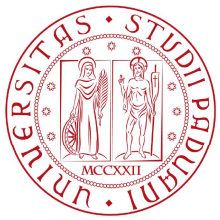
UNIVERSITÀ DEGLI STUDI DI PADOVA

Corso di Laurea Magistrale in Ingegneria Informatica

Calcolo Parallelo



Bitonic sort su ipercubo

Professore:
Michele Schimd

Studenti:
Davide Martini 1183732
Simone Nigro 1183535

Anno Accademico 2018-2019

1 Introduzione al problema

Il problema dell'ordinamento è un problema diffuso nel mondo dell'informatica. Esistono molti algoritmi sequenziali per la sua risoluzione, alcuni sono il Quicksort, Heapsort, e Mergesort che nel caso medio raggiungono prestazioni con andamento $O(nlog\,n)$. Molte analisi hanno portato a creazioni di algoritmi paralleli, tra i quali il Bitonic sort che nel caso medio ha un tempo di esecuzione $O(log^2 n)$.
In questa relazione tratteremo il Bitonic sort implementato su una topologia di rete ad ipercubo binario. Questa struttura tiene in considerazione delle limitazioni in campo reale, in quanto il numero di processori è limitato e ogni nodo (rappresentato da un processore) può comunicare con un sottoinsieme di nodi e non con tutti. Ogni nodo inoltre ha a disposizione una quantità di memoria limitata dalle caratteristiche tecniche dell'hardware dove viene eseguito.

2 Bitonic sort

Il Bitonic sort ha un funzionamento simile a quello del Mergesort, ed è stato proposto da Kenneth E. Batcher nel 1964 e pubblicato nel 1968 [1]. Il Bitonic sort implementa il paradigma *Divide and Conquer*. Nella fase di *Divide* la sequenza da ordinare viene divisa in due sottosequenze che vengono ordinate singolarmente, nella fase di *Conquer* le due sequenze ordinate vengono confrontate e viene ottenuta la sequenza ordinata voluta. Il parallelismo del Mergesort manca nell'operazione di merge, bisogna quindi trovare un modo più efficiente di svolgere tale operazione. Supponiamo di avere due sequenze A e B, non necessariamente della stessa lunghezza e di voler fare il merge. Prendo B, la rovescio e la concateno con A, in questo modo utilizzo due processori, uno che parte dai massimi e uno dai minimi e li ordina. Nel caso in cui avessimo più di due processori, è possibile suddividere ogni stringa a metà e su ognuna di queste applico due processori come visto prima. Ma questo non è corretto.
Dopo numerosi tentativi Batcher arrivò alla soluzione finale. Per comprendere questo algoritmo è necessario dare le seguenti definizioni:

- Una sequenza $(x_0,\ \ldots,\ x_{N-1})$ si dice bitonica se esiste una sua rotazione che è in forma 'sali-scendi'. Esiste un $j \in \{0,\ \ldots,\ N-1\}$ tale che ponendo
$$(y_0,\ \ldots,\ y_{N-1}) = SHIFT_j(x_0,\ \ldots,\ x_{N-1})$$
si ha che
$$\exists\, i \in \{0,\ \ldots,\ N-1\} \mid y_0 \leq y_1 \leq \ldots \leq y_i \geq y_{i+1} \geq \ldots \geq y_{N-1}$$

- L'operazione left cyclic shift è definita come:
$$SHIFT_j(x_0,\ \ldots,\ x_{N-1}) = (x_j,\ \ldots,\ x_{N-1},\ x_0,\ \ldots,\ x_{j-1})$$

- Sia N pari e $\vec{x} = (x_0, \ \ldots, \ x_{N-1})$. Comparison exchange è l'operazione che confronta e scambia direttamente due elementi della sequenza:

Figura 1: Comparator exchange

Dove $OUT_1 = min(IN_1, IN_2)$ e $OUT_2 = max(IN_1, IN_2)$. La notazione che viene solitamente usata per rappresentare un comparator exchange è la seguente:

