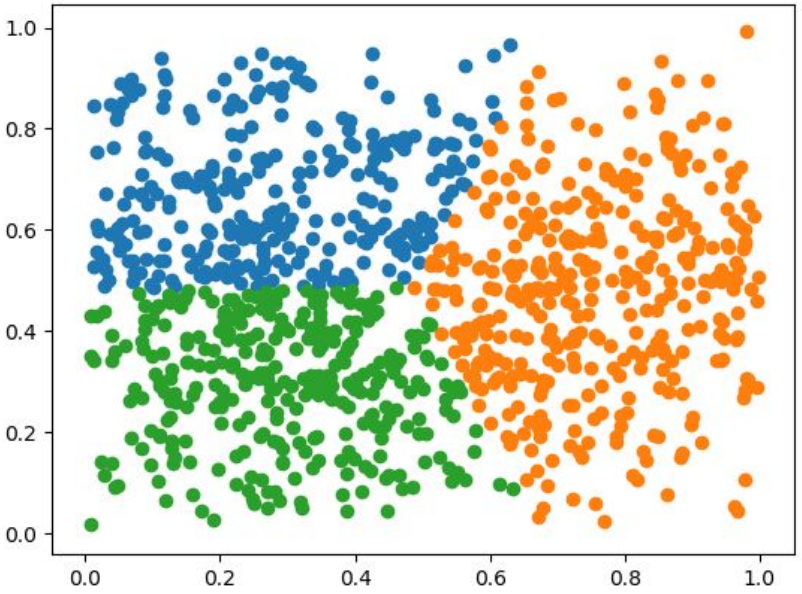
Implementazione dell’algoritmo “K-means clustering” con OpenMPI e OpenMP

L’algoritmo lavora su un insieme di punti (dataset) e tenta di raggruppare i punti in gruppi (cluster) in base alla loro somiglianza. Nella nostra implementazione abbiamo rappresentato i punti come vettori di reali. La somiglianza è stabilita dalla distanza euclidea fra due punti: più i due punti sono vicini nello spazio (n-dimensionale) e più è probabile che facciano parte dello stesso cluster. Un cluster è rappresentato da un centroide: un punto nello spazio che è il centro del cluster.

L’algoritmo procede come segue:

1. Vengono inizializzati i centroidi: gli viene assegnato un punto casuale del dataset
2. Ad ogni punto del dataset viene associato il cluster più vicino.
3. Vengono aggiornati i centroidi di tutti i cluster: i nuovi centroidi sono ottenuti dalla media dei punti che fanno parte del cluster.
4. Se i centroidi sono gli stessi dell’iterazione precedente o nessun punto ha cambiato cluster in questa iterazione o si è raggiunta una soglia massima di iterazioni termina altrimenti ritorna al punto 2.



Esempio: Applicazione dell’algoritmo con parametri: num_clusters=3, num_points=1000, num_attributes = 2

Parallelizzazione con OpenMPI

OpenMPI viene usato per il passaggio dei dati fra i processi che collaborano per identificare i cluster. All'inizio il processo con rank 0 legge le opzioni passate da linea di comando e le invia in broadcast a tutti gli altri processi. Dopodichè legge il dataset e divide i punti tra i processi sfruttando la funzione Scatterv. L'utilizzo della funzione Scatterv consente di specificare quanti elementi passare ad ogni processo in questo modo non si creano problemi se il numero di punti nel dataset non è divisibile per il numero di processi.
Dopo aver ricevuto i datapoints tutti i processi eseguono l'algoritmo localmente sui propri cluster locali. Una volta assegnati tutti i punti ai cluster vengono calcolati i nuovi centroidi parziali: ogni processo somma fra di loro tutti i punti che fanno parte di uno stesso cluster e conta quanti sono. In seguito i processi si scambiano le informazioni fra di loro e a questo punto hanno tutte le informazioni necessarie per aggiornare i centroidi dei cluster:

$$Centroide_i = \frac{\sum_{p \in cluster_i} p}{|cluster_i|}$$

Al termine di un iterazione quindi i processi avranno tutti i nuovi cluster globali. Per sapere quando i processi si devono fermare abbiamo utilizzato la funzione MPI_Allreduce su una variabile booleana che indica se per un dato processo in una data iterazione ci sono stati dei cambiamenti rispetto all'iterazione precedente. Se nessun processo ha riscontrato dei cambiamenti nei cluster il programma termina.

Parallelizzazione con OpenMP

Abbiamo modificato la versione seriale del programma ottimizzando tutti i cicli for (significativi) con delle pragma omp for. La creazione dei thread avviene una volta sola, per motivi di efficienza, prima del ciclo principale dell'algoritmo. Per valutare la correttezza del programma così ottenuto, abbiamo confrontato i risultati con quelli ottenuti dalla versione OpenMPI e seriale.
Nonostante i risultati siano corretti il programma impiega più tempo della versione seriale. Per capire il motivo di questa inefficienza abbiamo scritto più versioni del programma sfruttando strutture dati diverse per rappresentare i punti e modificando l'uso di OpenMP. La nostra ipotesi è che l'inefficienza di un approccio multi thread è causata dal false sharing che in questo caso è causato dall'accesso all'array contenente i punti del dataset. Per indagare di più sulle cause della lentezza abbiamo anche realizzato una versione con pthread che presenta la stessa problematica.

