Università degli Studi di Padova

Dipartimento di Ingegneria Informatica

Relazione di progetto per il corso di Calcolo Parallelo

Algoritmo di Bitonic Sort Parallelo: Analisi delle Prestazioni

A.A. 2018/19

Miani Eleonora - M. 1206908

Serratore Davide - M.1207660

Vesco Omar - M.1197699

Descrizione del Problema

Si è voluto analizzare il problema dell'ordinamento di una sequenza di numeri, una questione a prima vista banale, ma di importanza fondamentale in ambito informatico: per questo motivo esistono innumerevoli algoritmi e approcci alla sua risoluzione, nonché una vasta letteratura che esplora diverse possibili ottimizzazioni. Per le stesse ragioni è anche un problema che ben si presta ad un'analisi dell'implementazione e delle sue prestazioni nell'ambito della programmazione parallela: in particolare si è scelto di studiare una implementazione iterativa e parallela del *bitonic sort algorithm*. Sono inoltre state sviluppate una implementazione iterativa ed una ricorsiva dello stesso algoritmo single-core, allo scopo di gestire un ordinamento locale all'interno di ciascun processore, nel caso in cui il numero di elementi da ordinare sia maggiore del numero di processori a disposizione..

I dati da ordinare vengono passati all'algoritmo attraverso un file di testo, specificato come argomento all'esecuzione. Bisognerà inoltre fornire il numero di elementi contenuti nel file e il nome del file in cui l'algoritmo dovrà produrre la sequenza di elementi ordinati. È stata fatta l'ipotesi semplificativa che il numero di elementi da ordinare sia una potenza di due, in modo tale da poter distribuire equamente i dati tra i processori utilizzati; questa condizione viene verificata all'inizio dell'esecuzione, e in caso non sia rispettata causa la terminazione del programma.

Descrizione dell'Algoritmo parallelo

L'algoritmo nella sua versione parallela si può suddividere in 5 fasi:

1. Lettura e controllo dell'input da file e suddivisione dei dati nei vari processori
2. Ordinamento delle sequenze locali all'interno di ciascun processore
3. Scambio tra i processori delle sequenze secondo la logica Bitonic
4. Merge delle sottosequenze
5. Salvataggio della sequenza ordinata all'interno di un file di testo

1. Lettura e controllo dell'input da file e suddivisione dei dati nei vari processori

Nella prima parte dell'algoritmo il processore di rank 0 - denominato *master process* - si occupa di analizzare l'input fornito dall'utente, verificando se sono presenti tutti i parametri necessari, e se il numero di elementi da ordinare è maggiore del numero di processori richiesti ed è un potenza di due. In caso affermativo equi distribuisce i dati a tutti i processori disponibili; in caso contrario l'esecuzione del programma viene terminata.

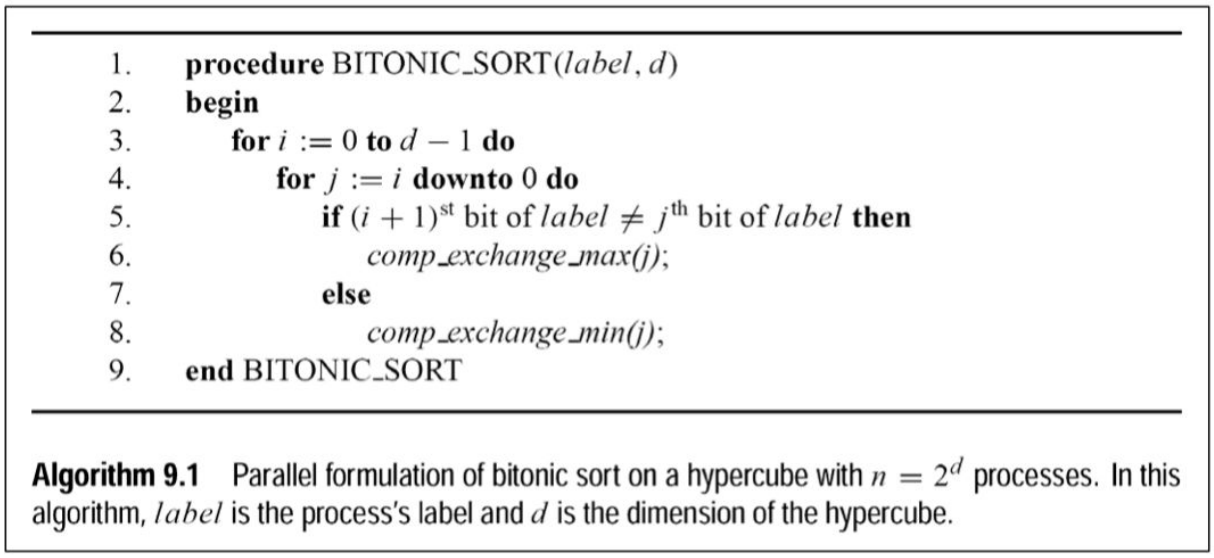
2. Ordinamento delle sequenze locali all'interno di ciascun processore

