Relazione Progetto APSD

Algoritmi di Ordinamento

1. Come eseguire il programma

Per compilare il programma su Linux:
make

Per eseguire il programma:
mpirun -np <num_processori> **./sorting.out** <dimensione_sequenza> <grafica>

esempio: **mpirun -np** 4 **./sorting.out** 1024 1

2. Breve descrizione

Sono stati implementati i seguenti algoritmi di ordinamento:

- Bitonic Sort
- Odd-even transposition
- Parallel Quicksort

Viene generata una sequenza random con la dimensione equivalente al numero fornito da linea di comando (dimensione_sequenza). Tale sequenza viene scritta su un file .txt preceduta dalla size della stessa. Tutti gli algoritmi ordinano la stessa sequenza per poter prendere i tempi di esecuzione corretti.

La grafica è stata realizzata con la libreria “Allegro5. Si fa notare che la velocità cambia se si decide di visualizzare graficamente l’esecuzione degli algoritmi di ordinamento. Se si decidere di verificare i tempi effettivi degli algoritmi, bisogna eseguire il programma senza la visualizzazione grafica. Per semplicità, la sequenza visualizzata graficamente è quella del processore *MASTER*.

Le stampe, il calcolo dei tempi, alcuni controlli, generazione di numeri random e lettura dei numeri avvengono solo dal processore *MASTER*.

La sequenza da ordinare viene distribuita equamente su tutti i processori (sistema democratico) tramite *MPI_SCATTER*. Ogni processore esegue l’algoritmo di ordinamento sequenziale sulla sua porzione di array, per poi unire la sequenza in un unico array tramite *MPI_GATHER*.

3. Bitonic Sort

Il Bitonic Sort è un algoritmo di ordinamento che consiste nella creazione di una sequenza bitonica, partendo da una sequenza non ordinata.
La sequenza bitonica è una sequenza il cui valore degli elementi prima aumenta e poi diminuisce.
Per far sì che l'algoritmo funzioni correttamente, il numero degli elementi dev'essere una potenza di 2.

Per prima cosa iniziamo dividendo la sequenza di dati in *n* parti, dove *n* è il numero di processi, per poi inviare una porzione a ciascun processo. Ogni processo ordina la propria sottosequenza e successivamente si avranno *n* sottosequenze di dati ordinati che devono essere unite per creare un unico risultato contiguo.

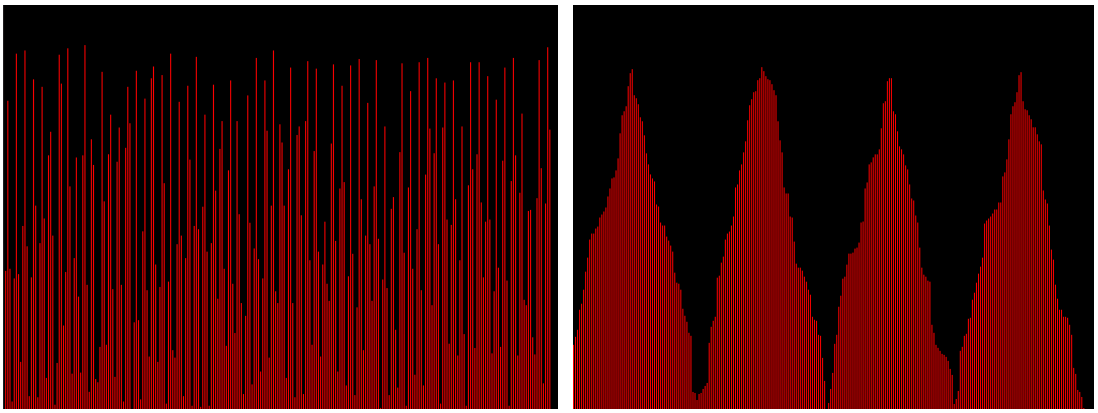
I dati verranno ordinati globalmente in log2 n passi. Questi passi seguiranno il modello di ordinamento bitonico standard.
I primi passi servono a creare una sequenza bitonica e il resto ad ordinarla. Ogni processo effettua un numero di scambi con vari "partner".

Vengono utilizzate delle chiamate bitonicSortCrescente o bitonicSortDecrescente. In questi metodi i dati dei partner vengono inviati l'uno all'altro e poi ognuno chiama una funzione di unione (unisciSequenza), che unisce solo la metà dei dati.

unioneSequenzaHigh:
Trasferisce i dati al processo partner
Riceve i dati del processo partner
Inizia ad unire i dati della sequenze di tipo **HIGH** fino a quando non ha unito dim elementi, dove di è la lunghezza dell'array dei dati originale

unioneSequenzaLow:
lo stesso tranne che inizia ad effettuare l'unione dalla sequenza di tipo **LOW**

Una volta completato tutto, i dati di tutti i processi vengono rispediti al master e sono in ordine. A causa del modo in cui i dati sono ordinati, questo è un algoritmo di ordinamento instabile.



