Laboratorio 5 - Clustering k-means parallelo

Alberto Bezzon, Tommaso Carraro, Alberto Gallinaro

Premesse all'esercizio

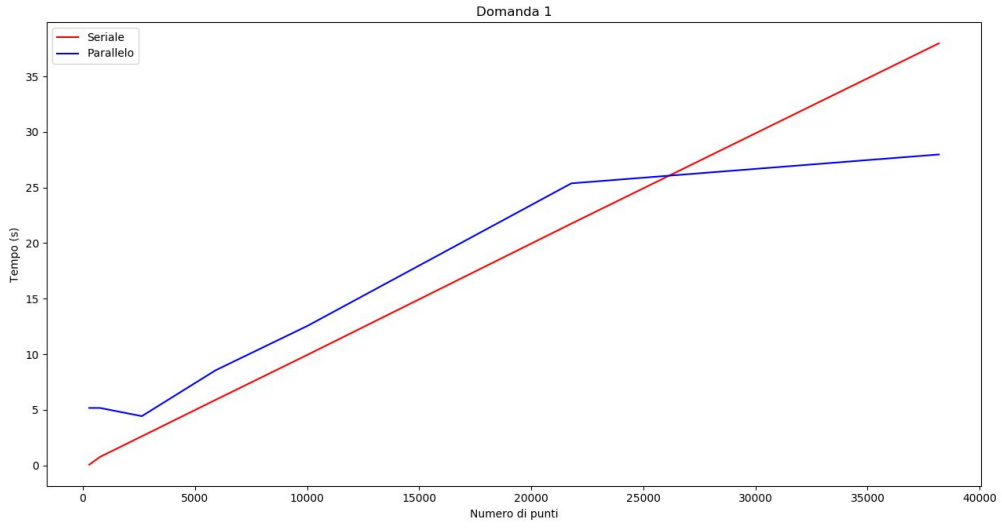
Per lo svolgimento dell'esercizio si è seguita la guida ufficiale Oracle linkata sulla sezione "Piattaforme per la concorrenza". Per utilizzare il fork/join framework di java si sono dovute effettuare le seguenti modifiche allo pseudocodice di k-means:

1. il parallel for per il calcolo delle distanze tra i punti e i cluster è stato trasformato in una classe Java che implementa una procedura divide-et-impera;
2. il parallel for per l'aggiornamento dei centroidi è stato trasformato in una classe Java che implementa una procedura divide-et-impera;
3. all'interno del ciclo for seriale che itera sul numero di iterazioni q si sono invocate queste due procedure parallele.

Purtroppo il framework non ha fornito i risultati sperati, infatti è risultato inefficiente rispetto all'algoritmo seriale in più di un esperimento. I risultati proposti sono frutto di un elevato numero di tentativi nel cercare il cutoff ottimale per rispondere ad ogni domanda. Nella maggior parte delle domande la soluzione è stata parallelizzare il meno possibile.
L'inefficienza è data dal funzionamento interno del framework, infatti il ForkJoinPool che si occupa di eseguire i task istanzia un nuovo thread ogni volta che non ci sono più thread disponibili (sono thread impegnati in altre chiamate parallele), il tutto per non far attendere l'utente. Quindi più si introduce parallelismo e più si aumenta l'inefficienza dell'algoritmo. In alcune esecuzioni il numero di thread è aumentato fino all'esaurimento della memoria.
Inoltre, ci si è accorti che una volta terminata la chiamata all'algoritmo parallelo, i thread continuano a rimanere in memoria e questo causa delle inefficienze con effetto a cascata sulle successive esecuzioni dell'algoritmo. Infatti, sono richieste circa 10 esecuzioni dell'algoritmo parallelo per ognuna delle domande. Tenendo un grado di parallelismo piuttosto basso si è riusciti a mitigare questo problema..
Infine, ci si è accorti che tramite l'utilizzo della libreria FastMath per il calcolo delle funzioni matematiche si sono ottenuti risultati di gran lunga migliori sia sulla versione seriale che sulla versione parallela.