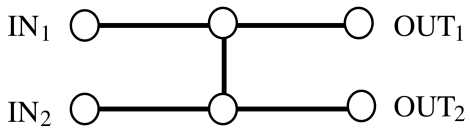


Figura 2: Rappresentazione convenzionale comparator exchange

Data quindi qualsiasi sequenza bitonica $\vec{x}$ posso effettuare l'ordinamento attraverso l'algoritmo di Batcher. Posiziono sui nodi in input la sequenza iniziale e applico i comparator exchange in modo tale da effettuare gli scambi necessari per produrre la sequenza ordinata in output. Come mostrato nella Figura 3 è possibile notare che se prendo una sequenza bitonica $\{3, 5, 7, 8, 6, 4, 2, 1\}$ ed effettuo i confronti seguendo la struttura riportata è possibile raggiungere in tre fasi l'output ordinato. In generale data in input una sequenza bitonica $\vec{x}$ lunga $N = 2^k$ con $k \ \in \ N^+$, dati N processori e una sequenza di confronti tramite comparator excange (seconda la struttura riportata) posso ordinare $\vec{x}$ in $log_2 n$ passi.
Sfortunatamente, le sequenze bitoniche sono molto rare. Per ovviare a questo problema è possible generare un algoritmo veloce per unire due sequenze ordinate. Supponiamo di avere due sequenze ordinate $\{x_0, \ \ldots \ , x_{m-1}\}$ e $\{y_0, \ \ldots \ , y_{m-1}\}$, se invertiamo la seconda sequenza e la concateniamo con la prima, il risultato è $\{x_0, \ \ldots \ , x_{m-1}, y_{m-1}, \ \ldots \ , y_0\}$ che è bitonica. Questa idea dà origine all'algoritmo merge di Batcher che porta ad una complessità totale a $O(log^2 n)$.

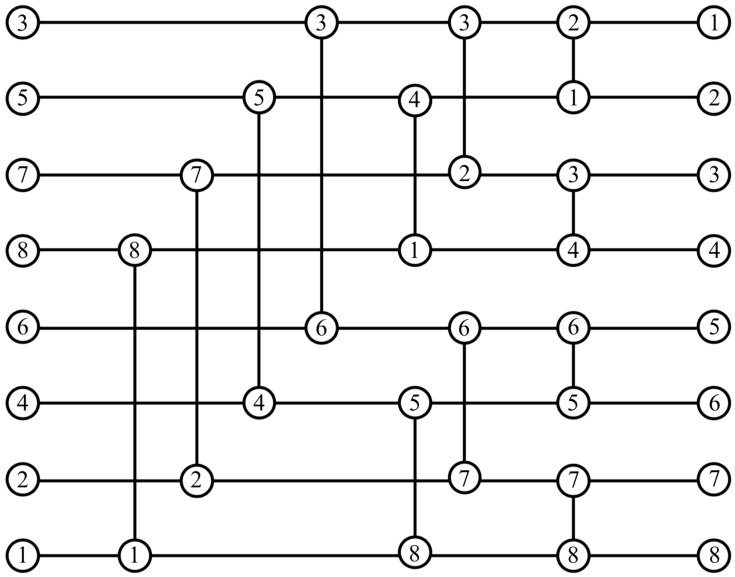


Figura 3: Bitonic sort

3 Ipercubo binario

L'ipercubo è una struttura composta da d dimensioni ed è definita come $H_d = (V,\ E)$:

- $V = \{0,\ \ldots\ 2^{d-1}\}$ con $2^d = P$ insieme dei nodi;
- $E = \sum_{j=0}^{d-1} E_j$ insieme degli archi, con $E_j = \{(x,\ c_j x) : x \in V\}$ con $j = 0,\ \ldots,\ d-1$.

Questa struttura ha la caratteristica di avere ogni processore rappresentato da un nodo che viene etichettato con un valore binario. Ogni coppia di nodi collegati da un arco differisce per un solo bit. Se consideriamo l'ipercubo binario di dimensione 3 possiamo quindi identificare i seguenti insiemi:

- E_0 insieme dei lati che collegano vertici che differiscono tra loro solamente per il bit meno significativo;
- E_1 insieme dei lati che collegano vertici che differiscono tra loro solamente per il bit nella posizione centrale;
- E_2 insieme dei lati che collegano vertici che differiscono tra loro solamente per il bit più significativo.