Tempi:

Thread	1024	16384	131072	1048576
1	0,0004976	0,0140633	0,129507	0,746101
2	0,0018371	0,0065515	0,0421884	0,320929
4	0,0085829	0,0056829	0,0216644	0,220796
8	0,0015382	0,0200332	0,0203935	0,185531

SpeedUp ($TempoSequenziale/TempoParallelo$) si assume che il tempo sequenziale è pari al tempo parallelo eseguito con 1 thread

Thread	1024	16384	131072	1048576
1	100%	100%	100%	100%
2	27,09%	214,66%	306,97%	232,48%
4	5,80%	247,47%	597,79%	337,91%
8	32,35%	70,20%	635,04%	402,14%

4. Odd Even Transposition

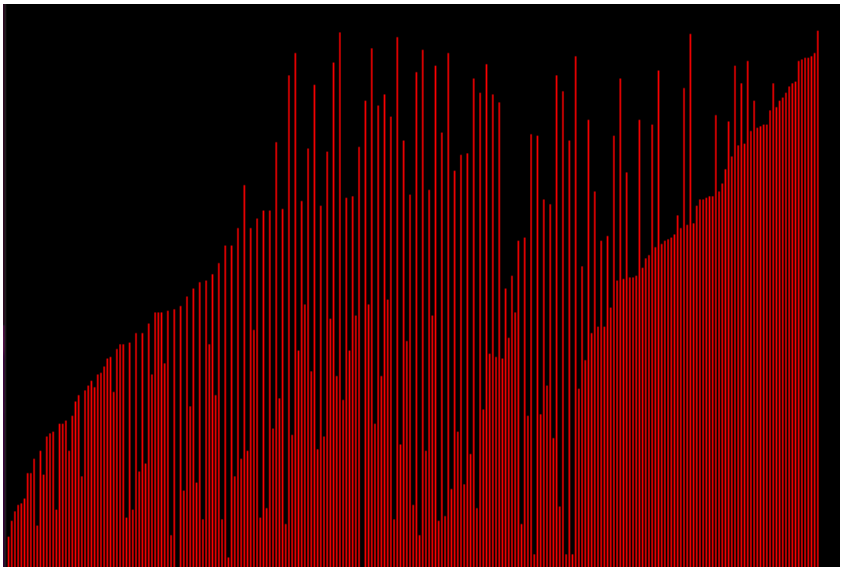
Esso opera comparando tutte le coppie dispari e pari degli elementi presenti in una lista e, se una coppia è nell'ordine sbagliato (il primo elemento è maggiore del secondo), scambia di posto i suoi elementi. Il controllo prosegue con le coppie di elementi adiacenti con posizione pari/dispari. L'algoritmo continua l'ordinamento alternando tra le comparazioni dispari/pari e pari/dispari finché tutta la lista non risulta ordinata.

Tempi:

Thread	1024	16384	131072	1048576
1	0,0016376	0,2690180	18,7366000	321,1225100
2	0,0007940	0,1070300	4,5741000	273,6657690
4	0,0004109	0,0226982	1,2081200	80,3679420
8	0,0014497	0,0213565	0,4750250	32,4912000

SpeedUp ($TempoSequenziale/TempoParallelo$) si assume che il tempo sequenziale è pari al tempo parallelo eseguito con 1 thread

Thread	1024	16384	131072	1048576
1	100%	100%	100%	100%
2	206,25%	251,35%	409,62%	117,34%
4	398,54%	1185,20%	1550,89%	399,57%
8	112,96%	1259,65%	3944,34%	988,34%



5. Quick Sort

Il quicksort è un algoritmo di tipo divide et impera partizionando gli elementi in due sequenze che vengono ordinate ricorsivamente e poi riunirle. La parte di divide, sceglie un elemento della sequenza, chiamato pivot, e partiziona la sequenza in due parti una con gli elementi minori uguali al numero pivot e quelli maggiori nella sottosequenza “di destra” per poi ordinare ricorsivamente le due sottosequenze ottenute. Infine si concatenano le sottosequenze ordinate.

Tempi:

Thread	1024	16384	131072	1048576
1	0,0001264	0,0029607	0,0136256	0,1332570
2	0,0002897	0,0018367	0,0117243	0,0758822
4	0,0006664	0,0014889	0,0093129	0,0746069
8	0,0003114	0,0097465	0,0132725	0,0737833

SpeedUp ($TempoSequenziale/TempoParallelo$) si assume che il tempo sequenziale è pari al tempo parallelo eseguito con 1 thread

Thread	1024	16384	131072	1048576
1	100%	100%	100%	100%
2	43,63%	161,20%	116,22%	175,61%
4	18,97%	198,85%	146,31%	178,61%
8	40,59%	30,38%	102,66%	180,61%