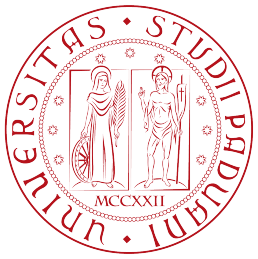


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

BITONIC SORT

Lavoro di gruppo per il corso:

Calcolo Parallelo A.A. 2018/19

Autori:

Cavallin	Storato
Massimo	Davide
1159787	1153692

Supervisore: Prof. Gianfranco Bilardi

22 luglio 2019

Indice

1 Scopo del progetto

Lo scopo del progetto è l'implementazione dell'algoritmo di ordinamento bitonic sort utilizzando il linguaggio di programmazione C++ e la libreria openMPI[1]. L'algoritmo viene eseguito nel cluster dipartimentale Power7 e i dati raccolti vengono utilizzati per capire le performance dell'algoritmo al variare del numero di processori disponibili, della dimensione dell'input da ordinare e dal tipo di ottimizzazioni del compilatore.

2 Descrizione dell'algoritmo

Il bitonic sort è un algoritmo parallelo, basato su confronti, per l'ordinamento di sequenze ideato da K.E. Batcher[2]. L'algoritmo è basato sul concetto di convertire qualunque sequenza data in una sequenza bitonica per poi ordinarne gli elementi. Una sequenza si definisce *bitonica* se i suoi elementi rispettano la seguente proprietà: $x_0 \leq x_1 \leq \cdots \leq x_k \geq x_{k+1} \geq \cdots \geq x_{n-1}$ per qualche k, $0 \leq k < n$.
Si può notare che questo comportamento è molto simile a quello di un gradiente che sale e poi scende di valore. Il punto in cui il gradiente si inverte è chiamato *punto bitonico*. Inoltre, il bitonic sort funziona solo su sequenze con una lunghezza che è una potenza di 2.
Una sequenza ordinata in ordine crescente è considerata bitonica se la corrispondente parte decrescente è vuota. Lo stesso vale se la sequenza è ordinata in ordine decrescente. Inoltre, una rotazione di una sequenza bitonica è anch'essa bitonica.

L'algoritmo è basato su confronti dato dai *comparator exchanger*, fig. 1. Si tratta di comparatori che dati due valori in input in un estremo dell'output del comparatore si ha il valore minore e nell'altro il valore maggiore tra i due.

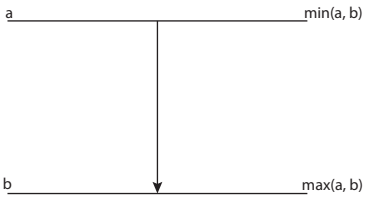


Figura 1: Comparator exchanger.

La rete per il bitonic sort incomincia con la costruzione della sequenza bitonica data la sequenza in input. Successivamente, la sequenza viene divisa in parti sempre più piccole e ricomposta fino ad ottenere la sequenza ordinata. Da come si può intuire il bitonic sort è basato sul paradigma *divide and conquer*. Come illustrato in fig. 2, si può vedere che i primi tre step costituiscono la fase di divide dell'algoritmo mentre l'ultimo step è la fase di conquer. È quest'ultima parte dell'algoritmo a richiedere che la sequenza in ingresso sia bitonica per poter ottenere alla fine la sequenza ordinata.