L'algoritmo proposto prevede che il numero di elementi da ordinare possa essere superiore al numero di processori disponibili. Per questo motivo nel momento in cui i dati vengono distribuiti a ciascun processore, ognuno di questi esegue un ordinamento locale prima di effettuare le comunicazioni con gli altri processori. Tuttavia - come emergerà dall'analisi delle prestazioni successiva - l'algoritmo bitonico risulta poco consono a questo scopo, in quanto presenta buone prestazioni solo nel caso parallelo: si è scelto quindi di utilizzare l'algoritmo quicksort, presente nella libreria standard, per effettuare questo ordinamento.

3. Scambio tra i processori delle sequenze secondo la logica Bitonic

Dopo che le sequenze locali sono state ordinate all'interno dei singoli processori, viene eseguito il Bitonic Sort vero e proprio. L'aspetto chiave di questo algoritmo è lo scambio e il confronto opportuno di sequenze bitoniche tra diversi processori, che permette di effettuare il sorting e il merging in più stadi. La scelta dei processori da far comunicare è stata fatta basandosi sul modello dell'ipercubo. L'i-esima iterazione del ciclo rappresenta uno stadio dell'algoritmo che consiste nell'esecuzione di un paradigma discendente di più dimensioni, a partire dall'i-esima. L'esecuzione di una j-esima dimensione corrisponde alla comunicazione dei processori che fanno parte della stessa, ovvero processori che differiscono per il j-esimo bit.

Lo pseudocodice dell'algoritmo di bitonic sorting per un ipercubo [3] è riportato in figura.



```
1.  procedure BITONIC_SORT(label, d)
2.  begin
3.      for i := 0 to d − 1 do
4.          for j := i downto 0 do
5.              if (i + 1)st bit of label ≠ jth bit of label then
6.                  comp_exchange_max(j);
7.              else
8.                  comp_exchange_min(j);
9.  end BITONIC_SORT
```

Algorithm 9.1 Parallel formulation of bitonic sort on a hypercube with $n = 2^d$ processes. In this algorithm, $label$ is the process's label and d is the dimension of the hypercube.

Fig. 1: Pseudocodice dell'algoritmo Bitonic Sort su ipercubo

L'algoritmo utilizza le funzioni *comp_exchange_max(i)* e *comp_exchange_min(i)*. Quest'ultime vengono richiamate dalle varie coppie di processori in accordo con il paradigma bitonico, eseguono un confronto tra le due sequenze e tengono rispettivamente gli elementi massimi o minimi.

4. Merge delle sottosequenze, sequenza finale

A questo punto tutti i processori, in ordine crescente e a partire dal master, hanno a disposizione delle sequenze ordinate, corrispondenti ai blocchi contigui che andranno a formare la sequenza finale. Per ottenere quest'ultima le varie sottosequenze vengono recuperate dai processori e unite. La sequenza ordinata finale viene resa disponibile al processo master.

