UNIVERSITÀ DI PADOVA  FACOLTÀ DI INGEGNERIA

Simulazione di algoritmi paralleli per il modello D-BSP su una gerarchia di cache ideali

Relatore: Ch.mo Prof. Pietracaprina Andrea

Correlatore: Ch.mo Prof. Pucci Geppino

Laureando: Silvestri Francesco

Corso di Laurea in Ingegneria Informatica

Anno Accademico 2004-2005

4 Ottobre 2005

Alle persone più importanti:
i miei genitori ed Elisa

Indice

Capitolo 1

Introduzione

In generale, gli algoritmi sequenziali presentano comportamenti regolari: il codice viene ripetuto più volte nei cicli; il valore della funzione booleana nei salti condizionati è prevedibile; gli accessi alla memoria localizzati e frequenti. Lo sfruttamento di queste caratteristiche può portare a velocizzare notevolmente gli algoritmi e può capitare che codici lenti diventino molto veloci (certamente i problemi NP-hard rimarranno tali...). Il modello di Von Neumann non considera nessuno di questi aspetti: quando fu ideato tali particolari erano trascurabili rispetto ad altri problemi computazionali, o addirittura ignoti. La tecnologia però è cambiata molto velocemente, e se prima era considerato veloce il tempo per una "pausa caffè", ora perfino l'accesso al disco (pochi millisecondi) è considerato lento. Quindi, ogni possibile miglioramento che valga per un'ampia famiglia di problemi è interessante e va analizzato. Negli ultimi anni l'attenzione di molti ricercatori si è focalizzata sull'analisi degli accessi alla memoria. Un algoritmo, in un breve arco di tempo, spesso accede agli stessi dati (si pensi ad un contatore in un ciclo), ma anche a dati vicini (la lettura di un array). Queste proprietà sono chiamate rispettivamente *località temporale* e *spaziale*, o in un unico modo *località di riferimento*. A livello pratico questo aspetto viene gestito con l'utilizzo di memorie di dimensioni e tempo di accesso variabili, ovvero con una *gerarchia di memoria*. Infatti le memorie veloci sono costose e quindi piccole; perciò è opportuno che vi siano contenuti solo i dati usati più frequentemente. Per accedere ai dati non presenti deve essere chiesto aiuto alla memoria sovrastante, più lenta ma più capiente. Una gerarchia di esempio può essere la seguente [1]:

Memoria	**Tempo di accesso (ns)**	**Dimensione (Byte)**
Register	10^0	10^2
On chip cache	10^0	10^3
L2 (SRAM)	10^1	10^6
Main RAM	10^2	10^9
Disk	10^7	10^{12}
Disk (tape)	10^9 (coffee break!)	10^{15}

La moltiplicazione di matrici è un esempio di come le località non siano prese in considerazione dalle vecchie mentalità di programmazione. Nel modello RAM, l'algoritmo iterativo

e quello ricorsivo hanno la stessa complessità asintotica, ma quello ricorsivo è ritenuto più lento perché richiede più volte l'inizializzazione di un livello ricorsivo. In pratica però esso si rivela molto più efficiente dell'iterativo. Infatti la sua struttura a sottoproblemi sempre più piccoli si adatta a pennello alle gerarchie di memoria: più piccolo è un sottoproblema più è probabile che venga contenuto nella memoria veloce. Molti modelli proposti si basano principalmente o su gerarchie a due livelli (EM [4], Cache ideale [17], ...) o su una funzione di costo crescente con l'indirizzo (HMM [2], BT [3], ...). Quest'ultimi si basano sul fatto che dati posizionati in indirizzi diversi nella stessa memoria richiedono tempo di accesso diverso per i percorsi diversi che compiono i segnali elettrici. Un modello molto elegante e semplice che si adatta alle architettture di memoria attuali è il modello a *cache ideale*, introdotto in [17] e che verrà usato come riferimento in questa tesi.

Anche nell'algoritmica parallela si riscontrano problemi simili al caso sequenziale: non esiste un modello che catturi tutte le caratteristiche desiderate. Infatti, i requisiti necessari a un modello parallelo sono spesso contrastanti tra loro: deve essere sufficientemente astratto per una classe di problemi, ma deve anche rispecchiare le caratteristiche di ogni elemento della stessa; deve essere semplice, ma deve anche restituire dei risultati corretti. Esistono molti modelli, ma hanno sempre qualche mancanza: ad esempio il modello PRAM offre primitive semplici e ricche e si basa su una memoria condivisa con tempo di accesso parallelo unitario, ma le macchine reali difficilmente hanno queste proprietà. Negli ultimi anni, però, è nata la necessità di un *bridging model* che sia in grado di trovare un compromesso tra tutte le richieste privilegiando quelle fondamentali. Una caratteristica cruciale è la *comunicazione* tra processori, ovvero lo scambio di informazioni (qualsiasi segnale inviato da un processore ad un altro). Si verifica di frequente che la comunicazione tra due nodi (processori) vicini è più efficiente di una tra nodi molto lontani. Il concetto di vicinanza si basa sulle tecnologie di rete utilizzate (circuiti integrati, cavo coassiale, fibra ottica, ...) ma anche sulla topologia (mesh, toro, ...). Ad esempio nell'interconnessione a mesh solo i nodi vicini comunicano a piena velocità, mentre le comunicazioni globali tra le due metà della rete sono rallentate di un fattore superficie/volume. Questa tendenza a preferire le comunicazioni "vicine" è detta *località di comunicazione*. Quindi una caratteristica fortemente desiderata dal bridging model è la possibilità di gestire questa località. Un modello che sembra adatto a questo scopo è il D-BSP [12], la cui portabilità su varie architettture viene analizzata in [6].

Il calcolo parallelo sembra l'unica via per la risoluzione di problemi che richiedono alte prestazioni; quindi un buon modo di procedere è quello di focalizzare prima l'attenzione su questo campo, poi riversare le conoscenze acquisite nel calcolo sequenziale con opportune simulazioni. Il concetto di località è presente sia nel calcolo sequenziale che nel calcolo parallelo e si può trasferire l'ampia conoscenza riguardo alla località di comunicazione alla località di riferimento. In letteratura esistono già articoli in proposito che riportano risultati interessanti [13, 14, 20, 23, 16] e propongono tecniche di prefetching e algoritmi sequenziali ottimi ottenuti da simulazioni su modelli con gerarchie di memoria.

In [16] vengono analizzate le relazioni tra la località di comunicazione del D-BSP e la località di riferimento dei modelli sequenziali HMM e BT, nei quali la posizione dei dati nella gerarchia di memoria è decisa direttamente dal programmatore. L'utilizzo di simula-

zioni permette di sottolineare le relazioni tra le località e allo stesso tempo ottenere algoritmi sequenziali ottimi partendo da algoritmi paralleli che utilizzano efficientemente la località di comunicazione. Nella presente tesi verrà sviluppata la procedura di simulazione affinché sia possibile utilizzarla in una gerarchia di cache. La principale differenza tra i modelli sequenziali utilizzati nel lavoro precedente e un modello con cache, è la gestione della posizione dei dati nella gerarchia di memoria: lo spostamento degli stessi nei vari livelli di cache non può essere esplicitamente diretto dal programmatore, ma viene gestito automaticamente dal modello. Per ottenere delle simulazioni efficienti di algoritmi paralleli è necessario, quindi, che la stessa procedura di simulazione segua certe caratteristiche compatibili con le pecularità della gerarchia. La procedura proposta è indipendente dai parametri della cache e dal numero di livelli della gerarchia; inoltre è generica, ovvero può essere applicata a qualsiasi algoritmo parallelo, anche quando il pattern di comunicazione è determinato durante l'esecuzione del programma. L'applicazione della simulazione ad algoritmi paralleli noti (moltiplicazione di matrici, trasformata discreta di Fourier, ordinamento) permette di ottenere programmi sequenziali quasi ottimi, ovvero l'aumento rispetto all'ottimo della complessità RAM e del numero di miss è solo logaritmico. È possibile ottenere algoritmi sequenziali ottimi analizzando le regolarità dei pattern di comunicazione dei programmi paralleli, a discapito però della generalità della simulazione. In questo modo è possibile ottenere algoritmi ottimi sia nel numero di miss che nella complessità RAM per la moltiplicazione di matrici e la trasformata discreta di Fourier.

La tesi è cosi strutturata: nel Capitolo 2 vengono descritti i modelli di gerarchia di memoria utilizzati, in particolare il modello a cache ideale; nel Capitolo 3 viene presentato il già citato lavoro [16] con un'introduzione al modello parallelo D-BSP; nel Capitolo 4 si trovano i risultati originali della tesi, ovvero la simulazione di una particolare versione del D-BSP su una cache ideale; nel Capitolo 5, invece, viene presentato il concetto di simulazione *ad hoc* e forniti alcuni esempi che valgono sia per il primo che per il secondo tipo di simulazione; infine nel Capitolo 6 verranno esposte le considerazioni finali sul lavoro.

Capitolo 2

Gerarchie di memoria

Buona parte degli algoritmi non accedono in maniera casuale ai dati in memoria, ma tendono a riutilizzare gli stessi dati (*località temporale*) e ad accedere a dati tra loro vicini (*località spaziale*) in brevi intervalli di tempo. Il modello RAM non considera queste caratteristiche e spesso algoritmi teoricamente buoni, risultano inefficienti quando implementati su macchine reali che contengono gerarchie di memoria.

Negli ultimi decenni sono stati proposti molti modelli che, in qualche modo, sono in grado di rappresentare le località temporale e spaziale, cioè in breve la *località di riferimento*. Il modello I/O (*Input/Output*) [4] assume che i dati risiedano su disco e debbano essere copiati in memoria per poter effettuare operazioni su di essi. L'HMM (*Hierarchical Memory Model*) [2], invece, è composto da un solo livello di memoria, ma il tempo di accesso dipende dall'indirizzo del dato: più "lontano" è il dato dal processore, ovvero più grande è l'indirizzo, maggiore sarà il tempo di accesso. Quest'ultimo modello gestisce solo la località temporale, mentre la località spaziale è trattata da una sua estensione, il BT (*Block Transfer*) [3], il quale è in grado di spostare blocchi di qualsiasi dimensione efficientemente. Il modello UMH (*Uniform Memory Hierarchy*) [5], infine, assume una funzione di costo a gradini. In tutti questi modelli la località è gestita direttamente dal programmatore, che può decidere quali blocchi spostare nella memoria (o zona di memoria) più veloce. Nei modelli che utilizzano la cache [17, 19] la gestione della località non può essere comandata direttamente dal programmatore, ma al più indirettamente con opportuni accorgimenti sull'algoritmo.

Il seguente capitolo è così strutturato: nel Paragrafo 2.1 verranno descritte le proprietà della cache; nel Paragrafo 2.2 il modello a cache ideale; nel Paragrafo 2.3 verranno descritti brevemente i modelli HMM e BT.

2.1 L'ABC della cache

La cache è una memoria che si interpone tra il processore e la memoria del sistema in cui risiedono i dati di interesse. È di dimensioni inferiori rispetto alla memoria centrale, ma ha una velocità di accesso ai dati maggiore. Se un algoritmo utilizza opportunamente le località di riferimento, la cache può migliorare notevolmente le prestazioni.

