UNIVERSITÀ DEGLI STUDI DI PISA

Bucket Sort con Skandium su Architettura Multicore

Marco Ponza

1 SCELTE PROGETTUALI

1.1 SCELTA DELLO SKELETON

Data la definizione del problema, possiamo fin da subito vedere che l'idea del Bucket Sort si combina molto bene con la map. Infatti, le varie fasi dell'algoritmo, possono essere strutturate dalla map come segue:

Splitting L'emitter della map effettua la fase di suddivisione dell'array in bucket, in modo che ciascuno di essi rispetti la proprietà definita nell'algoritmo:

Proprietà 1. Ogni bucket B_i contiene tutti gli elementi che sono più piccoli degli elementi contenuti nel bucket B_J: $\forall\, j > i$ ma che sono, però, più grandi degli elementi contenuti nei bucket B_k: $\forall\, k < i$.

Sorting I bucket vengono ordinati parallelamente dai worker.

Merging Il collector, una volta ricevuti i bucket ordinati, può creare l'array complessivamente ordinato semplicemente giustapponendo i bucket ricevuti visto che la Proprietà 1 continua a valere anche con i bucket ordinati.

Una seconda soluzione che si potrebbe pensare per implementare il Bucket Sort, è quella di utilizzare una farm che, a differenza della map, riceverebbe in ingresso direttamente i bucket e non l'array. Questa soluzione presenterebbe, quindi, da una parte, lo svantaggio di dover fornire allo skeleton già i bucket (e non direttamente l'array da ordinare) e dall'altra

parte il vantaggio, rispetto alla soluzione descritta precedentemente, di possedere una maggiore efficienza, dovuta al fatto che sarebbe possibile iniziare l'ordinamento di un bucket appena questo verrebbe calcolato, senza dover attendere di calcolare anche i successivi (del rispettivo array) prima di poter iniziare la fase di ordinamento: questo, infatti, nella map non accade, visto che, l'ordinamento di un bucket, può iniziare solamente al termine della fase di splitting e, quindi, solamente quando sono già stati calcolati tutti i bucket da inviare ai rispettivi worker per essere ordinati[1].

A discapito dell'efficienza, è stato alla fine deciso di realizzare comunque la soluzione che utilizza la map:

- La map si sposa, concettualmente, molto bene con le fasi dell'algoritmo Bucket Sort;
- Essendo il Bucket Sort interamente implementato dallo skeleton scelto, viene offerta una maggiore riusabilità del codice di quella che si avrebbe andando ad utilizzare la farm. In questo modo, se in futuro si volesse usare il Bucket Sort (nella sua versione parallela) in progetti diversi, basterebbe semplicemente riutilizzare lo skeleton che lo implementa.

1.2 Numero di Bucket

Il numero di bucket è un parametro dell'applicazione e rappresenta il numero di bucket in cui ogni array dello stream viene suddiviso. Esso è, inoltre, il limite superiore al numero di worker che possono ordinare i bucket di uno stesso array.

1.3 Grado di Parallelismo

Il grado di parallelismo è un parametro dell'applicazione e rappresenta il numero massimo di istruzioni MDF che possono essere eseguite concorrentemente dall'interprete MDF.

2 Cenni sull'implementazione

2.1 Flusso d'Esecuzione

Il flusso d'esecuzione principale della soluzione che utilizza la map è il seguente:

1. Vengono effettuati dei controlli sui parametri di input ricevuti;
2. Vengono letti e allocati in memoria gli array presenti in un file. In tale file sono presenti m array da ordinare, ognuno di lunghezza n.

[1] Essendo che la fase di splitting della map di Skandium deve ritornare un array di elementi suddivisi (`http://skandium.niclabs.cl/javadoc/1.0b2/cl/niclabs/skandium/muscles/Split.html`), si è supposto che la fase svolta dai worker (nel nostro caso l'ordinamento) possa essere eseguita solo al termine della fase precedente.

3. Vengono dati in input gli array da ordinare alla map che implementa il Bucket Sort.

4. Una volta terminato l'ordinamento di tutti gli array, questi vengono scritti su un file di output, rispettando l'ordine in cui sono stati letti.