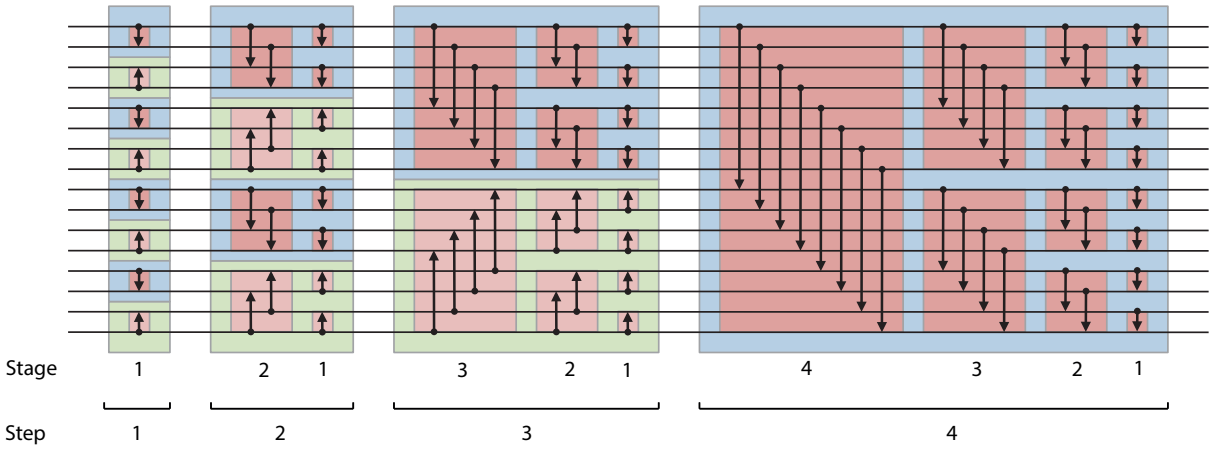


Figura 2: Rete bitonic sort con 16 elementi[3].

La rete bitonic sort gode del *principio 0-1* il quale afferma che se una rete di comparatori riesce a ordinare qualunque sequenza binaria allora è in grado di ordinare ogni sequenza formata da un insieme di valori arbitrari.

Dal punto di vista computazionale il bitonic sort ha una complessità temporale di $\Theta(Nlog^2N)$, con N il numero di elementi da ordinare. Questa complessità è dovuta agli $N/2$ confronti che vengono fatti ad ogni stadio della rete e dal numero di stadi totali della rete, che è circa log^2N.

3 Dettagli implementativi

L'algoritmo incomincia eseguendo un parse dell'input per prendere i parametri, passati in ingresso dall'utente, riguardanti il numero di processori e la dimensione dell'array da ordinare seguito dall'inizializzazione dell'ambiente di esecuzione MPI con il metodo `MPI_Init`. Successivamente si verifica che il numero di elementi nell'array sia una potenza di 2. Nel caso non è una tale potenza viene eseguito un padding dell'array di input alla prima potenza di 2 che contiene la dimensione data dall'utente. Una volta creato l'array della giusta dimensione viene popolato con numeri naturali utilizzando la funzione di libreria random.

Popolato l'array, l'algoritmo utilizza un modello di comunicazione master-slave in cui un processore ha controllo su uno o più processori. Con questo modello il processore master, una volta ottenuto l'array popolato con i dati, calcola il numero di elementi che ogni processore dovrà avere in locale. Ogni processore slave alloca lo spazio necessario per contenere la porzione di array che verrà a lui assegnata dal master. Con il metodo `MPI_Scatter` il processore master manda le porzioni di array ai processori slave. Ogni processore ordina localmente la porzione di array ricevuto utilizzando la funzione di libreria `qsort` in modo da ottenere una sequenza bitonica e tutti i procesori vengono sincronizzati con il metodo `MPI_Barrier`. Quando si hanno degli scambi di dati tra i processori si è scelto di inviare intere porzioni di array invece dei singoli elementi così da avere una riduzione nel numero di scambi tra i processori.

Una volta che i processori sono sincronizzati incomincia la parte di comunicazione tra i processori. Ogni processore calcola il valore del processore con cui dovrà comunicare (partner) a seconda dello step e dello stage in cui si trova, un esempio degli step e degli stage lo si può vedere in fig. 2. Una volta ottenuto il valore del partner avviene lo scambio della porzione di array che hanno in locale con i metodi non bloccanti `MPI_Isend` e `MPI_Irecv`. Questi due metodi vengono usati in sequenza in modo diverso a seconda che la direzione è ascendente (prima invio, poi ricevo) o discendente (prima ricevo, poi invio). Essendo i metodi non bloccanti è necessario, prima di effettuare altri passi dell'algoritmo, attendere che tutte le comunicazioni siano terminate, questo lo si può fare utilizzando il metodo `MPI_Waitall`.
Terminate tutte le comunicazioni avviene il merge dell'array locale con quello ricevuto e il processore salva la prima metà dell'array totale (quella con i valori minori) se la direzione è ascendente o la seconda metà (quella con i valori maggiori) se la direzione è discendente.
Una volta eseguiti tutti gli step, il processore master raccoglie tutti gli array locali dei processori slave ottenendo l'array finale con il metodo `MPI_Gather`. Una routine viene eseguita dal master per verificare se l'array è correttamente ordinato e l'esito di tale verifica viene visualizzato nello standard output assieme ai tempi di comunicazione e di esecuzione totali.