5. Salvataggio della sequenza ordinata all'interno di un file di testo

Il *master process* si occupa di salvare la sequenza ordinata all'interno di un file di testo, specificato dall'utente come argomento dell'esecuzione, e il programma termina.

Analisi Sperimentale delle Prestazioni

Per poter effettuare un'analisi efficace delle prestazioni dell'algoritmo sono state sviluppate due versioni del bitonic sort che utilizzano un solo processore: una utilizza un approccio "Divide & Conquer" come proposto nell'articolo [1], l'altra è di tipo iterativo, e applica la logica di comunicazione su ipercubo - su cui si basa l'algoritmo parallelo - ai confronti eseguiti tra gli elementi dell'array da ordinare. L'intento primario di quest'ultima versione era quella di eliminare il peso della ricorsione della prima versione in modo da ottenere un buon algoritmo per poter effettuare un sorting locale nella versione parallela, come discusso in precedenza. Tuttavia i risultati presenti nel grafico in figura [2] mostrano che la versione iterativa è più veloce ad ordinare i numeri, ma solo se il codice viene compilato senza ottimizzazioni.

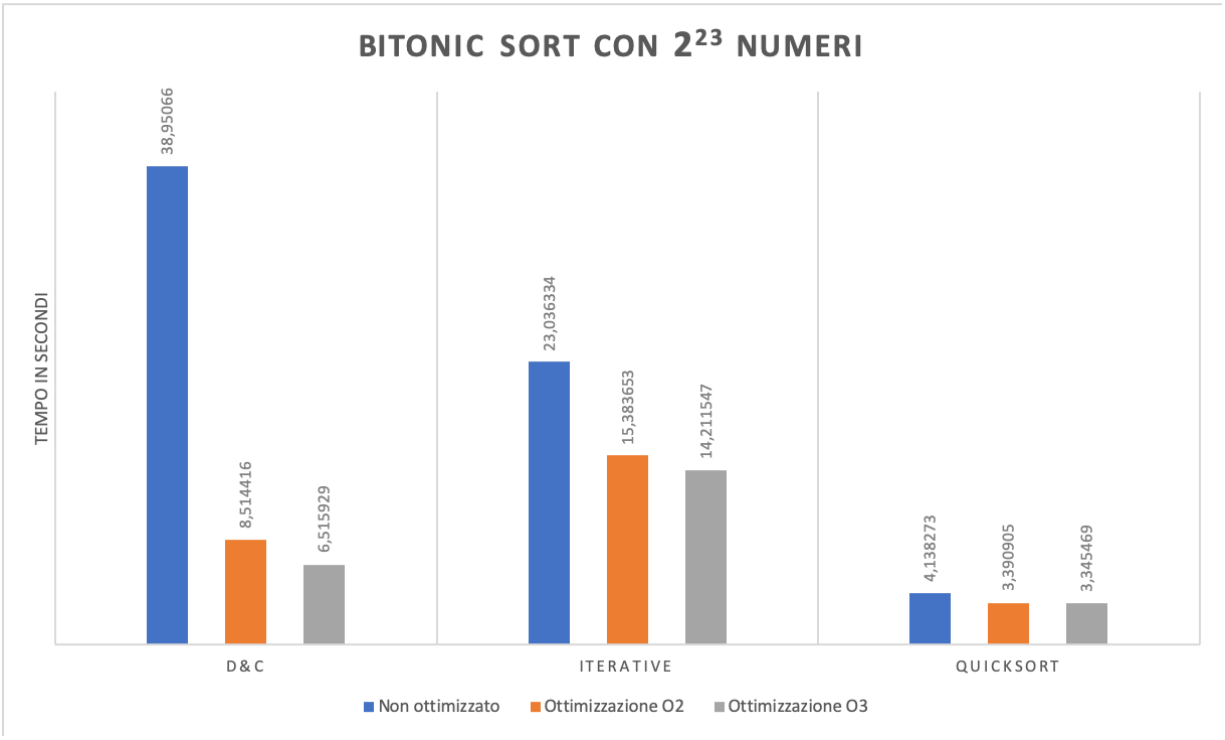


Fig. 2: Confronto tra implementazioni single-core di Bitonic Sort con approccio Divide & Conquer, Bitonic Sort iterativo e QuickSort.