Questi insiemi hanno la caratteristica che i lati della stessa dimensioni sono paralleli tra loro.
Per quanto riguarda i nodi, avendo d dimensioni, ognuno avrà esattamente d

archi collegati ad esso e quindi d vicini.
Sono presenti due paradigmi per l'ipercubo:

- Ascendente: le dimensioni dell'ipercubo vengono analizzate da $E_0,\ E_1,\ \ldots,\ E_{d-1}$;
- Discendente: le dimensioni dell'ipercubo vengono analizzate da $E_{d-1},\ E_{d-2},\ \ldots,\ E_0$.

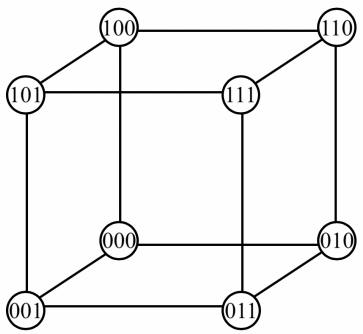


Figura 4: Ipercubo binario di dimensione 3

L'ipercubo, essendo molto strutturato, permette una buona realizzazione del Bitonic sort.

4 Implementazione - Bitonic sort su ipercubo

Il Bitonic sort su ipercubo effettua un accoppiamento delle righe consecutive facendo il merging di sequenze di lunghezza 1. Non tutte le dimensioni però vengono usate allo stesso modo, per esempio, E_0 viene usata d volte, mentre la dimensione E_{d-1} viene usata una sola volta. Questo è dovuto al fatto che la fase di sort è un'algoritmo ascendente e quindi attiva le dimensioni partendo E_0 fino a E_{d-1}.

5 Algoritmo sviluppato

Vogliamo ordinare una sequenza di N elementi, utilizzado P processori (P potenza di 2) in parallelo. Per fare ciò l'algoritmo sviluppato segue le seguenti fasi:

1. Ogni processore riceve $\frac{N}{P}$ elementi;
2. Ogni processore odina i propri elementi utilizzando l'algoritmo Mergesort;
3. Viene attivata una dimensione alla volta;
4. Ogni volta che una dimensione viene attivata i nodi dell'ipercubo si scambiano l'insieme dei valori precedentemente ordinati;

5. Viene eseguita la fase di merge tra le due sequenze ricevute nella fase precedente;

6. Questa sequenza verrà divisa a metà, la parte con i valori più bassi andrà ad un processore, mentre l'altra ad un altro;

7. Nell'ultima fase dell'algoritmo, quando tutte le dimensioni saranno attive, i nodi si comunicheranno le sequenze ordinate partendo da E_0 fino a E_{d-1}.

Nel caso in cui N non sia multiplo intero di P, è possibile aggiungere valori fittizi per fare in modo che la divisione degli elementi tra i processori sia intera.

6 Complessità dell'algoritmo

Sapendo che N è un multiplo intero di P ed essendo $P = 2^d$, d intero positivo, otteniamo $d = log_2 P$, per tale ragione

$$T_{ASC}(P) = T_{DESC}(P) = O(d) = O(log_2 P)$$

Per come è stato costruito l'algoritmo la prima fase necessita di un Mergesort di $\frac{N}{P}$ elementi:

$$T_{MERGE-SORT}(N, P) = O\left(\frac{N}{P} log\left(\frac{N}{P}\right)\right)$$

La fase di sorting utilizza un paradigma ascendente attivando quindi una dimensione alla volta in ordine crescente, mentre la fase di merge utilizza un paradigma discendente, attivando le dimensioni in senso opposto.
Come precedentemente spiegato, la dimensione E_0 viene usata d volte, mentre la dimensione E_{d-1} viene usata una sola volta.
L'attivazione di una dimensione è $O(\frac{N}{P})$ perchè ogni nodo deve effettuare l'ordinamento sugli elementi ricevuti, sfruttando il fatto che questi siano già ordinati nelle rispettive sequenze.
Sapendo che il numero totale di dimensioni attivate è:

$$O\left(\sum_{i=1}^{d} i\right) = O\left(\frac{d \cdot (d+1)}{2}\right) = O(d^2) = O(log^2 P)$$

il tempo di esecuzione totale del nostro algoritmo è:

$$T_{BITONIC-SORT}(N, P) = O\left(\frac{N}{P} log\left(\frac{N}{P}\right) + \frac{N}{P} log^2 P\right)$$

Si può notare che nel caso in cui N = P il tempo di esecuzione diventa proprio $O(log^2 N)$.