4 Risultati

Per studiare le performance dell'algoritmo bitonic sort implementato sono stati effettuati cinque test sul cluster dipartimentale Power7 per ogni diverso livello di ottimizzazione del compilatore, taglia della dimensione e numero di processori. Come valori per misurare le performance si è deciso di usare i tempi di esecuzione e i tempi di comunicazione. I test sono stati eseguiti con i seguenti livelli di ottimizzazione: solo compilazione (di seguito chiamata no opt), O0, O2, O3 e Os. Per ogni livello di ottimizzazione del compilatore si usano 2, 4, 8, 16 e 32 processori con i quali si risolvono istanze di dimensione 32K, 64K, 128K, 256K, 512K, 1M, 2M, 4M, 8M e 16M.
Ottenuti i dati li abbiamo analizzati e abbiamo calcolato la media geometrica di tutti i risultati ottenuti per le varie ottimizzazioni e per i vari test in modo da individuare la migliore dal punto di vista del tempo di esecuzione medio. Anche i tempi di esecuzione e di comunicazione per ogni combinazione del numero di processori con la dimensione dell'istanza sono medie geometriche dei valori ottenuti dai cinque test.

La *media geometrica* di n elementi è definita come la radice n-esima del prodotto degli n valori:

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 \cdot x_2 \cdot \ldots \cdot x_n}.$$

Come si vede in fig. 3, i tempi medi di esecuzione migliori si ottengono con l'ottimizzazione O3, seguita dalla ottimizzazione O2 (con una differenza molto piccola), Os, O0 e no opt.
Di seguito vengono descritte le varie ottimizzazioni a partire delle migliori dal punto di vista del tempo di esecuzione medio. Per motivi di spazio sono stati inseriti solo i grafici delle due ottimizzazioni migliori, vale a dire la O2 e O3.

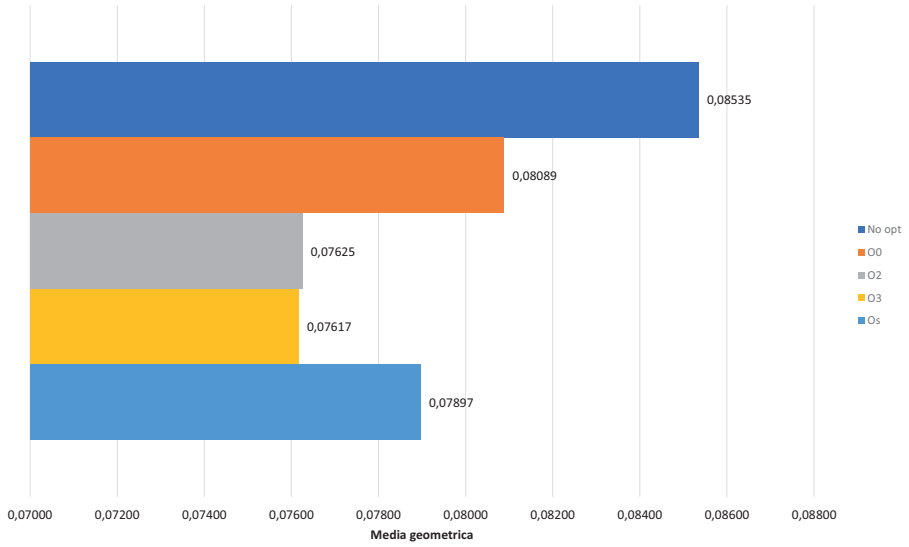
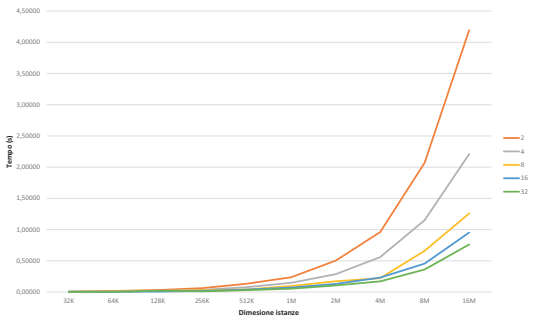


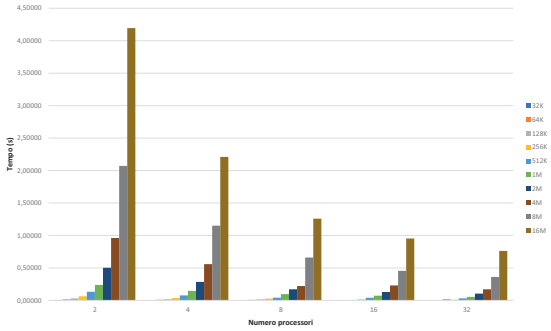
Figura 3: Media geometrica dei tempi di esecuzione per le varie ottimizzazioni.