La prima versione risulta invece più efficiente quando il codice viene compilato con le ottimizzazioni proposte dal compilatore. Nonostante questi risultati, l'algoritmo QuickSort ha comunque prestazioni migliori nell'ordinamento a singolo processore, motivo per cui nella versione parallela è stato scelto di utilizzare quest'ultimo per effettuare l'ordinamento locale necessario quando il numero di elementi da ordinare è maggiore del numero di processori a disposizione.

Le misurazioni dell'algoritmo parallelo sono state effettuate utilizzando 1, 2, 4, 8, 16 e 32 processori, confrontandoli tra loro mantenendo costante la taglia di input. La versione con 1 processore esegue localmente l'algoritmo di QuickSort, mentre i successivi test includono, oltre ad un'operazione di sorting locale di una taglia minore di numeri, tutte le comunicazioni derivanti dall'algoritmo bitonico. I risultati ottenuti ci mostrano che aumentando il numero di processori diminuiamo il tempo necessario per ordinare la lista di numeri, inoltre otteniamo prestazioni migliori ottimizzando il codice in compilazione.

# Processori	Tempi di esecuzione Bitonic Sort (s)		
	Non Ottimizzato	Ottimizzazione O2	Ottimizzazione O3
1	4,138273	3,390905	3,345469
2	2,273188	1,899827	2,086939
4	1,338127	1,037024	1,041295
8	0,883538	0,649733	0,712543
16	0,599453	0,490199	0,474222
32	0,391105	0,33216	0,336174

Tabella 1: Tempo di esecuzione del Bitonic Sort per ordinare 2^{23} elementi, al variare del numero di processori

Si può notare che fino a 8 processori il tempo necessario per eseguire l'ordinamento migliora di quasi il doppio, mentre eseguendo dei test con 16 e 32 processori il tempo migliora ma non ci sono grandi differenze. Questo probabilmente è dovuto al numero di comunicazioni che l'algoritmo deve eseguire nel corso dell'esecuzione, che aumentano all'aumentare dei processori utilizzati.