2.2 MAP

2.2.1 SPLITTING

L'emitter della map svolge la funzione di suddivisione dell'array in bucket. A seconda del numero di bucket e del grado di parallelismo richiesto, possiamo avere due casi:

1. *Il numero di bucket è minore o uguale al grado di parallelismo richiesto.* È stato deciso di eseguire una suddivisione dell'array in bucket che cerca di bilanciare il carico di lavoro che viene poi affidato ai worker. Tutti i bucket, escluso al più l'ultimo, conterranno lo stesso numero di floating-point. Questo è stato reso possibile utilizzando l'algoritmo quickselect che permette di calcolare il k-esimo elemento più piccolo di un array disordinato, mettendo tutti gli elementi minori o uguali del k-esimo elemento alla sua sinistra e quelli maggiori uguali al k-esimo elemento alla sua destra, in tempo medio $O(n)$. Facciamo un esempio.

   Esempio 1. Supponiamo che ogni bucket (affinché ci sia load balancing) debba essere grande r. Dopo aver eseguito una volta il quickselect per trovare l'$r-1$-esimo elemento, il primo bucket sarà costituito dall'$r-1$-esimo elemento e da tutti gli elementi alla sua sinistra. Il secondo bucket, invece, dopo aver eseguito il quickselect sulla porzione di array che va da r sino alla fine dell'array per trovare il $2(r-1)$-esimo elemento, sarà costituito da tutti gli elementi che vanno da r a $2(r-1)$.

   Ogni worker istanziato successivamente si occuperà poi di ordinare sequenzialmente un solo bucket

2. *Il numero di bucket è maggiore del grado di parallelismo richiesto.* È stato deciso di eseguire una suddivisione di array in bucket che cerca di effettuare load balancing come descritto nel punto precedente. In questo secondo caso, però, ogni worker dovrà ordinare (uno o) più bucket, essendo il grado di parallelismo richiesto minore del numero di bucket. I bucket calcolati verranno quindi raggruppati in sequenze di bucket, ognuna delle quali conterrà (circa) lo stesso numero di bucket contenuto nelle altre sequenze (si è cercato di bilanciare il numero di bucket contenuto in ogni sequenza di bucket). Se un bucket viene inserito in una sequenza, allora tutti gli elementi del bucket inserito sono maggiori o uguali agli elementi attualmente presenti nella sequenza in cui è stato inserito. Per ogni sequenza di bucket e per tutti i bucket presenti in una sequenza, continua quindi a valere la Proprietà 1 (per maggiori dettagli vedere la Sezione 2.3).
   Il numero di sequenze di bucket create sarà pari al grado di parallelismo richiesto.
   Ogni worker, istanziato successivamente, si occuperà di ordinare una sequenza di bucket.

Per far in modo che i worker possano ricevere sia singoli bucket che sequenze di bucket è stato deciso di mandare sempre, ad un generico worker, una sequenza di bucket. Nel caso in cui il worker debba ricevere un solo bucket, riceverà una sequenza di bucket contenente un solo bucket.

2.2.2 SORTING

Ogni worker si occupa di ordinare la sequenza di bucket ricevuta e di passarla al collector. Per verificare facilmente la scalabilità dell'applicazione è stato scelto di usare un algoritmo di ordinamento con una complessità quadratica: il Selection sort.

2.2.3 MERGING

Il collector riceve un array di sequenze di bucket e, giustapponendo i bucket presenti in ogni sequenza, crea l'array complessivamente ordinato e lo ritorna.

2.3 PROBLEMI RISCONTRATI

Durante la fase di implementazione, il problema principale è stato riscontrato durante la realizzazione di un buon load balancing nel caso in cui il numero di bucket fosse maggiore del grado di parallelismo richiesto. Una prima soluzione implementata è stata quella di assegnare a tutti i worker, tranne l'ultimo, delle sequenze di bucket contenenti $\left\lfloor \frac{nb}{pg} \right\rfloor$[2] bucket e, all'ultimo worker, una sequenza contenente i rimanenti bucket. In questo caso, il load balancing era buono, ma fino ad un certo punto. Facciamo un esempio.

