due o quattro permette di avere dei thread ben utilizzati e quindi un buono sfruttamento delle risorse.

La tabella ci permette di affermare che variare il numero di dati su cui lavora l'algoritmo non influisce più di tanto su come questo risponde a diversi gradi di parallelizzazione. Ciò è visibile anche dalla **Figura 4**, in cui l'andamento delle tre curve è simile.

4. Conclusioni

Considerando i test effettuati sull'algoritmo K-Means implementato, si può concludere che una sua parallelizzazione comporta effettivi benefici a livello di prestazioni. Per massimizzare questi benefici è importante assegnare ad ogni thread un carico di lavoro adeguato, cosicché siano ammortizzati i costi di sincronizzazione e comunicazione. É possibile assegnare un carico di lavoro maggiore ad ogni thread aumentando il numero di punti su cui l'algoritmo esegue e/o aumentando il numero K di cluster da identificare.

Per quanto invece riguarda il grado di parallelizzazione, è stato dimostrato che non è consigliato utilizzare una grande quantità di thread perché ciò comporta una perdita di efficienza. A fronte dei dati ottenuti, per questa specifica implementazione di K-Means si consiglia l'utilizzo di quattro thread perché risulta essere un buon compromesso fra il grado di parallelizzazione e l'efficienza dei thread.

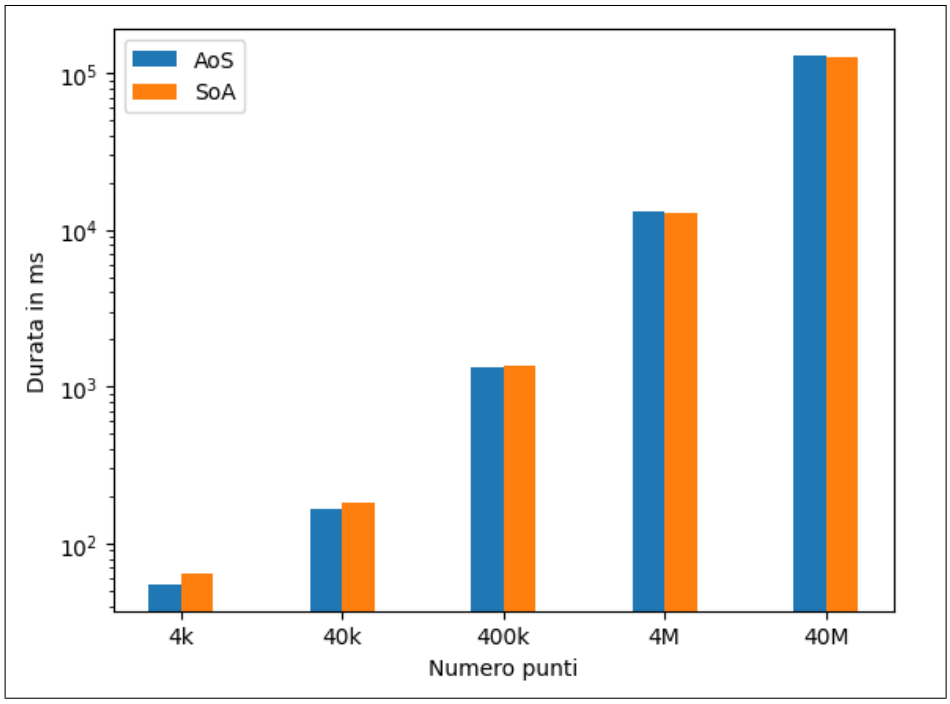

Figura 1: Confronto AoS-SoA al variare del numero di punti e con numero K di cluster fissato a quattro.

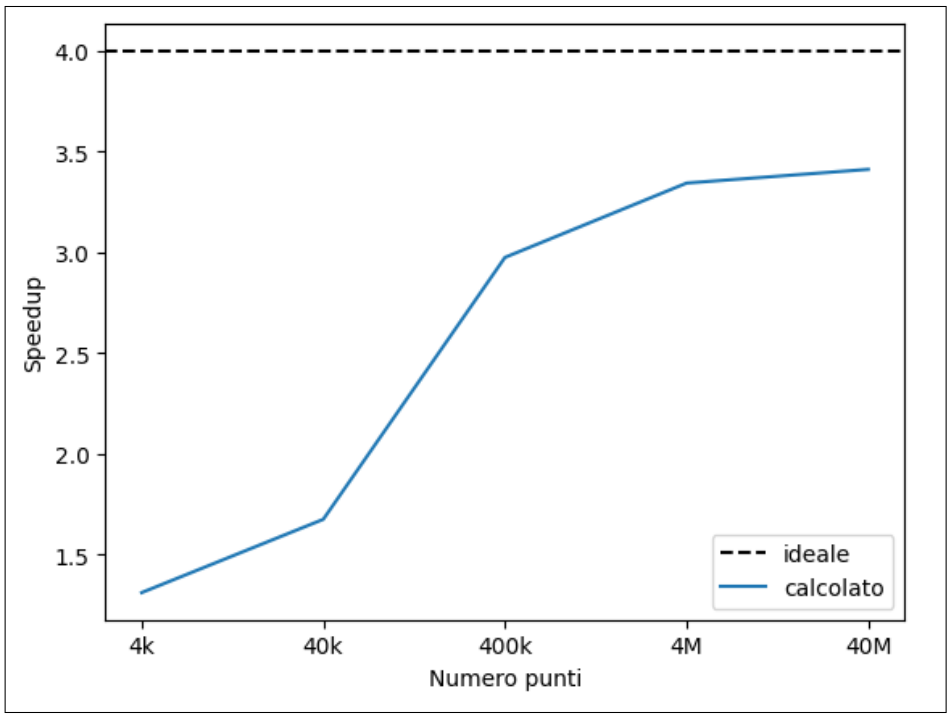

Figura 2: Calcolo speedup al variare del numero di punti con numero K di cluster e numero di thread pari a quattro.