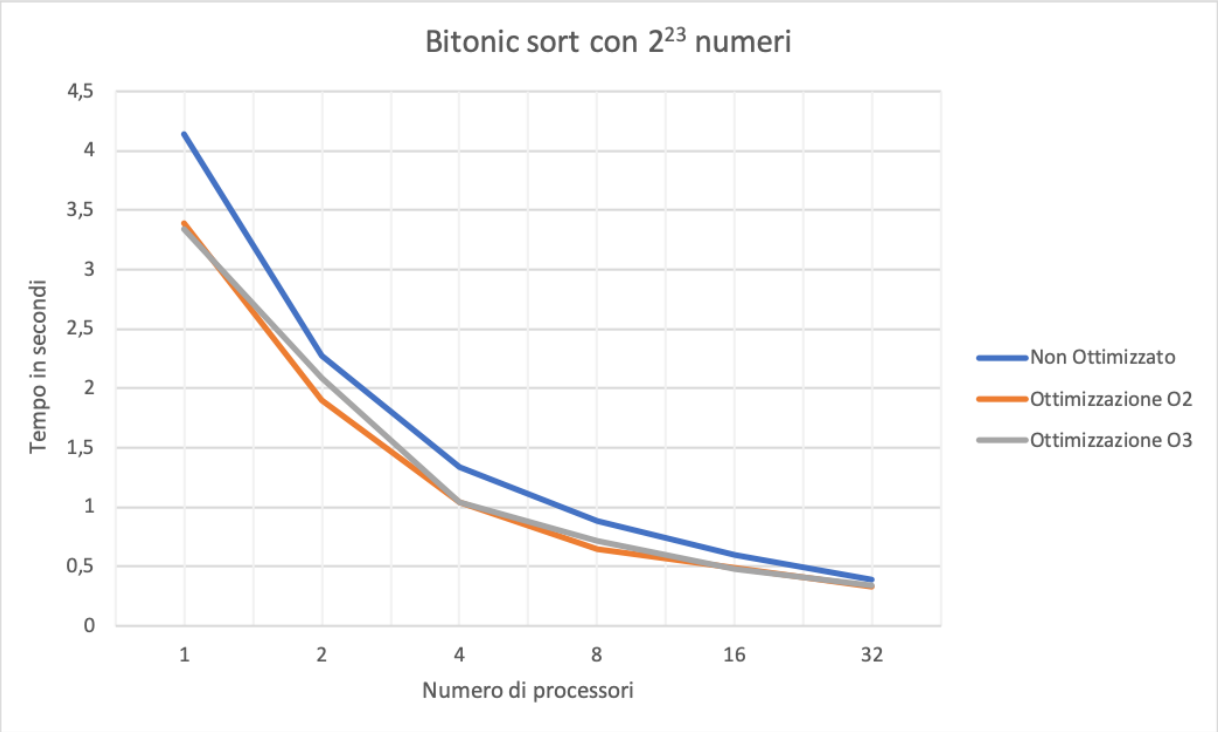


Fig. 3: Tempo di esecuzione del Bitonic Sort per ordinare 2^{23} elementi, al variare del numero di processori

In seguito abbiamo analizzato il comportamento dell'algoritmo al variare della taglia di input: sono stati ordinati 2^6 = 64, 2^{10} = 1024, 2^{13} = 8192 e 2^{15} = 32768 elementi. Per questa analisi l'algoritmo è stato eseguito utilizzando 8 processori. Dai risultati ottenuti e dalla figura [5] possiamo notare che il tempo di ordinamento cresce linearmente con la taglia dell'input. Dalla figura [4], inoltre, si può notare come l'ottimizzazione O2 porti a risultati migliori dell'ottimizzazione O3.

# Elementi	Tempi di esecuzione Bitonic Sort (s)		
	Non Ottimizzato	Ottimizzazione O2	Ottimizzazione O3
2^{15}	0,003521	0,002528	0,002585
2^{13}	0,000841	0,000663	0,000738
2^{10}	0,000132	0,000117	0,000112
2^{6}	0,000062	0,000056	0,000059

Tabella 2: Tempo di esecuzione del Bitonic Sort con 8 processori, al variare della taglia dell'input