Esempio 2. Prendiamo come numero di bucket 19 e grado di parallelismo 5. Con la tecnica appena descritta otterremo 4 sequenze di bucket, ciascuna contenete 3 bucket e una sequenza, l'ultima, contenete 7 bucket. Questa situazione non gode sicuramente di un buon load balancing.

Per risolvere il problema è stato deciso di fare degli aggiustamenti, dopo aver applicato il metodo di bilanciamento sopra descritto, alle sequenze di bucket create, in modo da ridurre il numero di bucket presenti nell'ultima sequenza e aumentare quello presente nelle restanti sequenze. Ciò è stato fatto sempre in modo da far rispettare la Proprietà 1 per ogni sequenza e per tutti i bucket presenti in una sequenza.

3 PERFORMANCE TEST

Dopo la fase d'implementazione, sono stati svolti dei test per verificare il comportamento della soluzione realizzata, confrontando il suo tempo di completamento con il tempo di completamento che, idealmente, dovrebbe avere e con il tempo di completamento del Bucket Sort implementato in maniera sequenziale .

[2] con nb il numero di bucket e pg il grado di parallelismo.

La latenza del *Bucket Sort Parallelo (Ideale)*, supponendo che le latenze dell'emitter e del collector siano trascurabili, è definita dalla seguente equazione:

$$L_{par}(nb)^{pg} = \frac{L_{seq}(nb)}{min(pg, nb)} \tag{3.1}$$

dove:

- nb è il numero di bucket in cui viene suddiviso ogni array;
- pg è il grado di parallelismo richiesto;
- m è il numero di elementi dello stream:
- $min(pg, nb)$ rappresenta il numero di sequenze di bucket, di uno stesso array, che verranno inviate ai worker. Ci potranno essere, quindi, al massimo $min(pg, nb)$ worker che ordinano parallelamente le sequenze di bucket di uno stesso array;
- $L_{seq}(nb)$ è la latenza del *Bucket Sort Sequenziale*.

Supponendo che i tempi di servizio delll'emitter e del collector siano trascurabili, la latenza ci rappresenta quindi anche il tempo di servizio ideale di un worker:

$$T_{S_{worker}} = L_{par}(nb)^{pg} = \frac{L_{seq}(nb)}{min(pg, nb)} \tag{3.2}$$

Il tempo di servizio del *Bucket Sort Parallelo (Ideale)*, sempre supponendo che i tempi di servizio dell'emitter e del collector siano trascurabili, è pari a:

$$T_{S_{par}}(nb)^{pg} = min(pg, nb)\frac{T_{S_{worker}}}{pg} = \frac{L_{seq}(nb)}{pg} \tag{3.3}$$

È necessario moltiplicare per $min(pg, nb)$ perché, se non lo facessimo, non otterremo il tempo medio ideale che intercorre tra l'invio di due successivi output (che sono due array interamente ordinati, costituiti dalla giustapposizione di tutti i bucket contenuti nelle sequenze di bucket), ma otterremo, invece, il tempo medio ideale che intercorre tra l'invio di due successive sequenze di bucket ordinate.
Il tempo di servizio calcolato in (3.3) vale perché, Skandium, possiede un'implementazione a macro data flow che, a differenza di una map "pura" (in cui, concettualmente, anche aumentando il grado di parallelismo, il numero di worker che in un certo istante possono ordinare i bucket è limitato superiormente dal numero di bucket stesso), permette di eseguire parallelamente un numero di worker (istruzioni MDF che rappresentano i worker) limitato superiormente dal grado di parallelismo richiesto e non dal numero di worker che possono ordinare i bucket di uno stesso array. Facciamo un esempio.

Esempio 3. Supponiamo di volere che ogni array venga suddiviso in 2 bucket e di voler usare un grado di parallelismo pari a 4: è quindi possibile che l'interprete MDF esegua parallelamente le due istruzioni MDF che ordinano i due bucket di un array e le due istruzioni MDF

che ordinano i due bucket di un altro array (supponendo che tutte e quattro le istruzioni MDF siano fireable). Avendo, quindi, un grado di parallelismo superiore al numero di bucket, la latenza non diminuisce, restando uguale a quella che si avrebbe eseguendo lo stesso programma ma con grado di parallelismo 2, ma diminuisce, invece, il tempo di servizio (e, di conseguenza, anche il tempo di completamento).