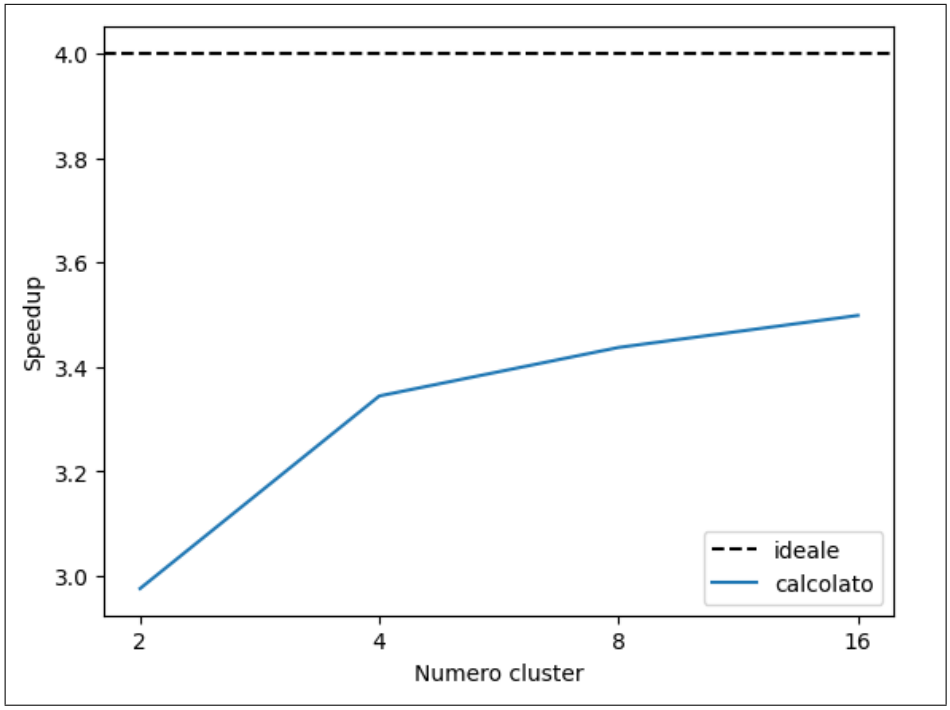

Figura 3: Calcolo speedup al variare del numero K di cluster con numero di punti uguale a 4M e numero thread pari a quattro.

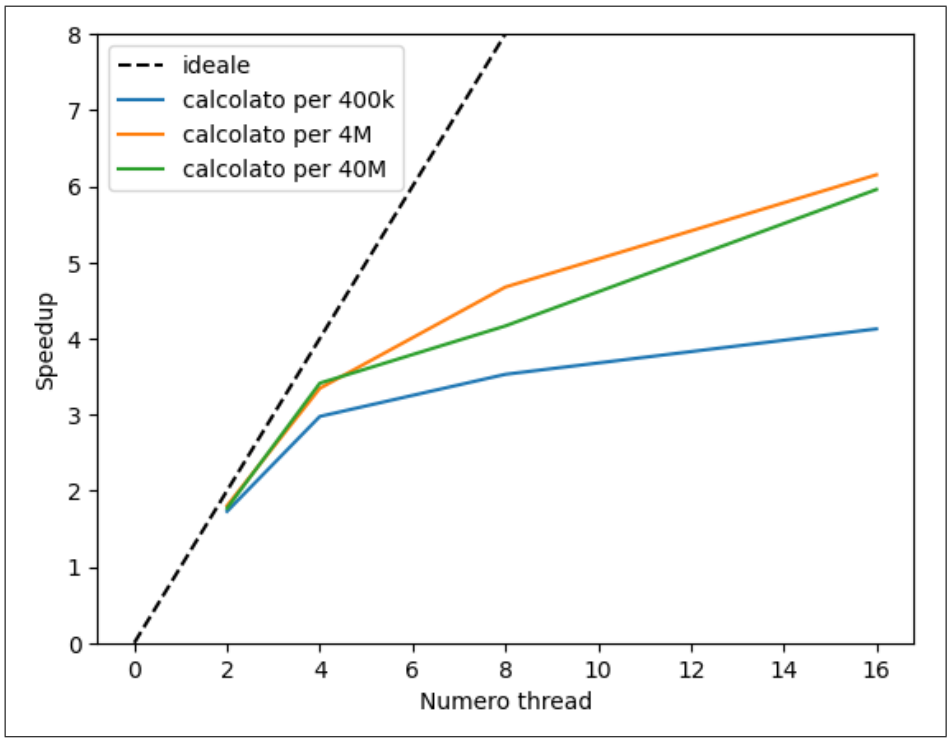

Figura 4: Calcolo speedup al variare del numero thread con numero di cluster pari a quattro. Il calcolo è eseguito su tre quantità diverse di punti: 400k, 4M e 40M.