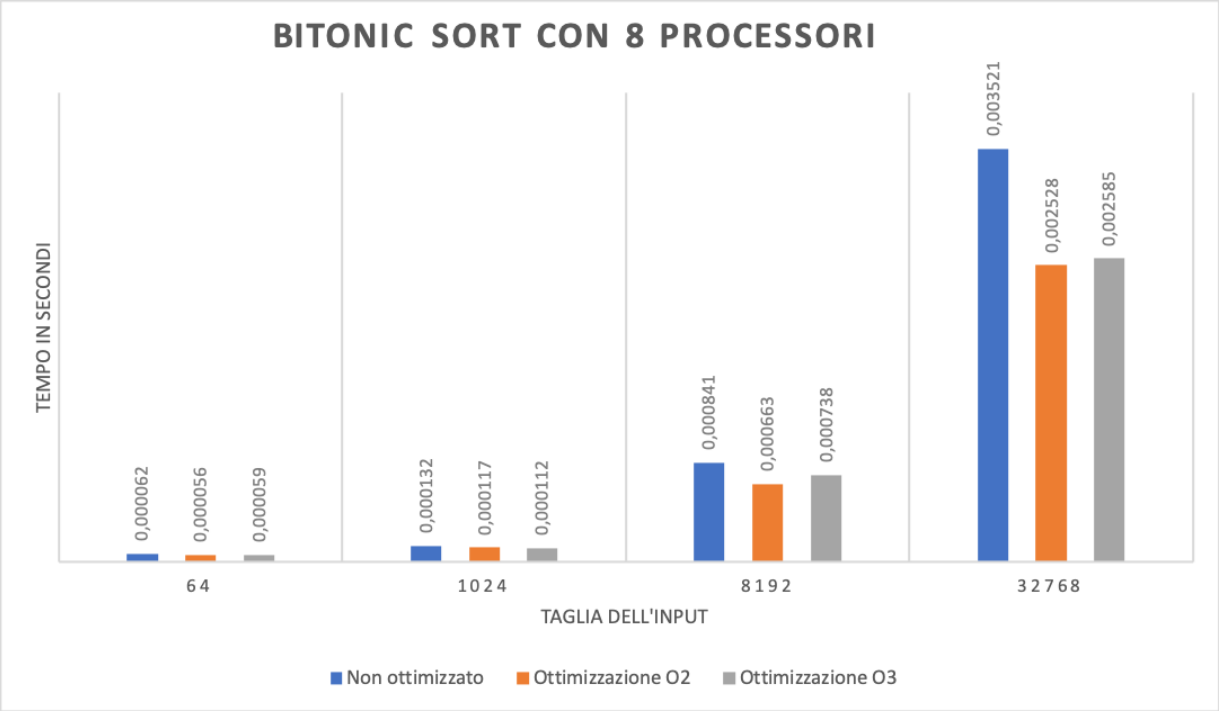


Fig 4: Tempo di esecuzione del Bitonic Sort con 8 processori, al variare della taglia dell'input