Il tempo di completamento del *Bucket Sort Parallelo (Ideale)* è, quindi, dato da:

$$T_{C_{par}}(nb)^{pg} = mT_{S_{par}}(nb)^{pg} = m\frac{L_{seq}(nb)}{pg} \tag{3.4}$$

Sono stati svolti, quindi, tre test:

1. Fissando il numero di bucket in cui suddividere ogni array ad 8, per verificare l'effettiva scalabilità dell'applicazione fino a raggiungere un grado di parallelismo pari al numero di bucket;

2. Fissando il numero di bucket in cui suddividere ogni array a 4, per verificare che, anche aumentando il grado di parallelismo ad un valore superiore al numero di bucket, l'applicazione continua comunque a scalare per via della diminuzione del tempo di servizio.

3. Fissando il numero di bucket in cui suddividere ogni array a 4, per verificare che, se il numero di elementi dello stream è pari ad 1, allora, anche aumentando il grado di parallelismo a valori superiori al numero di bucket, il tempo di completamento non cambia.

In tutti i casi, sono stati calcolati i rispettivi tempi di completamento facendo variare il grado di parallelismo con valori nell'intervallo [1; 8]. I tempi di completamento rilevati nei test non considerano la lettura e la scrittura su file degli array. Tutti i test sono stati svolti su $ottavinareale$.

3.1 8 Bucket Test

In questo test è stato utilizzato uno stream di 100 array, ognuno di 100K floating-point, fissando il numero di bucket ad 8. I valori della curva ideale sono stati calcolati usando il modello descritto precedentemente e con $L_{seq}(8) = 2200ms$, che è il valore calcolato eseguendo il *Bucket Sort Sequenziale* su un array di 100K floating-point. I risultati del test sono riportati nella Figura 3.1.
Analizzando il grafico possiamo osservare che tempi di completamento del *Bucket Sort Parallelo (Reale)* si avvicinano molto a quelli calcolati con il modello analitico (*Bucket Sort Parallelo (Ideale)*), avendo quindi un'efficienza vicina ad 1. Notiamo, inoltre, che i valori del *Bucket Sort Parallelo (Reale)* sono leggermente più alti di quelli ideali: questo è probabilmente dovuto all'overhead causato dai trasferimenti di memoria.

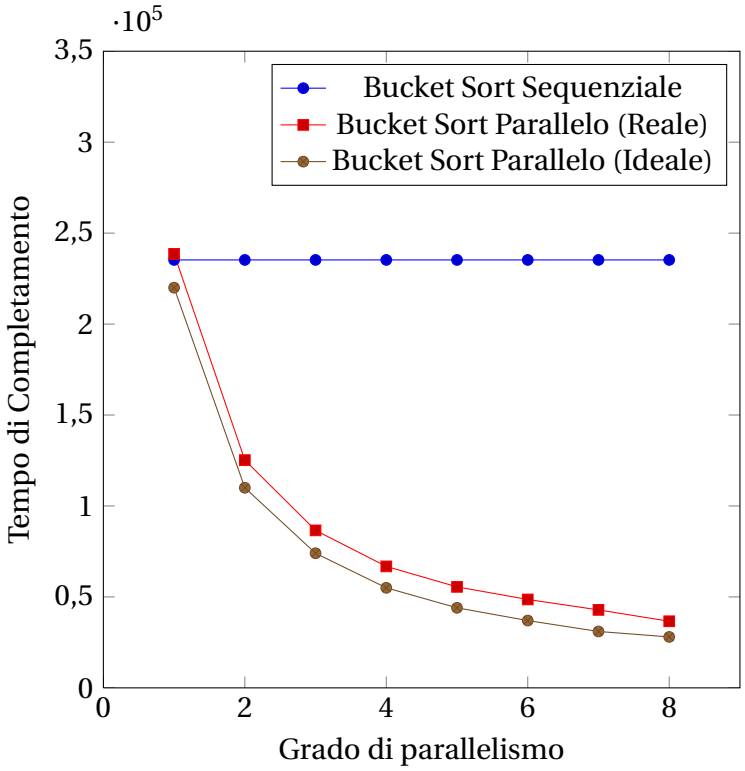

Figura 3.1: Test effettuato con uno stream di 100 array, ognuno di 100K floating-point. Il numero di bucket è stato fissato ad 8.