La cache è divisa in *frame* (o linee) di L word e, se la dimensione totale è di Z word, contiene $\frac{Z}{L}$ linee. Il processore ha accesso solo a dati presenti fisicamente in cache e se un dato richiesto non è presente in cache (**miss**) deve essere caricato dalla memoria. Più esattamente viene caricato il blocco di dimensione L contenente il dato poiché la memoria è suddivisa in blocchi di L word. Se invece il dato è presente in cache si ha un **hit**. Solitamente la cache gode della proprietà di **inclusione**, ovvero un dato in cache è presente anche in memoria.

L'**associatività** di una cache indica la cardinalità dell'insieme (*set*) di frame nei quali un blocco può essere copiato:

Fully associative: quando un blocco può essere allocato in qualsiasi frame della cache;

Direct mapped: quando un blocco può essere allocato in un solo frame della cache;

A-way set associative: quando un blocco può essere allocato in uno degli A frame disponibili. Se $A = 1$ si ottiene una cache direct mapped; se $A = \frac{Z}{L}$ si ottiene una cache fully associative.

Quando l'insieme dei frame in cui un blocco può essere allocato è pieno, è necessario scegliere il frame da liberare. Tale scelta è determinata dalla **politica di rimpiazzo** (*replacment policy*) utilizzata. Alcune tecniche sono:

LRU: viene svuotato il frame che contiene il blocco non referenziato da più tempo;

Random: viene scelto in maniera casuale quale frame liberare;

First-in, First-out: viene svuotato il frame che contiene il blocco che ha risieduto nella cache per più tempo;

Rimpiazzo Ottimo: si sceglie il frame contenente il blocco che verrà referenziato più in là nel futuro.

Evidentemente l'ultima politica è impossibile da realizzare perché richiede la conoscenza a priori degli accessi in memoria.

In [19] i miss vengono distinti in tre tipi:

1. **Cold miss** (o *compulsory miss*): quando un blocco viene caricato in cache per la prima volta;
2. **Capacity miss**: quando è miss in una cache fully-associative, ma non è cold;
3. **Conflict miss**: quando non è né un miss in una cache fully-associative né un cold miss, ovvero quando il set associato al blocco non contiene frame liberi.

È possibile estendere la gerarchia a due livelli cache-memoria ad $n \geq 2$ livelli. Il processore comunica con la cache più veloce e quando un dato viene richiesto ad una cache i si possono verificare due eventi: o questo è presente (hit) o deve essere richiesto il blocco che lo contiene alla cache $i + 1$ sovrastante (miss).

2.2 Il modello a cache ideale

Il modello a cache ideale CM(Z, L) è stato introdotto in [17] ed è costituito da una gerarchia a due livelli composta da una cache e una memoria ad accesso casuale di dimensione opportunamente grande. La cache è di tipo *fully associative* e con tecnica di rimpiazzo *ottima*; ha dimensione Z word[1] ed ogni blocco ha lunghezza pari a $L > 1$ word, quindi vi sono $\frac{Z}{L}$ linee di cache. Sulla struttura della cache si impone il vincolo non particolarmente restrittivo di **tall cache**, ovvero:

$$Z = \Omega(L^2). \tag{2.1}$$

Le prestazioni di un algoritmo di dimensione n vengono misurate rispetto alla complessità temporale in un modello RAM, indicata con $T(n, Z, L)$, e dal numero di miss $Q(n, Z, L)$. Se Z ed L sono chiare nel contesto, la complessità e il numero di miss verranno indicate rispettivamente con $T(n)$ e $Q(n)$.

Così come per la complessità RAM esiste un lower bound banale per un problema con n dati, ovvero $T(n) = \Omega(n)$, anche per il numero di miss esiste un certo limite inferiore banale. Questo è il numero di miss minimo per leggere i dati, supposti adiacenti, (cold miss) e quindi $Q(n, Z, L) = \Omega\left(1 + \frac{n}{L}\right)$. Di seguito si indicherà con "numero di miss lineare" proprio questa quantità, in anologia con "tempo lineare" che spesso indica il miglior tempo RAM possibile per un algoritmo.

Un algoritmo viene definito **cache aware** quando dipende dai parametri della cache che possono essere impostati sia in tempo di compilazione che di esecuzione. Se non vi sono, invece, parametri l'algoritmo è **cache-oblivious**.

Riportiamo di seguito alcuni algoritmi cache-oblivious descritti in [17, 18]. A fini espositivi riportiamo la dimostrazione della complessità della moltiplicazione di matrici, rimandando alle fonti citate per le rimanenti dimostrazioni;

2.2.1 Moltiplicazione di matrici

Siano A e B due matrici da moltiplicare di dimensione rispettivamente $m \times n$ e $n \times p$, con layout row-major. Il prodotto tra le due matrici C può essere calcolato ricorsivamente:

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} \qquad \text{se } \max\{m, n, p\} = m, \tag{2.2}$$

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 \qquad \text{se } \max\{m, n, p\} = n, \tag{2.3}$$

$$C = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} \qquad \text{se } \max\{m, n, p\} = p, \tag{2.4}$$

Il caso base si ha con $m = n = p = 1$ per il quale la moltiplicazione di matrici coincide con un prodotto tra scalari.

[1] Si assume che un word abbia dimensione costante $\Theta(1)$.

Teorema 2.1. *La complessità temporale è* $T(m,n,p) = \Theta(mnp)$*, mentre il numero di miss è*

$$Q(m,n,p) = O\left(1 + \frac{mn+np+mp}{L} + \frac{mnp}{L\sqrt{Z}}\right). \tag{2.5}$$

Dim. Con opportuni accorgimenti è possibile portare la somma dell'equazione (2.3) al caso base così che valga la seguente relazione su $T(m,n,p)$:

$$T(m,n,p) = \begin{cases} \Theta(1) & \text{se m=n=p=1,} \\ 2T(m/2,\ n,\ p) + \Theta(1) & \text{altrimenti se } \max\{m,n,p\} = m, \\ 2T(m,\ n/2,\ p) + \Theta(1) & \text{altrimenti se } \max\{m,n,p\} = n, \\ 2T(m,\ n,\ p/2) + \Theta(1) & \text{altrimenti se } \max\{m,n,p\} = p. \end{cases} \tag{2.6}$$

Sviluppando la ricorrenza si ottiene

$$T(m,n,p) = 2^{i+1}T\left(\frac{m}{2^x}, \frac{n}{2^y}, \frac{p}{2^z}\right) + 2^i\Theta(1),$$

dove $i = x+y+z$ e x rappresenta il numero di chiamate ricorsive dividendo m (analogamente y e z); per $i^* = \log m + \log n + \log p = \log mnp$ si raggiunge il caso base perché ad ogni chiamata ricorsiva viene diviso uno ed un solo termine e quindi:

$$T(m,n,p) = \Theta(mnp).$$

Sia α tale che se $\max\{m,n,p\} \leq \alpha\sqrt{Z}$, allora il problema può essere contenuto intermanete in cache. Per i miss vale la seguente relazione:

$$Q(m,n,p) = \begin{cases} O(1 + (mn+np+mp)/L) & \text{se } m,n,p \leq \alpha\sqrt{Z} \\ 2Q(m/2,n,p) + O(1) & \text{altrimenti se } \max\{m,n,p\} = m, \\ 2Q(m,n/2,p) + O(1) & \text{altrimenti se } \max\{m,n,p\} = n, \\ 2Q(m,n,p/2) + O(1) & \text{altrimenti se } \max\{m,n,p\} = p. \end{cases} \tag{2.7}$$

Essendo l'algoritmo ricorsivo, se un problema non può essere contenuto totalmente in cache allora il numero di miss è al più pari ai miss dei sottoproblemi più un numero costante di miss per l'elaborazione degli indici. Questa ricorrenza ha termine quando si raggiunge un sottoproblema tale che ogni lato delle matrici sia minore o uguale di $\alpha\sqrt{Z}$ e quindi il problema può essere contenuto totalmente in cache: in questo caso il numero di miss è al più quello per caricare il problema in cache (cold miss). Per caricare una matrice A di dimensione $m \times n$ sono necessari $O\left(1 + \frac{mn}{L}\right)$ miss. Questo valore è evidente se le righe sono allocate in locazioni contigue in memoria; se invece le righe non sono allocate con continuità, il numero di miss è $O\left(m + \frac{mn}{L}\right)$, ma tale caso si ha solo quando $\frac{\alpha\sqrt{Z}}{2} < n \leq \alpha\sqrt{Z}$ e quindi il termine m è trascurabile per (2.1). Questa affermazione è vera perché per ipotesi la matrice è allocata in memoria in row-major e solo le sottomatrici ottenute partizionando A rispetto le colonne possono avere righe non contigue. Considerando tutte e tre le matrici A, B e C si ottiene il

caso base.
Dimostriamo ora che la (2.7) ha soluzione

$$O\left(1+\frac{mn+np+mp}{L}+\frac{mnp}{L\sqrt{Z}}\right). \tag{2.8}$$

Sviluppando la ricorrenza si ottiene: $Q(m,n,p) = O(2^{i+1}Q(m/2^x, n/2^y, p/2^z) + 2^i)$, dove $i = x+y+z$ e x rappresenta il numero di chiamate ricorsive dividendo m (analogamente y e z). Indichiamo con i^* il valore di i con cui si raggiunge il caso base:

$$Q(m,n,p) = O(2^{i^*+1}CB + 2^{i^*}) = O(2^{i^*}CB) \tag{2.9}$$

dove CB indica il valore del caso base. Abbiamo ora 4 casi ortogonali tra loro che dipendono dai valori iniziali assunti da m, n, p:

1. $m,n,p > \alpha\sqrt{Z}$
   Entrambi i termini verranno ricorsivamente divisi per due finché $m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}]$, ovvero $m,n,p = \Theta(\sqrt{Z})$, $i^* = \log mnp/(\alpha^3 Z\sqrt{Z})$ e $CB = O(Z/L)$. La (2.9) diventa quindi $Q(m,n,p) = O((mnp/Z\sqrt{Z})Z/L) = O(mnp/L\sqrt{Z})$ che coincide con (2.8) per le ipotesi iniziali.

2. $n,p > \alpha\sqrt{Z}, m \leq \alpha\sqrt{Z}$
   In questo caso solo n e p verranno ricorsivamente divisi per due finché $n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}]$, ovvero $i^* = \log np/\alpha^2 Z$, $n,p = \Theta(\sqrt{Z})$ e $CB = O(Z/L)$. La (2.9) diventa $Q(m,n,p) = O((np/Z)(Z/L)) = O(np/L)$ che coincide con (2.8) per le ipotesi iniziali.

3. $m > \alpha\sqrt{Z}, n,p \leq \alpha\sqrt{Z}$
   In questo caso solo m viene ricorsivamente diviso per due così che $i^* = \log m/\alpha\sqrt{Z}$, $m = \Theta(\sqrt{Z})$ e $CB = O((n+p)\sqrt{Z}/L)$. La (2.9) diventa $Q(m,n,p) = O((m/\sqrt{Z})(n+p)\sqrt{Z}/L) = O((mn+mp)/L)$ che coincide con (2.8) per le ipotesi iniziali.