Domanda 1

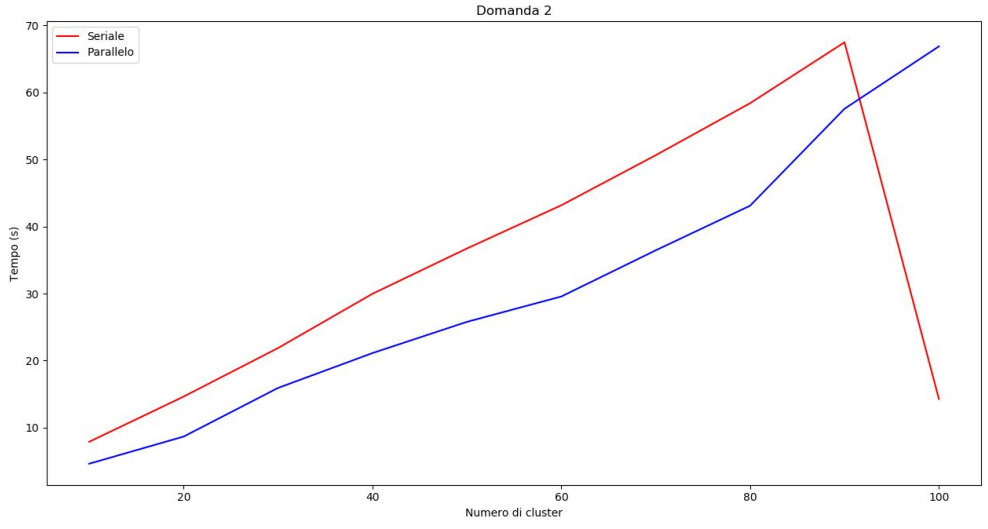
Soglie di popolazione minima	Algoritmo seriale	Algoritmo parallelo
	Tempo (s)	Tempo (s)
250	21.752	25.376
2 000	9.943	12.546
5 000	5.897	8.557
15 000	2.617	4.43
50 000	0.763	5.169
100 000	0.061	5.169
Tutta la popolazione	37.976	27.974



Per questa risposta è stato utilizzato un cutoff di 10000. Sembra infatti che introducendo meno parallelismo si ottengano risultati migliori per questa domanda. L'utilizzo dell'algoritmo standard senza cutoff ritornava risultati improponibili.
Come si può osservare, all'aumentare del numero dei punti nel dataset l'algoritmo seriale ha un andamento lineare nel tempo, mentre l'algoritmo parallelo si comporta allo stesso modo della versione seriale fino ai 20000 punti, per poi iniziare ad essere più efficiente.
Quindi, più il dataset è grande e più l'algoritmo parallelo funziona meglio della variante seriale.

Domanda 2

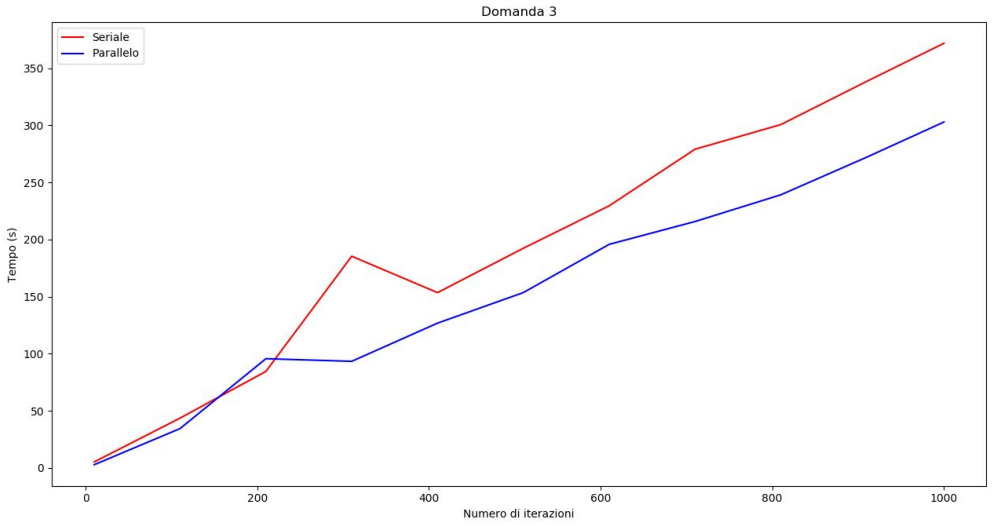
Numero di cluster	Tempo algoritmo seriale (s)	Tempo algoritmo parallelo (s)
10	7.9	4.632
20	14.641	8.678
30	21.877	15.93
40	29.966	21.124
50	36.719	25.78
60	43.162	29.56
70	50.626	36.445
80	58.385	43.098
90	67.464	57.554
100	14.281	66.858



Per questa risposta è stato utilizzato un cutoff di 10000. Come si può osservare dal grafico l'algoritmo parallelo si comporta sempre meglio della versione seriale. Nell'esecuzione con 100 cluster l'algoritmo seriale di comporta di gran lunga meglio rispetto all'algoritmo parallelo. Osservando l'andamento della curva si può però dedurre che si tratta di un esecuzione anomala legata al caso. Infatti, l'algoritmo seriale ha un andamento perfettamente lineare nel tempo a parte nell'esecuzione con 100 cluster.