3.2 4 Bucket Test

In questo test è stato utilizzato uno stream di 100 array, ognuno di 40K floating-point, fissando il numero di bucket a 4. I valori della curva ideale sono stati calcolati usando il modello descritto precedentemente e con $L_{seq}(4) = 572ms$, che è il valore calcolato eseguendo il *Bucket Sort Sequenziale* su un array di 40K floating-point. I risultati del test sono riportati nella Figura 3.2.
Analizzando il grafico possiamo osservare che l'efficienza del *Bucket Sort Parallelo (Reale)* ha dei valori ancor più vicini ad 1 rispetto al test svolto nella Sezione 3.1, probabilmente per il fatto che, in questo caso, stiamo usando array di dimensione inferiore rispetto a prima e, quindi, è probabilmente presente un minor numero di trasferimenti di memoria.
Nonostante la latenza smetta di decrescere con un grado di parallelismo superiore al numero di bucket, il tempo di servizio continua comunque a decrescere visto che aumenta il numero massimo di istruzioni MDF che possono essere eseguite parallelamente dall'interprete MDF. Al diminuire del tempo di servizio, diminuisce anche il tempo di completamento.

3.3 Test sulla Latenza

In questo test è stato utilizzato uno stream di un array di 100K floating-point, fissando il numero di bucket a 4. I valori della curva ideale sono stati calcolati usando il modello

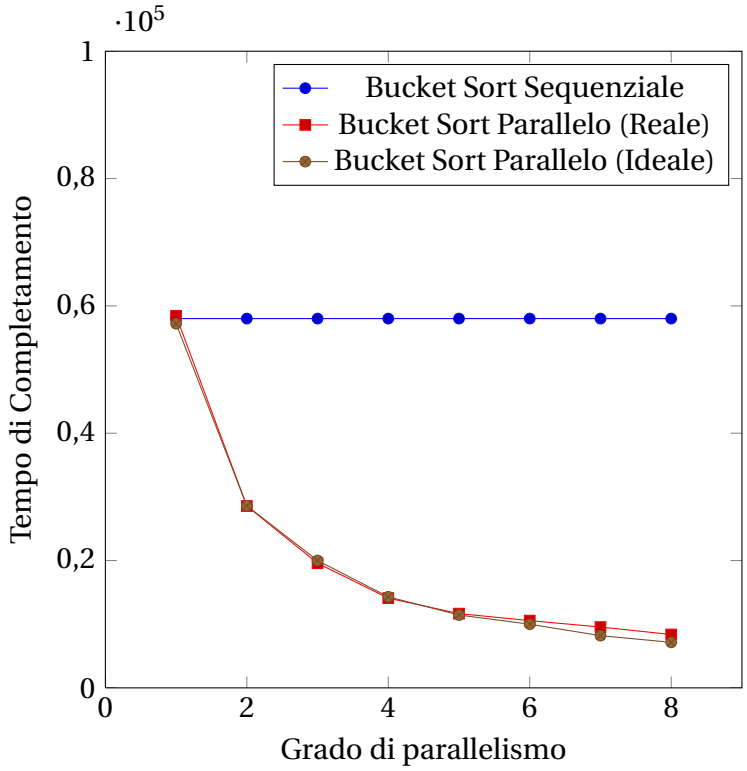

Figura 3.2: Test effettuato con uno stream di 100 array, ognuno di 40K floating-point. Il numero di bucket è stato fissato a 4.