4. $m,n,p \leq \alpha\sqrt{Z}$
   Per questi valori la ricorrenza entra subito nel caso base ($i^* = 0$) e $Q(m,n,p) = O(1+(mn+np+mp)/L)$ che coincide con (2.8) per le ipotesi iniziali.

□

2.2.2 Trasposta di una matrice

Siano A e B due matrici di dimensione rispettivamente $m \times n$ e $n \times m$ e $B = A^T$. La trasposizione di A può essere ottenuta con il seguente algoritmo ricorsivo ottimo:

$$\begin{aligned} B &= (A_1, A_2)^T = \begin{pmatrix} A_1^T \\ A_2^T \end{pmatrix} \quad \text{se } n \geq m \\ B &= \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}^T = \left(A_1^T, A_2^T\right) \quad \text{se } n < m \end{aligned} \tag{2.10}$$

Il caso base si raggiunge quando $m = n = 1$ e si ha $B = A$.

Teorema 2.2. *L'algoritmo ricorsivo è ottimo, ha complessità* $T(m,n) = \Theta(mn)$ *e il numero di miss vale*

$$Q(m,n) = \Theta\left(1 + \frac{mn}{L}\right). \tag{2.11}$$

2.2.3 Trasformata discreta di Fourier

La trasformata discreta di Fourier di un vettore può essere calcolata in modo ottimo con una piccola variante dell'algoritmo di Cooley-Tukey [10]. Sia $n = 2^k$ la dimensione del vettore V da trasformare, $n_1 = 2^{\lceil (\log n)/2 \rceil}$ e $n_2 = 2^{\lfloor (\log n)/2 \rfloor}$, consideriamo il vettore V come una matrice A di dimensione $n_1 \times n_2$ in row-major. L'algoritmo si compone dei seguenti sei passi:

1. Trasponi A in place.
2. Calcola n_2 trasformate di n_1 elementi. Per il punto precedente gli n_1 valori si trovano in locazioni adiacenti.
3. Moltiplica ogni elemento per il twiddle factor.
4. Trasponi A in place.
5. Calcola n_1 trasformate di n_2 elementi. Per il punto precedente gli n_2 valori si trovano in locazioni adiacenti.
6. Trasponi A in place.

Teorema 2.3. *L'algoritmo per la trasformata discreta di Fourier è ottimo, la complessità RAM vale* $T(n) = \Theta(n \log n)$ *e il numero di miss è:*

$$Q(n) = \Theta\left(1 + \frac{n}{L} + \frac{n \log n}{L \log Z}\right) \tag{2.12}$$

2.2.4 Ordinamento: Funnelsort

Sia n la dimensione dell'array da ordinare, funnelsort si basa sui seguenti tre passi:

1. Se $n < \overline{n}$ utilizza un algoritmo iterativo per ordinare l'array ed esce. $\overline{n} = \Theta(1)$ indica un opportuno valore;
2. divide l'array in $n^{\frac{1}{3}}$ sottoarray di $n^{\frac{2}{3}}$ elementi e li ordina ricorsivamente;
3. unisce i sottoarray con un $n^{\frac{1}{3}}$-merger.

Un k-merger permette di unire k sequenze ordinate mantenendo l'ordine e il seguente invariante:

Invariante 2.1. *Ogni attivazione di un* k*-merger restutuisce* k^3 *elementi ordinati unendo le* k *sequenze in ingresso.*

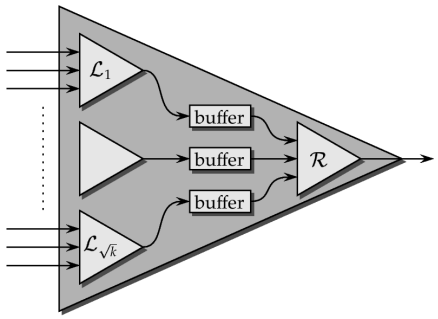

Figura 2.1: k-merger

La sua struttura è ricorsiva e costituita da $\sqrt{k}+1$ $\sqrt{k}$-merger (vedi Fig. 2.1). Gli ingressi vengono divisi in $\sqrt{k}$ gruppi da $\sqrt{k}$ sequenze che saranno l'input di altrettanti $\sqrt{k}$-merger $\mathcal{L}_1, \ldots, \mathcal{L}_{\sqrt{k}}$. L'uscita di ognuno di questi $\sqrt{k}$-merger è collegata ad uno dei $\sqrt{k}$ buffer FIFO di dimensione $2k^{\frac{3}{2}}$ che a sua volta è un ingresso del $\sqrt{k}$-merger $\mathcal{R}$. Per mantenere l'invariante, $\mathcal{R}$ deve essere attivato esattamente $k^{\frac{3}{2}}$ volte; prima di ogni attivazione i buffer devono essere riempiti dai rispettivi merger se contengono meno di $k^{\frac{3}{2}}$ (ciò spiega il sovradimensionamento dei buffer).

Teorema 2.4. *Funnelsort è ottimo e richiede tempo RAM* $T(n) = \Theta(n \log n)$*, spazio* $S(n) = \Theta(n)$ *e numero di miss*

$$Q(n) = \Theta\left(1 + \frac{n}{L} + \frac{n \log n}{L \log Z}\right) \tag{2.13}$$

2.2.5 Ordinamento: Distribution Sort

La seguente versione di distribution sort non utilizza algoritmi (deterministici o non) per determinare i punti di divisione (splitting point), ma costruisce i bucket al volo mantenendo certi invarianti. Sia A l'array da ordinare, n la sua dimensione, n_i la dimensione del bucket B_i, $1 \leq i \leq d$. L'algoritmo si divide in 4 fasi:

1. Se $n \leq \overline{n}$, dove $\overline{n} = \Theta(1)$ è un opportuna costante, ordina A con un algoritmo iterativo. Se invece $n > \overline{n}$ divide A in $\sqrt{n}$ subarray i, con $i \in [1, \sqrt{n}]$, di dimensione $\sqrt{n}$ e ordina ogni subarray ricorsivamente;

2. Distribuisce gli elementi dei subarray in d bucket $B_1, \ldots, B_d$ mantenendo i seguenti invarianti:

   (a) $max\{x \mid x \in B_i\} \leq min\{x \mid x \in B_{i+1}\}$

   (b) $n_i \leq 2\sqrt{n}$

3. Ricorsivamente ordina i d bucket;

4. Copia i d bucket in A sequenzialmente.

Teorema 2.5. *Distribution sort è ottimo e richiede tempo RAM* $T(n) = \Theta(n \log n)$*, spazio* $S(n) = \Theta(n)$ *e numero di miss*

$$Q(n) = \Theta\left(1 + \frac{n}{L} + \frac{n \log n}{L \log Z}\right) \tag{2.14}$$

2.2.6 Estensioni

Il modello a cache ideale è abbastanza restrittivo e non rispecchia la struttura delle cache reali, che solitamente né sono fully associative, né utilizzano la politica di rimpiazzo ottimo. Si può però dimostrare che la politica di rimpiazzo ottimo può essere sostituita con quella LRU (*least-recently used*) senza modificare la complessità di un'ampia classe di algoritmi.

Definiamo inizialmente il concetto di algoritmo regolare:

Definizione 2.1. *Un algoritmo per il modello a cache ideale si dice* ***regolare*** *se:*

$$Q(n, Z, L) = O(Q(n, 2Z, L)). \tag{2.15}$$

Riportiamo per chiarezza il Teorema 6 presentato in [21]:

Teorema 2.6. *Siano* n_{LRU} *e* n_{OPT} *il numero di linee in una cache con politica di rimpiazzo rispettivamente LRU e ottimo. Se entrambe le cache sono vuote e viene applicata ad entrambe una successione di richieste di accesso alla memoria, allora vale la seguente maggiorazione:*

$$Q_{LRU} \leq \frac{n_{LRU}}{(n_{LRU} - n_{OPT} + 1)} Q_{OPT},$$

dove Q_{LRU} *indica il numero di miss nella cache LRU,* Q_{OPT} *il numero di miss nella cache con politica di rimpiazzo ottimo.*

Valgono i seguenti teoremi:

Lemma 2.7. *Sia* $Q^*(n, Z, L)$ *il numero di miss in una cache ideale e* $Q(n, Z, L)$ *il numero di miss in una cache con strategia LRU. Allora* $Q(n, Z, L) \leq 2Q^*(n, \frac{Z}{2}, L)$*.*

Dim. Dal Teorema 2.6 si ottiene la seguente maggiorazione:

$$Q(n, Z, L) \leq \frac{Z}{L\left(\frac{Z}{L} - \frac{Z^*}{L} + 1\right)} Q^*(n, Z^*, L)$$

Se $Z^* = Z/2$ si ottiene la tesi $Q(n, Z, L) \leq 2Q^*(n, Z/2, L)$. □

Corollario 2.8. *Il numero di miss tra una cache ideale e una cache con strategia LRU non cambia asintoticamente se l'algoritmo è regolare.*

Dim. Poiché la strategia ottima dà il minor numero di miss rispetto ad altre tecniche e per il Lemma 2.7 e l'equazione (2.15) si ottiene:

$$Q^*(n, Z, L) \leq Q(n, Z, L) \leq 2Q^*(n, \frac{Z}{2}, L) = \Theta(Q^*(n, Z, L))$$

quindi $Q(n, Z, L) = \Theta(Q^*(n, Z, L))$. □

Il modello CM si può facilmente estendere a $k \geq 1$ livelli di cache. Indichiamo con $\mathcal{L}_i$ la cache al livello i ($\mathcal{L}_0$ è la cache più veloce) e con Z_i e L_i, $0 \leq i < k$, rispettivamente la dimensione totale e la lunghezza del frame di $\mathcal{L}_i$. Indichiamo la gerarchia con CM($\mathbf{Z}, \mathbf{L}$), dove le posizioni i-esime dei vettori $\mathbf{Z}$ ed $\mathbf{L}$ contengono rispettivamente i parametri Z_i ed L_i di $\mathcal{L}_i$. Supponiamo che ogni cache goda della proprietà di inclusione, che utilizzi la politca LRU, che $\mathcal{L}_{i+1}$, $0 \leq i < k-1$, contenga un numero maggiore di linee rispetto alla cache sottostante $\mathcal{L}_i$ ed infine che se due dati appartengono alla stessa linea di cache in $\mathcal{L}_i$, allora appartengono allo stesso frame anche in $\mathcal{L}_{i+1}$. Un frame è marcato se contiene almeno un dato presente nei livelli sottostanti. L'accesso ai dati è così gestito: uno hit nella cache $\mathcal{L}_i$ non viene percepito dai livelli sovrastanti; in caso di miss nella cache $\mathcal{L}_i$ il dato viene chiesto ricorsivamente al livello sovrastante ed inserito in $\mathcal{L}_i$, eventualmente eliminando il blocco non referenziato da più tempo e non marcato; il blocco rimosso viene posizionato in testa alla lista LRU della cache $\mathcal{L}_{i+1}$.