Valutazione di efficienza MPI e comparazione con versione seriale

Il K-Means clustering ha sostanzialmente tre parametri di input:

1) Il numero di datapoint
2) Il numero di dimensioni di ciascun datapoint (il numero di attributi)
3) Il numero di cluster in cui dividere gli attributi

Questi parametri, aggiunti al numero di processi, determinano il tempo di esecuzione.
Facendo diverse prove abbiamo notato come con l'aumentare del numero di processi il tempo di esecuzione diminuisce (si ha uno speedup) ma l'efficienza decresce sempre di più.
Abbiamo inoltre notato che al crescere della dimensione dell'input (in particolare del numero di datapoints che abbiamo fatto arrivare fino a 10 milioni) si ha uno speedup sempre maggiore tra la versione seriale e quella parallela.
L'efficienza maggiore si ha tuttavia quasi sempre utilizzando 2 processi, probabilmente a causa dell'overhead dovuto alla comunicazione tramite MPI.

Sperimentalmente abbiamo ottenuto i seguenti risultati:

1’000 Datapoint

1000 datapoint , 2 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,20 e-03	1	1
MPI	2 4 8	1,17 e-03 0,53 e-03 0,71 e-03	1,02 2,25 1,69	0,51 0,56 0,21

1000 datapoint, 2 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	3,78 e-03	1	1
MPI	2 4 8	2,19 e-03 1,09 e-03 1,09 e-03	1,73 3,46 3,46	0,86 0,87 0,43

1000 datapoint, 4 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	4,63 e-03	1	1
MPI	2 4 8	2,88 e-03 1,22 e-03 1,91 e-03	1,61 3,79 2,42	0,80 0,94 0,30

1000 datapoint, 4 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	5,44 e-03	1	1
MPI	2 4 8	3,63 e-03 2,94 e-03 3,92 e-03	1,50 1,82 1,39	0,75 0,45 0,17

100’000 Datapoint

100000 datapoint , 2 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	9,35 e-02	1	1
MPI	2 4 8	5,85 e-02 3,41 e-02 3,89 e-02	1,60 2,74 2,40	0,80 0,68 0,30

100000 datapoint , 2 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,56 e-01	1	1
MPI	2 4 8	0,83 e-01 0,48 e-01 0,36 e-01	1,88 3,25 4,33	0,94 0,81 0,54

100000 datapoint, 4 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,29 e-01	1	1
MPI	2 4 8	0,75 e-01 0,45 e-01 0,42 e-01	1,72 2,87 3,07	0,86 0,71 0,38

100000 datapoint, 4 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,15 e+00	1	1
MPI	2 4 8	0,62 e+00 0,37 e+00 0,32 e+00	1,85 3,11 3,59	0,92 0,78 0,45

1’000’000 Datapoint

1000000 datapoint , 2 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	8,82 e-01	1	1
MPI	2 4 8	4,96 e-01 3,25 e-01 8,90 e-01	1,78 2,71 0,99	0,89 0,67 0,12

1000000 datapoint , 2 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,84 e+00	1	1
MPI	2 4 8	0,95 e+00 1,40 e+00 1,27 e+00	1,94 1,31 1,45	0,97 0,33 0,18

1000000 datapoint, 4 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,73 e+00	1	1
MPI	2 4 8	0,94 e+00 0,51 e+00 0,57 e+00	1,84 3,39 3,04	0,92 0,85 0,38

1000000 datapoint, 4 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,31 e+01	1	1
MPI	2 4 8	0,76 e+01 0,49 e+01 0,39 e+01	1,72 2,67 3,36	0,86 0,66 0,42

10’000’000 Datapoint

10000000 datapoint , 2 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	1,21 e+01	1	1
MPI	2 4 8	0,65 e+01 0,46 e+01 0,66 e+01	1,86 2,63 1,84	0,93 0,66 0,23

10000000 datapoint , 2 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	3,62 e+01	1	1
MPI	2 4 8	1,72 e+01 1,26 e+01 1,77 e+01	2,01 2,87 2,04	1 0,72 0,25

10000000 datapoint , 4 attributi, 2 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	2,42 e+01	1	1
MPI	2 4 8	1,37 e+01 0,73 e+01 0,71 e+01	1,77 3,32 3,41	0,89 0,83 0,43

10000000 datapoint , 4 attributi, 4 cluster:

	# Processi / Thread	Tempo	Speedup	Efficienza
Versione Seriale	1	2,01 e+02	1	1
MPI	2 4 8	1,62 e+02 1,54 e+02 1,47 e+02	1,24 1,30 1,36	0,62 0,32 0,17

Conclusioni

In conclusione la nostra implementazione dell'algoritmo K-Means Clustering si presta ad essere parallelizzato su macchine a memoria distribuita.
Nelle macchine a memoria condivisa si presenta invece il problema del false sharing durante l'aggiornamento dell'appartenenza di un datapoint ad un cluster, che si può risolvere creando una copia locale (non shared) di parti del dataset per ciascun thread.

Realizzato da: Alessandro Torri e Leonardo La Rocca