descritto precedentemente e con $L_{seq}(8) = 2200ms$, che è il valore calcolato eseguendo il *Bucket Sort Sequenziale* su un array di 100K floating-point. I risultati sono riportati nella Figura 3.3.
Avendo un solo array da ordinare, in questo caso, il tempo di completamento è pari alla latenza. Infatti, andando ad aumentare il grado di parallelismo a valori superiori al numero di bucket in cui viene suddiviso l'array, non può essere sfruttato il fatto che potrebbero essere eseguite parallelamente un numero maggiore di istruzioni MDF rispetto ad esecuzioni con un grado di parallelismo inferiore. Infatti, essendoci un solo array da ordinare, il numero massimo di worker (istruzioni MDF che rappresentano i worker) che possono essere eseguiti parallelamente, in questo caso, è dato dal numero di bucket stesso.
Analizzando il grafico possiamo osservare che il tempo di completamento del *Bucket Sort Parallelo (Reale)* ha, anche in questo caso, un'efficienza molto vicina ad 1, tranne nel caso in cui il grado di parallelismo è pari a 3: questo, è dovuto al fatto che, anche aumentando il numero di worker che ordinano i bucket (rispetto all'esecuzione con grado di parallelismo 2), il tempo di completamento è sostanzialmente dato (trascurando i tempi introdotti dall'emitter e dal collector) dal tempo di completamento del worker più lento[3], non essendo presente, in questo caso, un perfetto load balancing. Nel nostro caso, infatti, abbiamo 4 bucket che devono essere ordinati da 3 worker: il miglior assegnamento del carico di lavoro che si può

[3] $T_{C_{par}} = max(T_{w_1}, T_{w_2}, T_{w_3})$ con T_{w_i} il tempo di completamento dell'i-esimo worker.

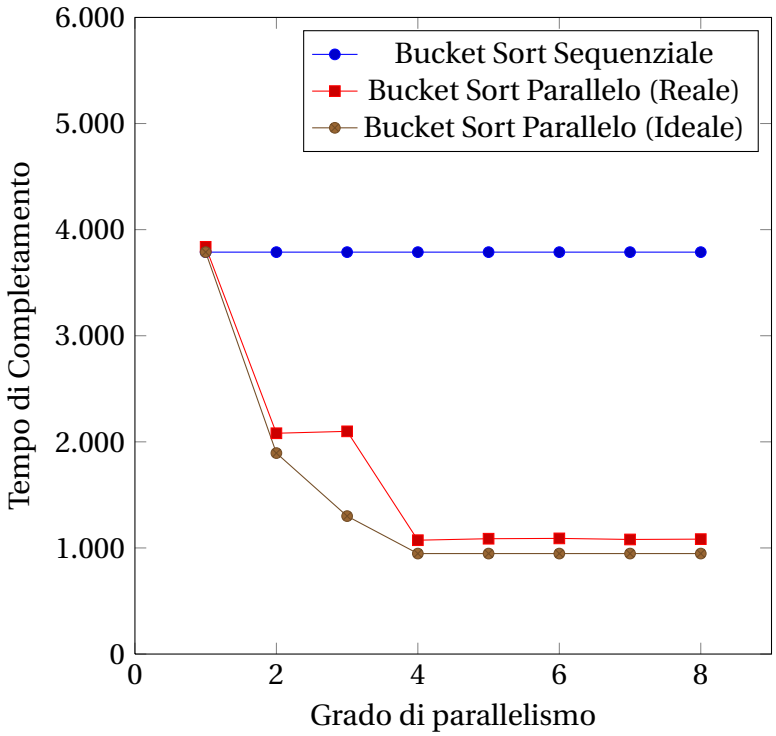

Figura 3.3: Test effettuato con uno stream di un array di 100K floating-point. Il numero di bucket è stato fissato a 4. Essendo lo stream composto da un solo array, in questo caso, il tempo di completamento è pari alla latenza.

fare è, chiaramente, quello di assegnare a due worker il compito d'ordinare un bucket a testa (due sequenze di bucket entrambe contenenti un solo bucket) e, al terzo worker, il compito d'ordinare i rimanenti due bucket (una sequenza di bucket contenente due bucket)[4]. Questa suddivisione del carico di lavoro presenta, infatti, la stessa latenza che si avrebbe nell'avere 4 bucket che devono essere ordinati da 2 worker: il miglior assegnamento del carico di lavoro che possiamo fare, in questo caso, è quello di assegnare ad entrambi i worker il compito di ordinare 2 bucket a testa (due sequenze di bucket contenenti due bucket ciascuna).

[4] l'implementazione realizzata assegna i bucket nel modo appena descritto