Domanda 3

Numero di iterazioni	Tempo algoritmo seriale (s)	Tempo algoritmo parallelo (s)
10	5.198	2.751
100	43.559	34.39
200	84.469	95.605
300	185.344	93.31
400	153.481	126.746
500	192.346	153.525
600	229.524	195.767
700	279.074	215.747
800	300.654	239.152
900	338.471	272.093
1000	371.848	302.905

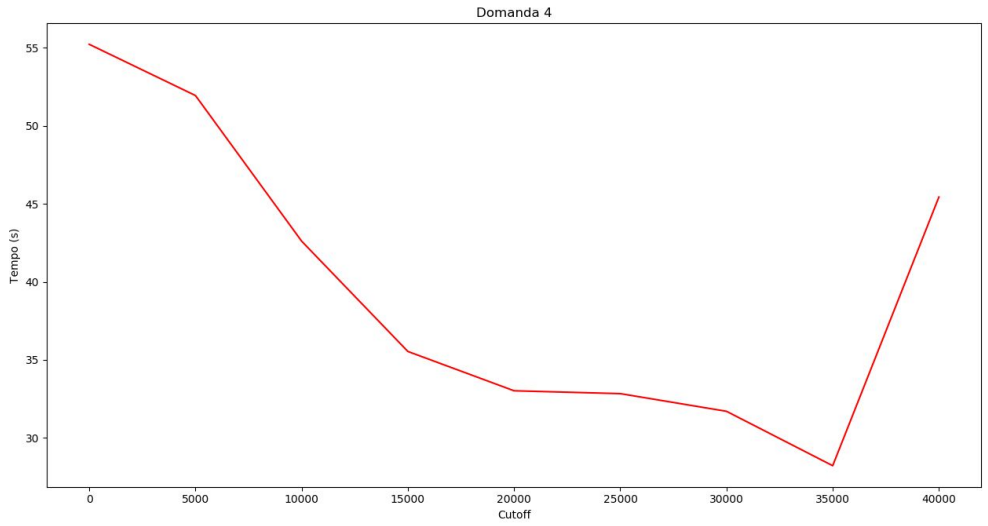


Per questa risposta è stato utilizzato un cutoff di 30000. Infatti, per ottenere dei buoni risultati è stato necessario introdurre meno parallelismo possibile poiché all'aumentare del numero di iterazioni aumentano anche il numero di thread in memoria e più thread ci sono in memoria, maggiore è l'inefficienza dell'algoritmo.
Come si può osservare, per questa domanda l'algoritmo parallelo fornisce risultati più efficienti rispetto alla versione seriale, infatti più iterazioni vengono effettuate in k-means e più

aumenta la sua complessità nella versione seriale. La versione parallela permette di ottenere uno span di O(q(log n + k)) e quindi riesce ad essere più efficiente nonostante l'aumento del numero di iterazioni.

Domanda 4

Soglie di cutoff	Tempo algoritmo parallelo (s)
0	55.231
5000	51.948
10000	42.607
15000	35.536
20000	33.014
25000	32.831
30000	31.698
35000	28.209
40000	45.44



Per questa domanda si sono analizzati i cutoff da 0 a 40000. In particolare si è cercato di capire se fosse più efficiente introdurre più o meno parallelismo nelle esecuzioni. Come si può notare dal grafico, meno parallelismo si introduce e più l'algoritmo acquista efficienza. Il cutoff 40000 corrisponde infatti al caso in cui non si inserisce parallelismo. La maggior efficienza si è ottenuta con il cutoff 35000. Questo significa che inserire il minimo grado di parallelismo possibile, senza diventare seriali, è la scelta migliore per questa domanda. Questo è causato dall'inefficienza del framework fork/join.

Domanda 5

Processore	Numero di core	Memoria	Sistema operativo
i5	4	8GB	Windows 10 64 bit