7 Risultati

Sono stati effettuati diversi test per verificare l'andamento temporale dell'ordinamento in funzione del numero di processori utilizzati. Si è deciso di considerare sequenze di lunghezza $N = 2^k$ con $k = 5,\ 6,\ \ldots,\ 27$ e un numero di processori $P = 1,\ 2,\ 4,\ 8,\ 16,\ 32$. In ogni test è stato calcolato il tempo di esecuzione trascurando la fase di lettura del file e di scrittura dei risultati. Ogni test è stato eseguito 5 volte per ammortizzare problemi estranei all'algoritmo che potevano influenzare il tempo di esecuzione e infine sono state calcolate le medie. I tempi medi ottenuti da ogni test sono riportati nella tabella seguente.

Elementi	**Processori**					
	1	**2**	**4**	**8**	**16**	**32**
32	0,00001	0,00003	0,00008	0,00022	0,52480	1,66282
64	0,00002	0,00005	0,00010	0,00111	0,53442	1,56059
128	0,00005	0,00007	0,00057	0,00023	0,46574	1,81272
256	0,00010	0,00011	0,00016	0,01327	0,70557	1,06002
512	0,00021	0,00024	0,00022	0,00035	0,59788	1,11905
1024	0,00046	0,00040	0,00045	0,01109	0,57899	1,18590
2048	0,00095	0,00080	0,00078	0,00083	0,36317	1,06053
4096	0,00219	0,00164	0,00136	0,02242	0,59088	0,87934
8192	0,00406	0,00285	0,00286	0,00238	0,51118	0,95628
16384	0,00991	0,00647	0,00542	0,00614	0,67382	0,89055
32768	0,01770	0,01241	0,01415	0,00887	0,72823	0,73615
65536	0,03913	0,02522	0,02124	0,01756	0,52723	1,25728
131072	0,07988	0,05734	0,04467	0,09431	0,81300	0,92852
262144	0,16790	0,11181	0,09086	0,76069	1,06317	0,91835
524288	0,35235	0,22576	0,20842	1,77501	1,67268	1,40836
1048576	0,73674	0,54094	0,58220	1,72624	1,86822	2,22338
2097152	1,53242	1,09689	1,01177	3,04734	2,93726	2,23640
4194304	3,81145	2,70437	3,70777	3,55917	3,77095	3,03442
8388608	11,53158	8,42322	6,41186	5,82234	5,11824	3,93710
16777216	24,66524	16,31734	13,09065	9,41838	9,54665	7,01238
33554432	22,12311	22,15827	22,22092	18,61637	14,01174	12,22581
67108864	29,89823	29,88745	29,90266	29,89874	19,96045	15,99531
134217728	46,72689	46,72851	46,75351	46,73847	46,78722	19,83512

Tabella 1: Tempi di esecuzione

Per una comprensione più immediata e semplificata dei risultati si è deciso di riportare dei grafici che rappresentino l'andamento del tempo di esecuzione in funzione del numero di elementi da ordinare. Dal grafico riportato in Figura 5 notiamo che sequenze più lunghe vengono ordinate in tempo maggiore. Per quanto riguarda l'andamento temporale nel caso in cui il numero di processori ammonti a 32 si può notare che se gli elementi da ordinare sono pochi le prestazioni peggiorano.