Lemma 2.9. *Siano (Z_i, L_i) i parametri della cache $\mathcal{L}_i$, $0 \leq i < k$, di una gerarchia di memoria CM($\boldsymbol{Z}, \boldsymbol{L}$). Il livello $\mathcal{L}_i$ contiene gli stessi blocchi di una semplice cache CM(Z_i, L_i) con politica LRU sottoposta alla stessa sequenza di accessi in memoria.*

Lemma 2.10. *Un algoritmo cache-oblivious ottimo e regolare incorre in un numero ottimo di miss in ogni livello di una gerarchia di cache LRU.*

Dim. Per il Lemma 2.9 ogni cache della gerarchia si comporta come se non vi fossero altri livelli; per il Lemma 2.8 l'algoritmo, essendo cache-oblivious ottimo e regolare, è ottimo anche per una cache LRU di dimensioni arbitrarie; da questo il lemma. □

In [17] non viene introdotta una funzione di costo unitaria. Per le finalità di questa tesi è importante avere una misura della complessità temporale che consideri sia il numero di operazioni sia il ritardo dovuto alla latenza di un miss; introduciamo quindi la funzione di costo definita in [19] per una gerarchia di cache. Sia $\mathcal{L}_i$ la cache del livello i, $0 \leq i < k$, e siano Z_i, L_i e t_i rispettivamente la dimensione della cache, del frame e il tempo richiesto da $\mathcal{L}_i$ per leggere un blocco nel livello sovrastante $\mathcal{L}_{i+1}$ (o memoria se $i = k-1$). Indichiamo inoltre con $\mathbf{Z}$, $\mathbf{L}$ e $\mathbf{t}$ i vettori contenenti nella posizione i i valori Z_i, L_i e t_i del livello $\mathcal{L}_i$. Se $T^{RAM}(n, \mathbf{Z}, \mathbf{L})$ è la complessità RAM di un algoritmo di taglia n, $Q_i(n, \mathbf{Z}, \mathbf{L})$ il numero di miss della cache $\mathcal{L}_i$, allora la funzione costo finale è

$$T(n, \mathbf{Z}, \mathbf{L}, \mathbf{t}) = T^{RAM}(n, \mathbf{Z}, \mathbf{L}) + \sum_{i=0}^{k-1} Q_i(n, \mathbf{Z}, \mathbf{L}) t_i. \tag{2.16}$$

Nei prossimi capitoli indicheremo con $T^{RAM}(n, \mathbf{Z}, \mathbf{L})$ la complessità RAM (eventualmente senza i parametri Z e L se non necessari, ad esempio per algoritmi cache-oblivious), con $Q(n, \mathbf{Z}, \mathbf{L})$ il numero di miss e con $T(n, \mathbf{Z}, \mathbf{L}, \mathbf{t})$ la misura di tempo definita dalla (2.16). Se la gerarchia è composta da una sola cache allora i vettori $\mathbf{Z}$, $\mathbf{L}$ e $\mathbf{t}$ devono essere sostituiti dal rispettivo valore scalare Z, L e t.

In [5] viene provato che un numero di miss ottimo per ogni livello dell'UMH non garantisce l'ottimalità rispetto alla funzione costo utilizzata. Se supponiamo che i dati non possano essere copiati in più livelli di cache contemporaneamente, allora la funzione definita in (2.16) risulta ottima appena sono ottimi il numero di miss per livello e la complessità RAM. Proviamo tale affermazione. Sia $\mathcal{P}$ un algoritmo ottimo sia nella complessità RAM che nel numero di miss. Se per assurdo la funzione costo T non fosse ottima, allora esisterebbe un algoritmo $\mathcal{P}'$ per cui o la complessità RAM o il numero di miss in un livello sarebbero minori dell'ottimo, ma questo è impossibile. È evidente che se la funzione costo (2.16) è ottima non necessariamente saranno contemporaneamente ottimi la complessità RAM e il numero di miss per ogni livello.

2.3 Altri modelli

Come accennato all'inizio esistono molti altri modelli che riescono a gestire il concetto di località di riferimento. Descriviamo brevemente due modelli che verranno utilizzati nei prossimi capitoli riguardanti la simulazione di algoritmi paralleli.

2.3.1 Il modello HMM

L'HMM($f(x)$) (*Hierarchical Memory Management*) [2] è una macchina ad accesso casuale dove la richiesta del dato all'indirizzo x richiede tempo $f(x)$, con $f(x)$ non decrescente. Il modello assume che N operazioni sulle celle $x_0, \ldots x_{N-1}$ richiedano tempo $1+\sum_{i=0}^{N-1} f(x_i)$, indipendentemente da N. Solitamente la funzione $f(x)$ è regolare (*well behaved* o polinomialmente limitata), ovvero:

Definizione 2.2. *Una funzione $f(x)$ è **regolare** se esiste una costante $c \geq 1$ tale che $\forall x$ $f(2x) \leq cf(x)$.*

Valori spesso utilizzati in letteratura per $f(x)$ sono x^α, $0 < \alpha < 1$, e $\log x$. Diversamente dal modello a cache ideale, l'HMM utilizza una funzione continua che porta quasi al limite il concetto di gerarchia di memoria. Riportiamo alcuni lower-bound, dimostrati sempre in [2], che saranno utilizzati nei successivi capitoli:

Lemma 2.11. *Sia T il costo di esecuzione di un algoritmo sul modello HMM($f(x)$). Se $f(x) = \log x$, allora valgono i seguenti lower bound:*

- *$T(N) = \Omega(N^3)$ per la moltiplicazione di matrici di dimensione $N \times N$ con solo operazioni di semianello;*

- $T(N) = \Omega(N \log N \log\log N)$ *per la trasformata di Fourier e l'ordinamento di un vettore con* N *elementi.*

Lemma 2.12. *Sia* T *il costo di esecuzione di un algoritmo sul modello HMM*$(f(x))$*. Se* $f(x) = x^\alpha$ *con* $0 < \alpha < 1$*, allora valgono i seguenti lower bound:*

- $$T(N) = \Omega \begin{cases} N^{2\alpha+2} & se\ \alpha > \frac{1}{2} \\ N^3 \log N & se\ \alpha = \frac{1}{2} \\ N^3 & se\ \alpha < \frac{1}{2} \end{cases}$$

  per la moltiplicazione di matrici di dimensione $N \times N$ *con solo operazioni di semianello;*

- $T(N) = \Omega(N^{\alpha+1})$ *per la trasformata di Fourier e l'ordinamento di un vettore con* N *elementi.*

2.3.2 Il modello BT

Il modello BT$(f(x))$ [3] è un HMM$(f(x))$ con la possibilità di copiare efficientemente blocchi di celle. Come nell'HMM, il tempo per l'accesso all'indirizzo x è $f(x)$, ma copiare un blocco con b celle $[x-b+1, x]$, $\forall\ b \geq 1$, in un altro blocco disgiunto dal primo $[y-b+1, y]$ richiede tempo $f(\max\{x, y\}) + b$. Ristringiamo il nostro campo di interesse alle sole funzioni $f(x)$ regolari. Nei prossimi capitoli verrà utilizzato il seguente teoremi dimostrato sempre in [3]:

Lemma 2.13. *Sia* T *il costo di esecuzione di un algoritmo sul modello BT*$(f(x))$*. Se* $f(x) = \log x$ *o* $f(x) = x^\alpha$ *con* $0 < \alpha < 1$*, allora valgono i seguenti lower bound:*

- $T(N) = \Omega(N^3)$ *per la moltiplicazione di matrici di dimensione* $N \times N$ *con solo operazioni di semianello;*

- $T(N) = \Omega(N \log N)$ *per la trasformata di Fourier e l'ordinamento di un vettore con* N *elementi.*

È interessante notare che i risultati non cambiano per le due funzioni di costo considerate.

Capitolo 3

Dal modello D-BSP ai modelli HMM e BT

In letteratura è viva da molto tempo l'idea di poter ottenere dei buoni algoritmi sequenziali da algoritmi paralleli e quindi di poter sfruttare (e incrementare) le conoscenze su quest'ultimi a favore dei primi. Sia nei modelli paralleli che nei modelli sequenziali recenti si è sviluppato il concetto di località: di comunicazione nei primi e di riferimento (temporale e spaziale) nei secondi. Gli studi attuali cercano, con buoni risultati, un punto di convergenza tra i due tipi di località che permetta il passaggio tra le due forme e una trattazione uniforme.

I primi lavori riguardavano la simulazione di algoritmi fine-grained per BSP su un modello EM e si basano sulla somiglianza dell'alternarsi di calcolo e comunicazione dei modelli paralleli con l'alternarsi di accessi alla memoria veloce e accessi al disco del modello EM [13, 14, 20].

In [23] tecniche di prefetching della cache sono state ricavate dalla simulazione di algoritmi per il modello PRAM. In [9], inoltre, viene provato come algoritmi per PRAM, che coinvolgono insiemi di processori di dimensione geometricamente decrescente, possono generare algoritmi per EM efficienti.

Uno studio più generico è stato proposto in [7] dove viene introdotto il modello $M_d(n, p, m)$, una mesh d-dimensionale composta da p processori HMM con una memoria di dimensione $\frac{nm}{p}$ e funzione di accesso $f(x) = \lceil \frac{x+1}{m} \rceil^{\frac{1}{d}}$. Il costo per inviare un messaggio ad un nodo vicino è pari al tempo di accesso alla cella più lontana della memoria locale. Lo slowdown della simulazione di una $M_d(n, n, m)$ su una $M_d(n, p, m)$, $p < n$, vale $\frac{n}{p}\Lambda(n, p, m)$, ovvero un fattore $\Lambda(n, p, m)$ in più rispetto alla pura perdita di parallelismo. Il termine supplementare in certi casi non è trascurabile e può crescere fino a $\left(\frac{n}{p}\right)^{\frac{1}{d}}$.

Nel lavoro [16], che verrà successivamente descritto, viene presentata la simulazione di algoritmi per D-BSP su HMM. Se la banda del D-BSP equivale al tempo di accesso alla memoria del HMM lo slowdown equivale alla semplice perdita di parallelismo e si ha quindi un passaggio di località netto. Inoltre viene provato come un D-BSP con nodi HMM unisca in maniera efficiente la località di comunicazione con la località di riferimento. Infine viene

presentata la simulazione di algoritmi per D-BSP sul modello sequenziale BT che permette di ricavare algoritmi ottimi e quasi-ottimi per quest'ultimo modello.

Il capitolo è così suddiviso: nel Paragrafo 3.1 viene descritto il modello parallelo D-BSP; nel Paragrafo 3.2 la simulazione sul modello HMM introdotta in [16]; infine nel Paragrafo 3.3 la simulazione sul modello BT presentata sempre in [16].

3.1 Il modello D-BSP

Il D-BSP (*Decomposable BSP*) [12] è una variante del BSP (*Bulk Synchronous Parallel*) introdotto in [22]. È costituito da un insieme di N processori $P_0, \dots P_{N-1}$ comunicanti attraverso un router e partizionati in cluster. Ogni cluster può eseguire delle istruzioni (superstep) indipendentemente dagli altri ed è caratterizzato dall'inverso g della banda di comunicazione e da una latenza l. Se come spesso avviene la banda diminuisce al crescere della taglia del cluster, il modello enfatizza la località di comunicazione perché le comunicazioni tra nodi vicini, ovvero contenuti in cluster piccoli, sono più efficienti rispetto a quelle tra processori lontani.