4.1 Ottimizzazione O3

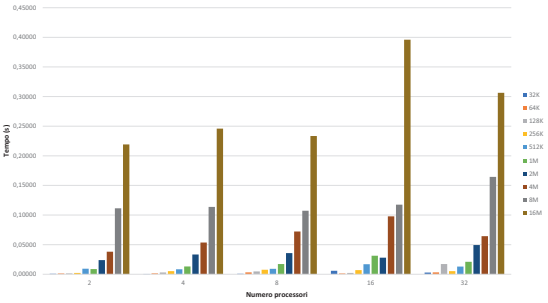
L'ottimizzazione O3, come si vede in fig. 3, è quella che ci ha dato la migliore media dei tempi di esecuzione. In figura fig. 4 si può vedere come variano i tempi di esecuzione al variare della dimensione dell'istanza (fig. 4a) e del numero di processori (fig. 4b), la variazione del tempo di comunicazione al variare del numero di processori (fig. 4c) e il rapporto tra il tempo di esecuzione e il tempo di comunicazione al variare del numero di processori (fig. 4d).
Come si può vedere dalla fig. 4a, all'aumentare della dimensione dell'istanza si ha un incremento del tempo totale di esecuzione. Mentre all'aumentare del numero di processori si ha una riduzione dei tempi di esecuzione, come illustrato in fig. 4b.
Per quanto rigurada i tempi di comunicazione, a parte alcuni casi particolari, si ha un aumento dei valori all'aumentare della dimensione dell'istanza e all'aumentare del numero di processori, come si vede in fig. 4c.
Si può notare dalla fig. 4d che all'aumentare del numero dei processori il tempo dedicato alle comunicazioni aumenta. L'andamento del rapporto al variare della dimensione dell'istanza non è lineare, questo pensiamo sia dovuto alla posizione fisica dei processori utilizzati per eseguire l'algoritmo.



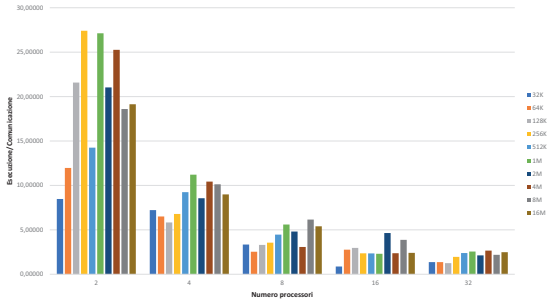
(a) *Tempi di esecuzione al variare della dimensione dell'istanza.*



(b) *Tempi di esecuzione al variare del numero di processori.*



(c) *Tempi di comunicazione al variare del numero di processori.*



(d) *Rapporto tempo di esecuzione e tempo di comunicazione.*

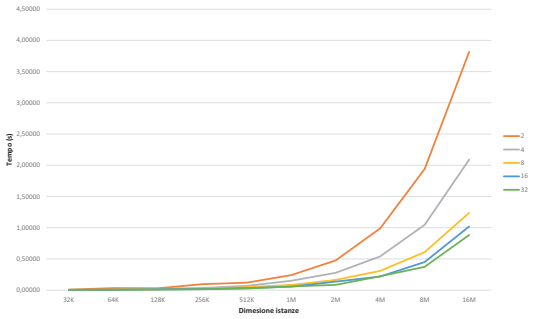
Figura 4: Dati ottenuti con ottimizzazione O3.