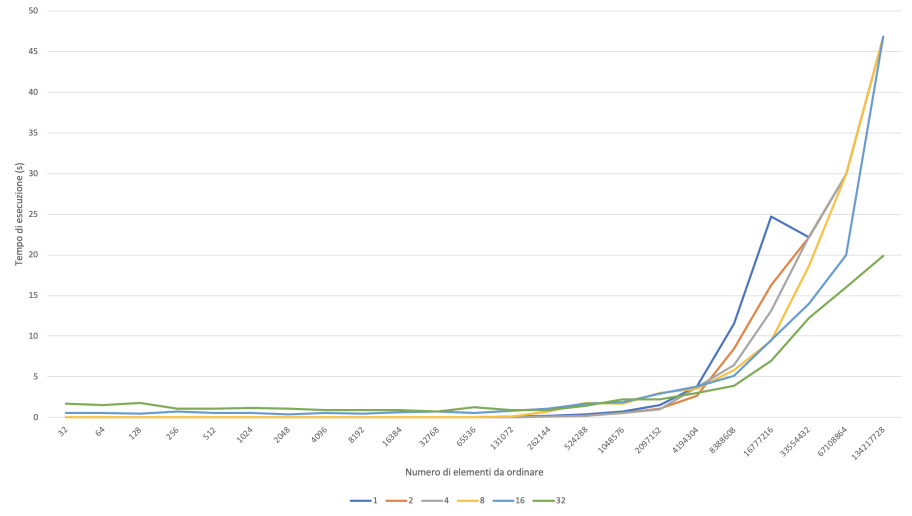


Figura 5: Prestazioni algoritmo in funzione dell'input

Si riporta inoltre un grafico in Figura 6 che mostra il tempo di esecuzione dell'algoritmo evidenziandolo in base alla dimensione dei dati di input, diviso per numero di processori utilizzati.

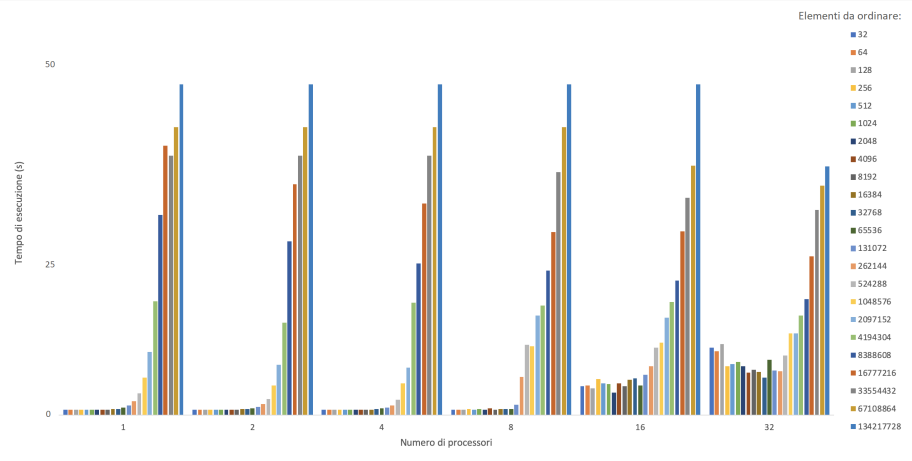


Figura 6: Prestazioni algoritmo a parità di processori

Anche in questo grafico è possibile notare che nel caso vengano utilizzati pochi processori, si ha un peggioramento delle prestazioni solamente all'aumentare della taglia dei dati in input. Quando invece si utilizza un numero maggiore di processori il peggioramento, anche se meno accentuato, si nota anche per gli input di taglia più bassa.

Viene inoltre riportato il fattore di speedup ottenuto dall'utilizzo di un numero maggiore di processori, che viene definito come

$$S(p) = \frac{T(1)}{T(p)},\ 1 < p \leq P$$

dove $T(p)$ è il tempo di esecuzione con p processori. Il valore p è strettamente maggiore di 1 in quanto $S(1)$ ha valore pari a 1.

Numero elementi	**Processori**				
	2	**4**	**8**	**16**	**32**
32	0,41	0,40	0,38	0,00	0,32
64	0,48	0,49	0,10	0,00	0,34
128	0,75	0,13	2,49	0,00	0,26
256	0,91	0,69	0,01	0,02	0,67
512	0,86	1,09	0,65	0,00	0,53
1024	1,15	0,89	0,04	0,02	0,49
2048	1,20	1,02	0,93	0,00	0,34
4096	1,34	1,21	0,06	0,04	0,67
8192	1,42	1,00	1,20	0,00	0,53
16384	1,53	1,19	0,88	0,01	0,76
32768	1,43	0,88	1,60	0,01	0,99
65536	1,55	1,19	1,21	0,03	0,42
131072	1,39	1,28	0,47	0,12	0,88
262144	1,50	1,23	0,12	0,72	1,16
524288	1,56	1,08	0,12	1,06	1,19
1048576	1,36	0,93	0,34	0,92	0,84
2097152	1,40	1,08	0,33	1,04	1,31
4194304	1,41	0,73	1,04	0,94	1,24
8388608	1,37	1,31	1,10	1,14	1,30
16777216	1,51	1,25	1,39	0,99	1,36
33554432	1,00	1,00	1,19	1,33	1,15
67108864	1,00	1,00	1,00	1,50	1,25
134217728	1,00	1,00	1,00	1,00	2,36