Una particolare versione è il *recursive* D-BSP, in cui i nodi vengono partizionati in cluster da una decomposizione binaria gerarchica. Sia N, potenza di due, il numero di nodi e C_j^i, $0 \leq i \leq \log N$ e $0 \leq j < 2^i$, il j-esimo cluster di tipo i (i-cluster); le partizioni (cluster) si ottengono ricorsivamente:

$$C_j^i = \begin{cases} \{P_j\} & \text{se } i = \log N \\ C_{2j}^{i+1} \cup C_{2j+1}^{i+1} & \text{altrimenti} \end{cases} \quad \text{dove } 0 \leq i \leq \log N \text{ e } 0 \leq j < 2^i. \tag{3.1}$$

Ogni i-cluster contiene per costruzione $\frac{N}{2^i}$ nodi e ha parametri di banda e latenza rispettivamente g_i e l_i. Poniamo per generalità $g_{\log N} = 0$ e $l_{\log N} = 0$ perché un $\log N$-cluster è costituito da un solo processore e dunque non vi possono essere comunicazioni.

Un programma $\mathcal{P}$ per D-BSP è costituito da una serie di i-superstep $0 \leq i \leq \log N$ e si può supporre che termini sempre con uno 0-superstep. In un i-superstep, ogni processore esegue una computazione locale di al più $O(\tau)$ operazioni e comunica all'interno del suo i-cluster inviando o ricevendo al più $O(h)$ messaggi (h-relation), i quali saranno però disponibili al processore solo all'inizio del successivo superstep. Al termine di ogni i-superstep i processori all'interno dello stesso i-cluster vengono sincronizzati tra loro. Il tempo totale $\tilde{\tau}$ per eseguire un i-superstep è quindi:

$$\tilde{\tau} = O(\tau + hg_i + l_i) \tag{3.2}$$

Se $\mathcal{P}$ è composto da k_i i-superstep, $0 \leq i \leq \log N$, allora il tempo $\tilde{T}$ richiesto da tutto l'algoritmo è:

$$\tilde{T} = O\left(\sum_{i=0}^{\log N} k_i(\tau + hg_i + l_i)\right) = O\left(T + \sum_{i=0}^{\log N} k_i(hg_i + l_i)\right), \tag{3.3}$$

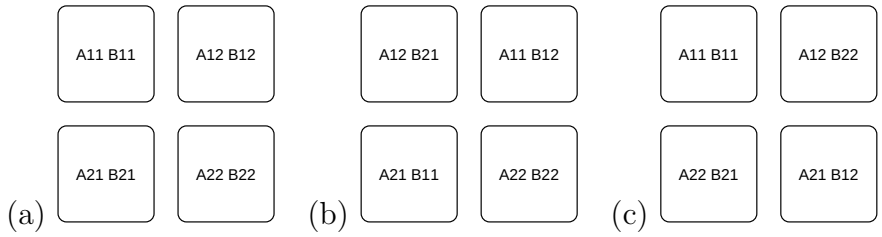

Figura 3.1: Esecuzione ricorsiva della moltiplicazione tra A e B

dove $T = \tau \sum_{i=0}^{\log N} k_i$.

Di seguito con μ verrà indicato il *contesto* di un processore, ovvero il suo stato; μ viene usato anche per indicare la dimensione in word del contesto. Con *simulazione* o *esecuzione* di un contesto indicheremo lo svolgimento della computazione locale a partire dallo stato rappresentato dal contesto. Un programma per D-BSP è *fine-grained* se il contesto associato ad ogni processore ha dimensione $\mu = \Theta(1)$.

Il modello appena descritto viene indicato con D-BSP$(N, \mathbf{g}, \mathbf{l})$, dove $\mathbf{g} = \{g_0, g_1, \ldots, g_{\log N}\}$ e $\mathbf{l} = \{l_0, l_1, \ldots, l_{\log N}\}$. Nella seguente trattazione la latenza verrà trascurata rispetto alla banda di comunicazione e quindi non verrà mai specificato il parametro $\mathbf{l}$.

Il modello parallelo indicato con D-BSP$(N, g(x))$ è un modello D-BSP dove il parametro di banda per un i-cluster vale $g(\frac{\mu N}{2^i})$.

3.1.1 La moltiplicazione di matrici

Siano A e B due matrici di dimensione $\sqrt{N} \times \sqrt{N}$, $N = 2^{2n}$, e sia $C = A \times B$. La moltiplicazione delle due matrici può essere calcolata in un D-BSP$(N, \mathbf{g})$ con il classico algoritmo ricorsivo che si basa sulla seguente proprietà di sottostruttura:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix} \tag{3.4}$$

Distribuiamo uniformemente e arbitrariamente gli elementi di ogni matrice tra gli N processori, cossiché ogni processore avrà contesto $\Theta(1)$ (ogni nodo contiene un dato per ognuna delle matrici A; B e C). Siano ora A' e B' le due matrici di un sottoproblema con $\frac{N}{2^i}$ elementi ciascuna e contenute in un i-cluster, $0 \leq i < \log N$. Gli 8 sottoproblemi definiti possono essere risolti in due round in cui ognuno dei quattro sottocluster esegue una singola moltiplicazione di matrici $\frac{1}{2}\sqrt{\frac{N}{2^i}} \times \frac{1}{2}\sqrt{\frac{N}{2^i}}$. Precedentemente ad ogni round, viene effettuato un i-superstep per distribuire gli elementi delle matrici ai rispettivi $i+2$-cluster. Un esempio di assegnazione di sottoproblemi a sottocluster è rappresentato in Fig. 3.1. La ricorsione termina quando si raggiunge un $\log N$-cluster in cui viene effettuato il prodotto tra due scalari in tempo $\Theta(1)$. In ogni superstep il processore compie $\Theta(1)$ operazioni e scambia $\Theta(1)$ dati

(viene spedito al più un solo elemento per matrice) e ogni nodo esegue in totale k_i i-superstep dove:

$$k_i = \begin{cases} 2^{j+1} & \text{se } i = 2j \ \forall\ 0 \leq j \leq \frac{\log N}{2} = n \\ 0 & \text{altrimenti.} \end{cases}$$

Dimostriamo la correttezza della ricorsione per induzione. È evidente che superstep di indice dispari non sono mai usati (infatti uno 0-cluster viene sempre diviso ricorsivamene in quattro) e che $k_0 = 2$. Supponiamo per ipotesi che $k_{2j} = 2^{j+1}$; per costruzione dopo ogni $2j$-superstep, con $2j < \log N$, vengono eseguiti sequenzialmente due $2(j+1)$-superstep, quindi $k_{2(j+1)} = 2 * k_{2j} = 2 * 2^{j+1} = 2^{j+2}$, ovvero la tesi. Per essere effettivamente un programma per D-BSP è necessario inserire uno 0-superstep conclusivo, ma a fini asintotici il suo contributo è trascurabile poiché $k_0 = 2 > 0$.

Lemma 3.1. *Sia T il costo di esecuzione dell'algoritmo di moltiplicazione qui descritto per il D-BSP$(N, g(x))$. Se $g(x) = x^\alpha$ con $0 < \alpha < 1$, allora:*

$$T(N) = \Theta\left(\begin{cases} N^\alpha & \text{se } \alpha > \frac{1}{2} \\ \sqrt{N}\log N & \text{se } \alpha = \frac{1}{2} \\ \sqrt{N} & \text{se } \alpha < \frac{1}{2} \end{cases}\right)$$

Se $g(x) = \log x$ allora $T = \Theta(\sqrt{N})$. In entrambi i casi i risultati sono ottimi.

3.1.2 Trasformata discreta di Fourier

Sia A un vettore con N elementi di cui vogliamo determinare la trasformata discreta di Fourier in un D-BSP$(N, \mathbf{g})$. L'algoritmo parallelo si basa sull'algoritmo a sei-passi di Cooley-Tukey descritto nel Paragrafo 2.2.3: il problema viene diviso in due round in ciascuno dei quali vengono risolte $\sqrt{N}$ trasformate di vettori di taglia $\sqrt{N}$; precedentemente ad ogni round e al termine della risoluzione di tutti i sottoproblemi viene eseguita una permutazione.

Distribuiamo gli elementi unformemente e arbitrariamente tra i processori. Ogni sottoproblema può essere risolto da un $\frac{\log N}{2}$-cluster e precedentemente ad ogni round eseguiamo una permutazione (0-superstep). L'algoritmo termina quando si raggiunge un $\log N - 1$-cluster in cui la trasformata di due elementi viene risolta banalmente utilizzando la definizione di trasformata di Fourier. Si ottiene quindi un algoritmo che utilizza solo cluster del tipo $i = \left(1 - \frac{1}{2^j}\right)\log N$ con $0 \leq j \leq \log\log N$ e $k_i = 2^{j+1}$. Infatti se $j = 0$ si ottiene $i = 0$ e $k_0 = 2$, ovvero i due 0-superstep iniziali, e se si suppone di essere in un $i = \left(1 - \frac{1}{2^j}\right)\log N$-cluster con $k_i = 2^{j+1}$, per quanto detto precedentemente, i prossimi cluster saranno del tipo $\frac{1}{2}(\log N + i) = \left(1 - \frac{1}{2^{j+1}}\right)\log N$ e $k_{\left(1-\frac{1}{2^{j+1}}\right)\log N} = 2^{j+1}$ perché l'algoritmo esegue sequenzialmente due sottoproblemi per ogni $\frac{1}{2}(\log N + i)$-cluster. Quindi:

$$k_i = \begin{cases} 2^{j+1} & \text{se } i = \frac{1}{2}(\log N + j) \\ 0 & \text{altrimenti} \end{cases} \quad \forall\ 0 \leq j \leq \log\log N \tag{3.5}$$

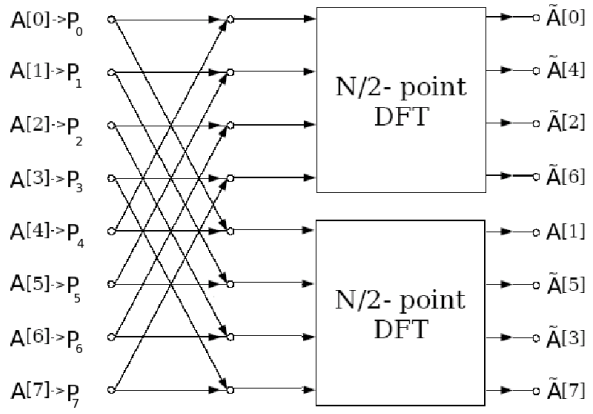

Figura 3.2: Percorso dei dati nella trasformata discreta di Fourier per $N = 8$ processori. Il processore P_i riceve all'inizio il dato $A[i]$; $\tilde{A[i]}$ è l'elemento i-esimo della trasformata del vettore A.

Il superstep iniziale (per distribuire in maniera ottimale i messaggi) e quello di sincronizzazione finale sono asintoticamente trascurabili. È inoltre evidente che $\mu = \Theta(1)$ e $\tau = \Theta(1)$.