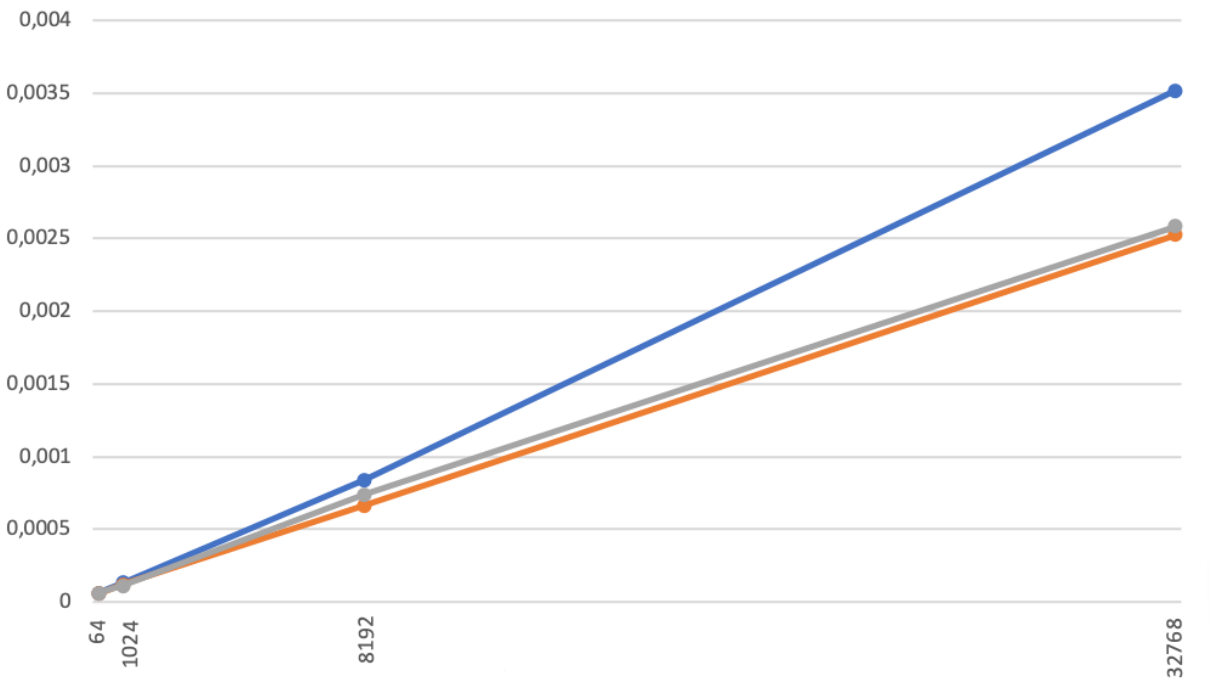


Fig. 5: Tempo di esecuzione del Bitonic Sort con 8 processori,
al variare della taglia dell'input

Mettendo a confronto i tempi di esecuzione di figura [4] e figura [6] possiamo notare che, eseguendo con taglie di input molto piccole, la differenza in termini temporali tra l'algoritmo parallelo e quello single core è molto bassa. Con la taglia di input pari a 64 gli algoritmi "Divide & Conquer" e "Iterativo" proposti risultano più rapidi, sintomo del fatto che nella versione parallela i tempi di comunicazione richiesti per scambiare i dati appesantiscono i tempi di esecuzione. Quando la taglia aumenta, la versione parallela nel complessivo non risente allo stesso modo delle comunicazioni, e ottiene prestazioni migliori.