Tabella 2: Speedup

Dalla Tabella 2 possiamo notare che nei casi in cui la taglia dell'input è elevata, l'utilizzo di più processori non porta vantaggi, in quanto prevale la fase di merge rispetto alla fase di ordinamento. Con taglie piccole invece, vediamo che risulta poco efficiente utilizzare un numero elevato di processori in quanto il tempo di comunicazione e scambio di dati tra i processori prevale sul tempo di ordinamento.
Come riportato dai risultati, il valore massimo di speedup raggiunto si trova nel caso in cui vengano utilizzati 32 processori e la taglia dell'input è massima tra quelle considerate. Infatti andando ad analizzare il tempo di esecuzione con 32 processori e taglia dell'input massima, notiamo che il tempo impiegato è poco meno della metà rispetto a quello utilizzato negli altri casi.

8 Conclusioni

In questa relazione è stato affrontato il problema dell'ordinamento, si è mostrato come parallelizzare la risoluzione del problema andando ad esaminare i problemi principali degli algoritmi sequenziali. Si è ripreso il lavoro di Batcher e si sono esposte alcune definizioni che hanno portato all'algoritmo Bitonic sort. Da qui si è deciso di analizzare il caso in cui si abbaino $N = 2^k$ elementi da ordinare con $P = 2^j$ processori a disposizione con il vincolo che $N \geq P$. Dai test effettuati si è potuto concludere che non necessariamente un numero maggiore di processori porti a prestazioni migliori. Si è potuto verificare che effettivamente l'andamento dei tempi d'esecuzione sia circa

$$T_{BITONIC-SORT}(N, P) = O\left(\frac{N}{P} log\left(\frac{N}{P}\right) + \frac{N}{P} log^2 P\right)$$

Data questa equazione è possibile notare che dato un numero N di elementi da ordinare, il numero di processori P che minimizza il tempo di esecuzione è calcolabile derivando rispetto P:

$$\frac{\partial T_{BS}}{\partial P} = -\frac{N}{P^2}\left(log\left(\frac{N}{P}\right) + log\, e + log^2 P - 2log\, P \cdot log\, e\right)$$

Dato quindi un N fissato è possibile porre la derivata uguale a zero ed individuare un punto di minimo di T_{BS} che permetterà di individuare il numero di processori che vanno ad ottimizzare l'esecuzione dell'algoritmo con la sequenza in input di cardinalità N. Dai dati raccolti in fase di test possiamo affermare che con N basso, un numero alto di processori risulta meno efficiente in quanto l'algoritmo tende ad avere un comportamento simile ad uno sequenziale e la potenza di calcolo dei processori non verrebbe completamente sfruttata. Per quanto riguarda istanze con N grande possiamo notare che un maggior numero di processori porta performance migliori.
Andando a riguardare la derivata precedente e fissando $N = 2^k$ con $k = 5, 6, \ldots, 23$ è possibile notare che il punto di minimo tende ad avanzare e quindi significa che più processori performano meglio su sequenze più lunghe. Tuttavia se andassimo a derivare T_{BS} rispetto ad $N = kP$ noteremmo il seguente risultato:

$$\frac{\partial T_{BS}}{\partial N} = \frac{1}{P}\left(log\left(\frac{N}{P}\right) + log\, e + log^2 P\right) = \frac{1}{P}\left(log\, k + log\, e + log^2 P\right)$$

Dato quindi un numero di processori P fissato, le performance peggiorano all'aumentare di k. Questa analisi è coerente con la precedente e con i risultati dei nostri test quindi possiamo concludere che se volessimo ottimizzare al massimo l'algoritmo, la scelta del numero di processori deve essere fatta in funzione della lunghezza della sequenza da ordinare.

Riferimenti bibliografici

[1] Batcher, *Sorting Networks and their Applications*, AFIPS Spring Joint Comput, 1968.

[2] Smith, *The Design and Analysis of Parallel Algorithms*, Oxford University Press, 1993.

[3] Blelloch, Leiserson, Maggs, Plaxton, Smith, Zagha, *A Comparison of Sorting Algorithms for the Connection Machine CM-2*, ACM, 1991.