Lemma 3.2. *Sia T il costo di esecuzione dell'algoritmo di moltiplicazione ricorsivo qui descritto per il D-BSP$(N, g(x))$. Se $f(x) = x^\alpha$ con $0 < \alpha < 1$, allora si ottiene il valore ottimo $T = \Theta(N^\alpha)$; se $g(x) = \log x$ allora $T = O(\log N \log\log N)$, che coincide con il miglior tempo noto in letteratura.*

Un altro semplice algoritmo si ottiene risolvendo il *dag* (direct acyclic graph) della trasformata discreta di Fourier. Come si vede dalla Fig. 3.2, ogni processore P_h di un i-cluster comunica con il processore P_k tale che $|h - k| = 2^{i-1}$, ovvero la prima metà del cluster comunica con la seconda metà. Ogni processore esegue un i-superstep $\Theta(1)$ volte.

Lemma 3.3. *Sia T il costo di esecuzione dell'algoritmo basato sul DFT dag qui descritto per il D-BSP$(N, g(x))$. Se $g(x) = x^\alpha$ con $0 < \alpha < 1$, allora $T = \Theta(N^\alpha)$; se $g(x) = \log x$ allora $T = O(\log^2 N)$. Solo nel primo caso si ottiene un risultato ottimo.*

3.1.3 Ordinamento: k-sorting

Il k-sorting consiste nell'ordinare kN elementi contenuti in N nodi, ognuno dei quali ne contiene k: al termine dell'algoritmo il processore P_0 deve contenere i primi k elementi più piccoli, P_1 i successivi k elementi più piccoli, ..., e P_{N-1} i k elementi più grandi.

L'algoritmo di base è la simulazione di bitonic sort su un ipercubo di dimensione $\log N$ in cui il confronto tra elementi contenuti nei processori avviene attraverso un merge; più

dettagliamente in una comunicazione tra due processori P_a e P_b ($a < b$), ognuno dei quali contiene k elementi *ordinati*, avviene la seguente procedura:

- P_b comunica i k elementi (ordinati) a P_a;
- P_a fonde le due sequenze di k elementi in $\Theta(k)$;
- P_a comunica i k elementi più grandi a P_b mantenendo l'ordine.

Se inseriamo i processori a distanza j, $0 \leq j < \log N$, in un $i = \log N - j - 1$-cluster, ogni i-superstep viene eseguito $i + 1$ volte poiché ogni processore comunica con un processore a distanza j un numero di volte pari a $\Theta(\log N - j) = \Theta(i+1)$. In ogni superstep, ogni processore esegue $\Theta(k)$ operazioni (in realtà solo un processore esegue le operazioni mentre l'altro rimane in attesa), per un tempo totale di calcolo $T = \Theta\left(k \sum_{i=0}^{\log N}(i+1)\right) = \Theta(k \log^2 N)$. A questo bisogna aggiungere $\Theta(k \log k)$ operazioni per l'ordinamento locale iniziale, ma se k è polinomiale rispetto a N tale incremento è trascurabile. È evidente che ogni processore avrà un contesto di taglia $\Theta(k)$.

Lemma 3.4. *Sia T il costo di esecuzione dell'algoritmo basato su bitonic sort qui descritto per il D-BSP$(N, g(x))$ e $k = O(1)$. Se $g(x) = x^\alpha$ con $0 < \alpha < 1$, allora si ottiene l'ottimo $T = \Theta(N^\alpha)$; se $g(x) = \log x$ allora $T = O(\log^2 N)$.*

3.2 Simulazione su HMM

Indichiamo con *guest* il modello D-BSP$(N, g(x))$ e con *host* la macchina HMM$(f(x))$ su cui verrà simulato l'algoritmo. Supponiamo che la funzione $f(x)$ sia regolare, ovvero $\forall x$ $f(2x) < cf(x)$ con $c \geq 1$.

La memoria dell'host è suddivisa in blocchi di dimensione μ ordinati in modo che l'i-esimo blocco contenga il contesto del processore P_i (nel corso della simulazione tale configurazione può cambiare). Trasformiamo il programma $\mathcal{P}$ del guest in un programma $\mathcal{L}$-smooth.

Definizione 3.1. *Sia $\mathcal{L}$ un insieme di etichette di superstep $0 = l_0 < l_i < \ldots < l_m = \log N$. Un programma $\mathcal{P}$ per D-BSP$(N, g(x))$ è $\mathcal{L}$-smooth se: 1) ogni superstep di $\mathcal{P}$ ha etichetta in $\mathcal{L}$; 2) se un superstep di etichetta l_i segue un l_j-superstep con $l_j > l_i$, allora i=j-1.*

Un programma $\mathcal{P}$ si può sempre trasformare in un programma $\mathcal{L}$-smooth, ma il passaggio potrebbe aumentare il tempo di esecuzione di una quantità non trascurabile. È possibile costruire un insieme $\mathcal{L}'$ tale che per due opportuni valori $0 < c_1 \leq c_2 < 1$: 1) $g\left(\frac{\mu N}{2^{l_{i+1}}}\right) \geq c_1 g\left(\frac{\mu N}{2^{l_i}}\right)$ $\forall\ 0 \leq i < m$; 2) $g\left(\frac{\mu N}{2^{l_{i+1}}}\right) \leq c_2 g\left(\frac{\mu N}{2^{l_i}}\right)$ $\forall\ 0 \leq i < m$. Trasformare $\mathcal{P}$ in un programma $\mathcal{L}'$-smooth aumenta il tempo di esecuzione di un fattore trascurabile.

L'algoritmo di simulazione è descritto dal seguente pseudocodice:

Algorithm 1: `Simulation(`$\mathcal{P}$`)`

```
1.1  while true do
1.2      P ← processore il cui contesto è nella posizione iniziale della memoria;
1.3      s ← numero del prossimo superstep da simulare per P;
1.4      C ← i_s-cluster contenente P;
1.5      Simula il superstep s per C;
1.6      if P ha concluso 𝒫 then return;
1.7      if i_{s+1} < i_s then
1.8          b ← 2^{i_s − i_{s+1}};
1.9          Sia Ĉ l'i_{s+1}-cluster contenente C e siano Ĉ_0 ... Ĉ_{b−1} gli i_s-cluster contenuti in
             Ĉ;
1.10         if j > 0 then scambia i contesti di C con quelli di Ĉ_0;
1.11         if j < b − 1 then scambia i contesti di C_0 con quelli di Ĉ_{j+1};
```

L'esecuzione del superstep s del i_s-cluster C (riga 1.5) avviene portando iterativamente ogni contesto di C in testa alla memoria e simulandone il contenuto. Le comunicazioni vengono simulate in modo banale, cioè consegnando iterativamente i messaggi di ogni processore.

Per la dimostrazione completa della correttezza e della complessità dell'algoritmo rimandiamo a [16], ricordando però che la validità si basa sui seguenti invarianti validi per ogni iterazione del ciclo while alla riga 1.1 (*round*):

Invariante 3.1. *C è s-ready;*

Invariante 3.2. *I contesti di tutti i processori di C si trovano nei primi $|C|$ blocchi ordinati rispetto l'indice del processore. Inoltre, per ogni altro cluster C' tutti i processori sono allocati in blocchi contigui sebbene non necessariamente ordinati.*

Definizione 3.2. *Un cluster C è s-ready se ogni processore di C ha eseguito i primi $s-1$ superstep, ma non il superstep s.*

Dall'analisi dell'algoritmo si ottiene il seguente teorema:

Teorema 3.5. *Sia $\mathcal{P}$ un programma fine-grained per D-BSP$(N, g(x))$ con λ_i i-superstep. Siano inoltre τ e T rispettivamente il tempo di computazione in ogni superstep e in tutto il programma. Se $f(x)$ è regolare, allora $\mathcal{P}$ può essere simulato su un HMM$(f(x))$ con $\Theta(N)$ memoria in tempo:*

$$O\left(N\sum_{i=0}^{\log N}\lambda_i\left(\tau+\mu f\left(\frac{\mu N}{2^i}\right)\right)\right)=O\left(N\left(T+\mu\sum_{i=0}^{\log N}\lambda_i f\left(\frac{\mu N}{2^i}\right)\right)\right). \tag{3.6}$$

Corollario 3.6. *Se $f(x)$ è regolare, allora ogni programma $\mathcal{P}$ per D-BSP$(N, f(x))$ che richiede tempo $\tilde{T}$ può essere simulato in tempo $\Theta(N\tilde{T})$ su un HMM$(f(x))$.*

Applicando il corollario appena enunciato agli algoritmi paralleli descritti nel Paragrafo 3.1 si ottengono i seguenti risultati:

Lemma 3.7. *Gli algoritmi ottenuti tramite la simulazione dei programmi paralleli descritti nel Paragrafo 3.1 hanno le seguenti complessità nel HMM(x^α), $0 < \alpha < 1$:*

- *L'algoritmo ottenuto dalla moltiplicazione parallela di matrici è ottimo e ha complessità*

$$T(N) = \Theta \begin{cases} N^{\alpha+1} & \text{se } \alpha > \frac{1}{2} \\ N\sqrt{N}\log N & \text{se } \alpha = \frac{1}{2} \\ N\sqrt{N} & \text{se } \alpha < \frac{1}{2} \end{cases} ;$$

- *Gli algoritmi ottenuti dai due programmi paralleli per DFT sono ottimi e hanno complessità $\Theta(N^{\alpha+1})$;*

- *L'algoritmo ottenuto dell'ordinamento parallelo è ottimo e ha complessità $\Theta(N^{\alpha+1})$.*

Lemma 3.8. *Gli algoritmi ottenuti tramite la simulazione dei programmi paralleli descritti nel Paragrafo 3.1 hanno le seguenti complessità nel HMM($\log x$):*

- *L'algoritmo ottenuto dalla moltiplicazione parallela di matrici è ottimo e ha complessità $T(N) = \Theta(N\sqrt{N})$;*

- *L'algoritmo ottenuto dalla simulazione dell'algoritmo per la DFT basato sulla scomposizione in $\sqrt{N}$ sottoproblemi è ottimo e ha complessità $T(N) = \Theta(N \log N \log\log N)$;*

- *L'algoritmo ottenuto dell'ordinamento parallelo non è ottimo e ha complessità $T(N) = O(N \log^2 N)$*

Il risultato non ottimo nella simulazione dell'ordinamento su HMM($\log x$) è dovuto alla scelta dell'algoritmo parallelo. La simulazione, comunque, riesce in tutti gli altri casi a trasformare la località di comunicazione del D-BSP nella località di riferimento dell'HMM.

Un analogo al lemma di Brent

Per poter gestire l'auto-simulazione è necessario che ogni nodo del D-BSP possa gestire contesti non costanti. Consideriamo ogni processore di un D-BSP($N, g(x)$) come un HMM($g(x)$) di taglia μ. Chiamiamo *guest* il D-BSP($N, g(x)$) da simulare e *host* il D-BSP($M, g(x)$) su cui avverà la simulazione ($M \leq N$).

La procedura per eseguire un i-superstep sul host varia a seconda che i sia minore o non minore di $\log M$. Consideriamo superstep del primo tipo. In questo caso un i-cluster è distribuito su più nodi del host e quindi la simulazione è composta da un i-superstep seguito da un $\log M$-superstep del host. Infatti ogni processore deve simulare gli $\frac{N}{M}$ contesti contenuti, inviare i messaggi all'interno del i-cluster del guest, ovvero di un i-cluster dell'host, ed infine distribuire i messaggi ricevuti tra i contesti contenuti.