4.2 Ottimizzazione O2

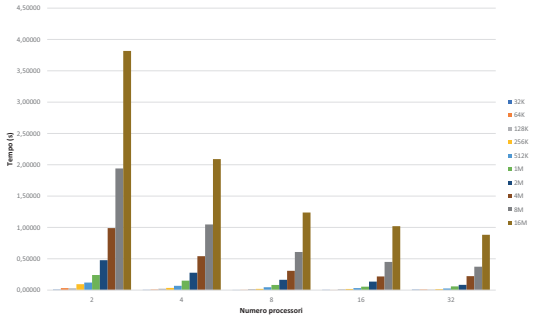
L'ottimizzazione O2, come si vede in fig. 3, è la seconda migliore per quanto riguarda la media dei tempi. Rispetto all'ottimizzazione O3, va meglio dal punto di vista del tempo di esecuzione per istanze di taglia maggiore o uguale a 1M mentre va peggio per le istanze di taglia minore. Anche per quanto riguarda i tempi di comunicazione sono migliori rispetto all'ottimizzzazione O3 per le dimensioni dell'istanza grandi, mentre va peggio per le dimensioni dell'istanza piccole. Anche per l'ottimizzazione O2 vale ciò che è stato detto per l'ottimizzazione O3 riguardo ai tempi di comunicazione. In fig. 5 sono riportati i tempi di esecuzione e di comunicazione medi e il rapporto tra tempo di esecuzione e tempo di comunicazione per l'ottimizzazione O2.

4.3 Restanti ottimizzazioni

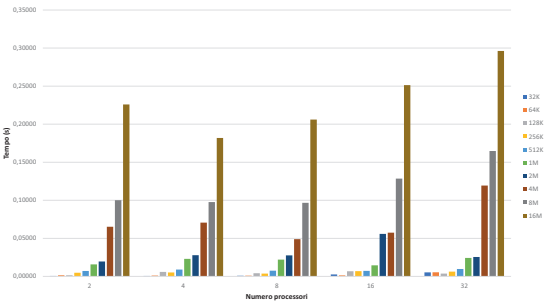
Per quanto riguarda le rimanenti ottimizzazioni (no opt, O0 e Os) si è visto che l'andamento del tempo di esecuzione all'aumentare del numero di processori o della dimensione dell'istanza è quasi uguale a quello delle ottimizzazioni descritte precedentemente. L'unica diversità si ha nel fatto che non c'è una differenza rilevante da far prediligere un'ottimizzazione rispetto ad un'altra sulla base della taglia piccola o grande dell'istanza come è accaduto per le ottimizzazioni O2 e O3. Per quanto ri-



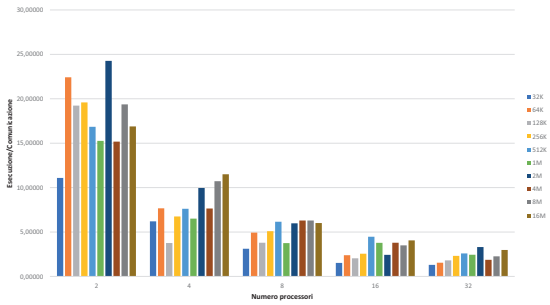
(a) *Tempi di esecuzione al variare della dimensione dell'istanza.*



(b) *Tempi di esecuzione al variare del numero di processori.*



(c) *Tempi di comunicazione al variare del numero di processori.*



(d) *Rapporto tempo di esecuzione e tempo di comunicazione.*

Figura 5: Dati ottenuti con ottimizzazione O2.

guarda i tempi di comunicazione e del rapporto tempo di esecuzione con il tempo di comunicazione vale quanto già detto per i casi precedentemente descritti.

5 Conclusione

Come si è visto tra le due migliori ottimizzazioni nessuna domina in tutti gli aspetti rispetto all'altra. Infatti si è visto che mentre una ottiene tempi di esecuzione migliori per istanze grandi, non si ha lo stesso anche per le istanze di taglia piccola. Inoltre abbiamo notato un miglioramento inaspettato dei tempi medi tra l'utilizzo dell'ottimizzazione O0 e la semplice compilazione.
Per la parte di programmazione è stata interessante la differenza tra l'utilizzo del programma in locale e nel cluster. Dove, nel secondo caso, sono sorti alcuni problemi nell'implementazione non visibili in locale. Inoltre abbiamo constatato quanto più complesso è il debug rispetto al caso sequenziale.
Di ulteriore interesse potrebbe essere il confronto del bitonic sort con altri algoritmi di ordinamento paralleli.

Riferimenti bibliografici

[1] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[2] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[3] Wikipedia contributors. Bitonic sorter — Wikipedia, the free encyclopedia, 2019. URL https://en.wikipedia.org/wiki/Bitonic_sorter. [Online; accessed 16-July-2019].