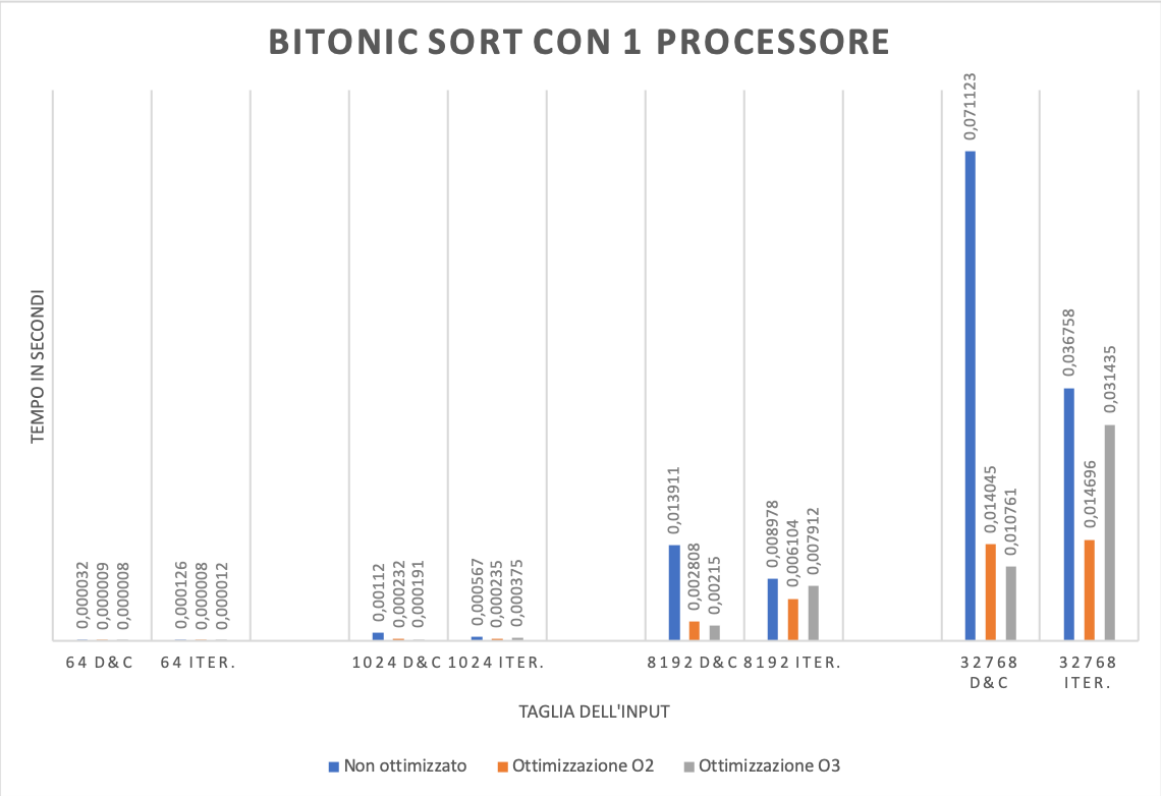


Fig. 6: Confronto dei tempi di esecuzione con un solo processore con algoritmo Bitonic Sort Divide & Conquer e iterativo.

# Processori	Fattore di Speed Up		
	Non Ottimizzato	Ottimizzazione O2	Ottimizzazione O3
1	1	1	1
2	1,820471	1,6954525	1,60305
4	3,092586	3,269842	3,212796
8	4,683752	5,21892	4,695111
16	6,903415	6,917404	7,054647
32	10,580976	10,208649	9,951599

Tabella 3: Fattori di Speed Up

La tabella [3] riporta i fattori di Speed Up calcolati sui tempi di esecuzione con 2^{23} elementi di input, rispetto al tempo di esecuzione con un singolo processore. In generale, all'aumentare del numero di processori, migliorano le prestazioni (fattore di Speed Up superiore); inoltre si può notare come le varie ottimizzazioni presentino dei fattori di Speed Up comparabili. Ciò significa che, nonostante i tempi di computazione ottenibili con le ottimizzazioni del compilatore siano più brevi, aumentare il numero di processori comporta un miglioramento delle prestazioni netto.

Conclusioni

Dall'analisi dei tempi di esecuzione al variare della taglia di input si è potuto verificare che l'algoritmo scala bene anche con un numero di elementi da ordinare elevato, mentre confrontando i risultati al variare dei processori utilizzati - e mantenendo a 2^{23} il numero di elementi da ordinare - è emerso che il miglioramento delle prestazioni non continua linearmente. Ciò accade perché il peso delle comunicazioni - che diventano sempre più numerose all'aumentare del numero di processori - incide maggiormente della riduzione del numero di elementi da ordinare localmente all'interno di ciascuno.
È stato inoltre riscontrato che per taglie di input piccole (ad esempio 64 elementi) non ha senso parallelizzare l'algoritmo di ordinamento, in quanto ancora una volta le comunicazioni pesano in modo eccessivo sul tempo di esecuzione, e le prestazioni di un algoritmo single core risultano migliori. Infine, in generale, si è potuto verificare che le ottimizzazioni del compilatore (flag -O2 e -O3) portano ad un notevole miglioramento delle prestazioni.

All'interno di questo progetto si è posta l'attenzione principalmente sull'efficienza dell'algoritmo di Bitonic Sort vero e proprio, ma analizzando sperimentalmente i tempi computazionali dell'algoritmo nel suo complesso si è verificato che l'impatto delle operazioni di lettura dell'input e di scrittura dell'output acquisiscono un peso sempre maggiore all'aumentare della taglia dell'input, fino anche a vanificare ulteriori ottimizzazioni dell'algoritmo bitonico. In futuro sarebbe quindi interessante analizzare ed implementare una gestione parallela della lettura dell'input, che al momento viene gestita da un singolo *master process*, che si occupa di distribuire poi i dati agli altri processori.

Riferimenti Bibliografici

[1] G. Bilardi, F. P. Preparata, (1984) An Architecture for Bitonic Sorting with Optimal VLSI Performance, IEEE Transactions On Computers. VOL c-33, NO. 7.

[2] M. Jain, S. Kumar, V.K Patle, (2015) Bitonic Sorting Algorithm: A Review, International Journal of Computer Applications (0975 – 8887) Volume 113 – No. 13.

[3] A. Grama, A. Gupta, G. Karypis, V. Kumar, (2003) Introduction to Parallel Computing, Addison Wesley.