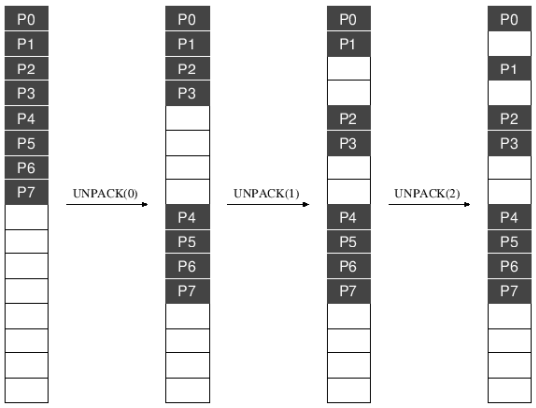

Figura 3.3: Layout dopo l'applicazione di `unpack(0)` con 8 processori.

Consideriamo ora i-superstep con $i \geq \log M$. La simulazione del superstep coincide con la simulazione su un singolo processore di un $i - \log M$-superstep per D-BSP$\left(\frac{N}{M}, g(x)\right)$. Infatti il $i - \log M$-cluster di un D-BSP$\left(\frac{N}{M}, g(x)\right)$ contiene $\frac{N/M}{2^{i-\log M}} = \frac{N}{2^i}$ nodi.

Si può dimostrare che:

Teorema 3.9. *Sia $\mathcal{P}$ un programma per D-BSP$(N, g(x))$ con λ_i i-superstep. Siano inoltre τ e T rispettivamente il tempo di computazione in ogni superstep e in tutto il programma. Se $g(x)$ è regolare, allora $\mathcal{P}$ può essere simulato su un D-BSP$(M, g(x))$ con $\Theta\left(\frac{\mu N}{M}\right)$ memoria per nodo, $1 \leq M \leq N$, in tempo:*

$$O\left(\frac{N}{M}\sum_{i=0}^{\log N}\lambda_i\left(\tau + \mu g\left(\frac{\mu N}{2^i}\right)\right)\right) = O\left(\frac{N}{M}\left(T + \mu\sum_{i=0}^{\log N}\lambda_i g\left(\frac{\mu N}{2^i}\right)\right)\right) \tag{3.7}$$

Se inoltre $\mathcal{P}$ è *full*, ovvero le comunicazioni sono delle $\Theta(\mu)$-relazioni, vale il seguente corollario, analogo al lemma di Brent [8]:

Corollario 3.10. *Se $\mathcal{P}$ è full e richiede tempo $\tilde{T}$ su un D-BSP$(N, g(x))$, allora la sua simulazione su un D-BSP$(M, g(x))$ con contesto $\Theta\left(\frac{\mu N}{M}\right)$ per ogni nodo, $1 \leq M \leq N$, richiede tempo $\Theta\left(\frac{N}{M}\tilde{T}\right)$.*

3.3 Simulazione su BT

Supponiamo che la funzione di accesso alla memoria del BT sia $f(x) = O(x^\alpha)$, $0 < \alpha < 1$, e regolare e vi sia almeno $\Theta(N \log\log N)$ memoria. L'algoritmo di simulazione descritto nel capitolo precedente per l'HMM è un programma valido per il BT, ma non riesce a utilizzare al meglio le caratteristiche del modello. Il BT non è in grado di scambiare due blocchi

di memoria sovrapposti e perciò è necessario inserire dello spazio libero. Più esattamente: durante la simulazione di un cluster, posizionato in testa alla memoria, è necessario inserire adiacentemente a questo dello spazio libero della stessa dimensione del cluster. A tal fine vengono definite due procedure `pack(`i`)` e `unpack(`i`)`. La prima richiede che un i-cluster sia allocato in testa alla memoria e seguito da $\Theta\left(\frac{\mu N}{2^i}\right)$ spazio libero; al termine della chiamata a `pack(`i`)`, lo spazio libero all'interno del cluster sarà distribuito come in Fig. 3.3. La procedura `unpack(`i`)`, invece, ricomprime lo spazio libero distribuito da `pack(`i`)`. Posizionando un `pack(`i_s`)` ed un `unpack(`i_s`)` rispettivamente all'inizio e alla fine di un round definito nell'Algoritmo 1, si eviterà l'impossibilità di effettuare scambi di blocchi sovrapposti. La complessità non cambia perché l'indirizzo di un dato al più raddoppia e $f(x)$ è regolare.

Diversamente dal HMM, la simulazione banale di un cluster non è efficiente perché non sfrutta la capacità del BT di spostare blocchi di dati efficientemente. Nel BT la simulazione di un cluster si divide in due fasi: la prima sposta iterativamente in testa alla memoria blocchi di c processori adiacenti (c è funzione di $f(x)$) e ricorsivamente ne simula i contesti; la seconda, invece, distribuisce i messaggi con un algoritmo di ordinamento (*Approx-Median-Sort* descritto in [11]) che richiede $\Theta\left(\frac{\mu N}{2^i}\log\log\frac{\mu N}{2^i}\right)$ spazio ausiliario per un i-cluster. Più dettagliamente il cluster viene diviso in $\Theta\left(\mu\frac{N}{2^i}\right)$ blocchi di dimensione $\Theta(1)$ ed ordinati utilizzando un'opportuna chiave la quale garantisce che i messaggi vengano distribuiti correttamente e i dati locali non persi. Poiché lo spazio ausiliario richiesto dall'ordinamento è maggiore di quello disponibile adiacentemente al cluster, viene utilizzata la funzione `pack()` al fine di aumentare tale spazio libero. Infine, per ristabilire il layout in memoria presente prima dell'ordinamento, viene chiamata la funzione `align()` per riallineare i contesti.

Utilizzando la simulazione presentata è possibile ottenere il seguente risultato:

Teorema 3.11. *Sia $\mathcal{P}$ un programma fine-grained per D-BSP$(N, g(x))$ con λ_i i-superstep. Siano inoltre τ e T rispettivamente il tempo di computazione in ogni superstep e in tutto il programma. Se $f(x)$ è regolare, allora $\mathcal{P}$ può essere simulato su un BT$(f(x))$ con $\Theta(N\log\log N)$ memoria in tempo:*

$$O\left(N\sum_{i=0}^{i=\log N}\lambda_i\left(\tau+\mu\log\left(\frac{\mu N}{2^i}\right)\right)\right)=O\left(N\left(T+\sum_{i=0}^{i=\log N}\lambda_i\log\left(\frac{\mu N}{2^i}\right)\right)\right) \tag{3.8}$$

A parte il termine NT, l'ordinamento usato per le comunicazioni domina l'andamento della simulazione. Inoltre è interessante notare che il tempo non dipende da $f(x)$ in accordo con quanto affermato in [3], cioè che un buon algoritmo per BT riesce a nascondere il tempo di accesso del modello.

Applicando il teorema precedente agli algoritmi descritti nel Paragrafo 3.1 si ottengono i seguenti risultati per un modello BT.

Lemma 3.12. *L'algoritmo descritto nel Paragrafo 3.1.1 per la moltiplicazione di matrici richiede tempo ottimo $O\left(N\sqrt{N}\right)$ per un BT$(f(x))$ con $f(x)=x^\alpha$, $0<\alpha<1$, o $f(x)=\log x$.*

Lemma 3.13. *L'algoritmo basato sul DFT dag e quello basato sulla divisione in $\sqrt{N}$ sottoproblemi, descritti nel Paragrafo 3.1.2 richiedono rispettivamente tempo $O(N \log^2 N)$ e $O(N \log N \log\log N)$ in un BT($f(x)$) con $f(x) = x^\alpha$, $0 < \alpha < 1$, o $f(x) = \log x$.*

Gli algoritmi per D-BSP della moltiplicazione di matrici e della trasformata di Fourier restituiscono programmi (quasi) ottimi per entrambe le bande del BT analizzate. La scelta di progettare algoritmi D-BSP con parametro di banda $f(x)$ per ottenere algoritmi ottimi su un BT($f(x)$) non risulta vincente: le simulazioni dei due algoritmi paralleli per la DFT, che hanno la stessa complessità su un D-BSP(N, x^α), hanno comportamenti diversi su un BT(x^α). Una buona scelta, invece, è porre la banda del D-BSP pari a $\log x$; in questo modo in un D-BSP l'algoritmo basato sul DFT dag risulta peggiore di quello basato sulle $\sqrt{N}$ chiamate ricorsive, lo stesso avviene nel BT(x^α). Inoltre con un D-BSP($N, \log x$) si ottiene che la simulazione ha slowdown lineare $\Theta(N)$; quindi se A_1 e A_2 sono due algoritmi per il D-BSP che risolvono lo stesso problema e la simulazione su BT di A_1 è migliore di quella di A_2, allora A_1 è più efficiente di A_2.

Capitolo 4

Dal modello CD-BSP al modello a CM

Nel capitolo precedente sono stati descritti i risultati della simulazione da N a $M < N$ nodi di un modello D-BSP costituito da processori con gerarchia di memoria HMM e successivamente BT. Di seguito verrà presentata la procedura di simulazione applicata ad un modello D-BSP composto da processori con cache ideale a due livelli (CD-BSP).

La simulazione utilizza l'ordinamento per la distribuzione dei messaggi, che garantisce la generalità degli algoritmi da simulare e la possibilità di definire il pattern di comunicazione a tempo di esecuzione. Inoltre la simulazione è cache-oblivious.

Il capitolo è così strutturato: nel Paragrafo 4.1 verrà descritto il modello CD-BSP; nel Paragrafo 4.2 verrà presentato l'algoritmo di simulazione su un singolo processore e dimostrata la sua correttezza; nel Paragrafo 4.3 verrà analizzata la complessità RAM e il numero di miss della simulazione; infine nel Paragrafo 4.4 verrà presentata una versione analoga del lemma di Brent.

4.1 Il modello CD-BSP

Il CD-BSP($N, \mathbf{g}, \mathbf{l}, Z, L, t$) (*Cached D-BSP*) è costituito da un modello D-BSP($N, \mathbf{g}, \mathbf{l}$) in cui ogni nodo contiene una gerarchia a due livelli CM(Z, L, t). Un programma $\mathcal{P}$ per CD-BSP è equivalente ad un programma per D-BSP poiché la cache è trasparente, ovvero non può essere controllata direttamente dal programmatore. Quindi $\mathcal{P}$ è costituito da k_i i-superstep, $0 \leq i \leq \log N$, ed ogni i-superstep richiede tempo $\tilde{\tau}$:

$$\tilde{\tau} = \tau + Q_\tau(Z, L)t + hg_i + l_i, \tag{4.1}$$

dove: τ (complessità RAM) è il tempo di computazione di un processore in un modello RAM, $Q_\tau(Z, L)$ è il numero di miss nella computazione interna, t è il tempo di latenza di un miss, h è il numero di messaggi ricevuti e inviati, infine g_i e l_i sono rispettivamente l'inverso della banda (*parametro di banda*) e la latenza in un i-cluster. Supponiamo che

in ogni superstep il tempo RAM, il numero di miss e il numero di messaggi siano limitati superiormente rispettivamente da τ, $Q_\tau(Z, L)$ ed h e che l_i sia trascurabile rispetto ad hg_i. Il tempo di esecuzione di $\mathcal{P}$ è:

$$\tilde{T} = O\left(\sum_{i=0}^{\log N} k_i \left(\tau + Q_\tau(Z, L)t + hg_i\right)\right) \tag{4.2}$$

che ricorda il valore trovato nel modello D-BSP semplice, ad eccezione del numero di miss.

Per ogni processore definiamo il valore μ che indica la dimensione del blocco di memoria che contiene i dati caratterizzanti il suo stato in un certo istante (*contesto*). Supponiamo che ogni contesto abbia soglia μ e che $\tau = \Omega(\mu)$, ovvero che la computazione utilizzi tutto il contesto, e che $h = \Theta(\mu)$ (*full relation*).

Si può inoltre supporre che il peso dei cold miss nella computazione locale, $O\left(\left(1 + \frac{\mu}{L}\right) t\right)$, possa essere trascurato rispetto al peso delle comunicazioni, ovvero che:

$$g_i = \Omega\left(\left(1 + \frac{\mu}{L}\right) \frac{t}{h}\right), \ \forall\, 0 \leq i < \log N. \tag{4.3}$$

In questo modo i cold miss, inevitabili per definizione, non penalizzano la simulazione e l'attenzione può essere così focalizzata sui capacity miss. Questa assunzione è giustificata anche in pratica perché nelle macchine general pourpose, in genere, la comunicazione avviene tramite scheda di rete, la quale comunica solo con la memoria e l'accesso a dati contigui avviene in maniera non più efficiente rispetto al processore. Con questa assunzione non ha più importanza se, tra un superstep ed il successivo, la cache viene invalidata dalle comunicazioni (come dovrebbe avvenire secondo le ipotesi appena enunciate) poiché il costo aggiuntivo della ricostruzione della cache invalidata è $O\left(\left(1 + \frac{\mu}{L}\right) t\right) = O(hg_i)$ ed è quindi trascurabile rispetto alla comunicazione appena effettuata.

Sotto tale ipotesi, il CD-BSP$(N, \mathbf{g}, \mathbf{l}, Z, L, t)$ diventa un D-BSP$(N, \mathbf{g}, \mathbf{l})$ quando $Z = \Omega(\mu)$: all'inizio di ogni superstep sono necessari solo $O\left(1 + \frac{\mu}{L}\right)$ cold miss in ogni processore e il tempo di uno i-superstep è quindi

$$\tilde{\tau} = O\left(\tau + \left(1 + \frac{\mu}{L}\right) t + \mu g_i\right) = O\left(\tau + \mu g_i\right). \tag{4.4}$$

pari al tempo di un D-BSP. Questa assunzione ci permette di analizzare algoritmi per D-BSP su CD-BSP anche quando i miss non sono di interesse (basta porre $Z = \Omega(\mu)$); se però entrano in gioco capacity miss, la (4.4) non è più valida e il modello permette di analizzare l'efficienza nell'utilizzo della cache. Inoltre se $N = 1$, il CD-BSP diventa il modello di cache ideale descritto precedentemente.

Gli algoritmi per la moltiplicazione di matrici, la trasformata di Fourier e l'ordinamento per D-BSP descritti precedentemente sono caratterizzati da contesti $\Theta(1)$ e quindi si avranno $\Theta(1)$ miss per superstep e il tempo $\tilde{T}$ totale sarà pari a quello calcolato nel D-BSP.

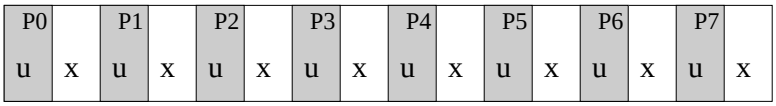

Figura 4.1: Contesto della simulazione per 8 processori (u: contesto, x: spazio aggiuntivo)

4.2 Algoritmo di simulazione

Vogliamo ora simulare un algoritmo $\mathcal{P}$ con contesto μ per il modello CD-BSP$(N, \mathbf{g}', Z', L', t')$ su una macchina CD-BSP$(1, \mathbf{g}, Z, L, t)$, ovvero su CM(Z, L, t). Il contesto della simulazione è costituito dai contesti μ dei nodi P_k, $0 \leq k < N$, ordinati in ordine cresente rispetto all'indice del processore, ognuno dei quali è seguito da $\Theta(\mu)$ spazio aggiuntivo che verrà utilizzato nella simulazione della comunicazione (vedi Fig. 4.1). Di seguito indicheremo con $\tilde{\mu}$ il contesto costituito da μ e dallo spazio aggiuntivo; evidentemente $\tilde{\mu} = \Theta(\mu)$. Senza perdita di generalità si può supporre che i messaggi in ingresso e uscita siano raccolti in un buffer alla fine di μ.

L'algoritmo di simulazione si evolve secondo le stesse linee dell'algoritmo di [16] per la simulazione su BT:

Algorithm 2: `Simulation(`$\mathcal{P}$`)`

```
   // Algoritmo di simulazione;
   // i_s indica il tipo (label) dell's-esimo superstep
2.1 P ← 0 primo processore da simulare;
2.2 while true do
2.3     s ← indice del prossimo superstep da simulare per P;
2.4     C ← i_s-cluster contenente P;
2.5     ClusterSimulation(C);
2.6     if P ha concluso then
2.7         return;
2.8     if i_{s+1} < i_s then
2.9         Sia Ĉ l'i_{s+1}-cluster contenente C, con Ĉ = Ĉ_0 ... Ĉ_{2^{i_s - i_{s+1}} - 1};
2.10        dove Ĉ_i indicano gli i_s-cluster in Ĉ e C = Ĉ_j per un certo j;
2.11        if j = i_s - i_{s+1} + 1 then
2.12            P ← P + N/2^{i_s} - N/2^{i_{s+1}} (processore con indice più basso di Ĉ);
2.13        else
2.14            P ← P + N/2^{i_s} (processore con indice più basso di Ĉ_{j+1});
```

Algorithm 3: `ClusterSimualtion(`C`)`

```
   // Algoritmo per l'esecuzione del cluster C;
3.1 begin
3.2 |  Compute(C);
3.3 |  Communication(C);
3.4 end
```

Algorithm 4: `Compute(`C`)`

```
   // Algoritmo per l'esecuzione dei contesti di C;
4.1 begin
4.2 |  i ← label del cluster C;
4.3 |  if i = log N then
4.4 |  |  Simula il contenuto dell'unico processore di C;
4.5 |  |  return
4.6 |  Sia C = Ĉ_0 Ĉ_1, dove Ĉ_i sono (i+1)-cluster;
4.7 |  Compute(Ĉ_0);
4.8 |  Compute(Ĉ_1);
4.9 end
```

Algorithm 5: `Communication(`C`)`

```
   // Algoritmo per le comunicazioni;
5.1 begin
5.2 |  if label di C = log N then
    |  |  // Se ho un solo processore non ho comunicazioni
5.3 |  |  return;
5.4 |  PackAndTag(C);
5.5 |  Ordina tutti i blocchi del cluster C;
5.6 |  UnpackAndUntag(C);
5.7 end
```

Algorithm 6: `PackAndTag(`C`)`

```
   // Crea lo spazio per l'ordinamento e inserisce le chiavi;
6.1 begin
      // L'algoritmo comprime i contesti μ alla fine del cluster;
      // #(C) numero blocchi in C, dimBlocco dimensione del blocco in
      // word, dimEtichetta dimensione etichetta in word.  I blocchi sono
      // numerati da 0 a #(C) − 1.
6.2   l ← ultimo blocco di μ;
6.3   m ← indirizzo dell'ultima word del (#(C) − 1)-esimo blocco;
6.4   while Finché tutti i blocchi di μ non sono stati spostati do
6.5       Etichetta blocco l;
6.6       m ← m − dimBlocco − dimEtichetta;
6.7       Copia il blocco l e la rispettiva chiave a partire dalla word m;
6.8       l ← blocco precedente a quello appena spostato;
      // m contiene l'indirizzo della word del primo blocco dei contesti μ
         etichettati;
6.9   l ← primo blocco e rispettiva chiave del contesto appena spostato;
6.10  m ← indirizzo della prima word del cluster;
6.11  while Finché tutti i blocchi non sono stati spostati all'inizio del cluster do
6.12      Copia il blocco l in m;
6.13      m ← m + dimBlocco + dimEtichetta;
6.14      l ← blocco successivo a quello appena spostato e rispettiva chiave;
6.15 end
```

Teorema 4.1. *L'algoritmo di simulazione è corretto.*

Dim. La dimostrazione ricalca abbastanza fedelmente quella proposta in [16] per il modello BT. Poiché nella simulazione non vengono fatte ipotesi sul modello utilizzato, la dimostrazione può essere effettuata utilizzando un modello RAM.

Innanzitutto verifichiamo la correttezza della simulazione di un singolo i-cluster con $\frac{N}{2^i}$ processori, allocato in $\Theta(\frac{\tilde{\mu}N}{2^i})$ locazioni contigue in memoria, che coincide con la simulazione di un programma per CD-BSP con $\frac{N}{2^i}$ processori e costituito da un solo 0-superstep. L'esecuzione dei contesti avviene ricorsivamente dividendo lo i-cluster in due $i+1$-cluster finché non si ottiene un $\log N$-cluster, ovvero un solo processore. La correttezza di questo passo è evidente in quanto l'algoritmo simula sequenzialmente le attività dei vari processori.

Successivamente, lo scambio dei messaggi all'interno del cluster avviene con l'ordinamento: ogni contesto originale (ovvero μ) viene diviso in blocchi[1] di dimensione *dimBlocco* word, univocamente rappresentati da una tripletta (x, y, z) (etichetta o chiave). Per la struttura di μ, i blocchi all'inizio del contesto di ogni processore conterranno solo dati da *non* spedire,

[1]Il termine blocco *non* fa riferimento al concetto di blocco di una cache, altrimenti si perderebbe la proprietà di cache-obliviousness.

mentre nella parte finale dello stesso, che coincide con il buffer dei messaggi, ogni blocco conterrà parte di un solo messaggio. La chiave deve garantire che, dopo l'ordinamento, i dati da non inviare siano nella propria posizione iniziale e che i messaggi vengano consegnati ai rispettivi destinatari. Una possibile n-pla è la seguente:

- x: processore a cui appartiene il blocco;
- y: indice del blocco nel contesto;
- z: $\begin{cases} \text{destinatario, se il blocco contiene un messaggio;} \\ x \text{ altrimenti (il destinatario è il mittente).} \end{cases}$

Un ordinamento che garantisce una corretta distribuzione dei messaggi è:

$$\begin{aligned} (x,y,z) > (x',y',z') \iff & (z > z') \\ & \vee((z = z') \wedge (z = x) \wedge (z' = x') \wedge (y > y)') \\ & \vee((z = z') \wedge (z \neq x) \wedge (z' = x')) \\ & \vee((z = z') \wedge (z \neq x) \wedge (z' \neq x') \wedge (x > x')); \\ (x,y,z) = (x',y',z') \iff & ((x = x') \wedge (y = y') \wedge (z = z')). \end{aligned} \tag{4.5}$$

Ovvero se $P = (x,y,z)$ e $Q = (x',y',z')$ allora $P > Q$ (il blocco P è memorizzato di seguito a Q) se e solo se:

- i destinatari dei due blocchi sono diversi e $z > z'$;
- i blocchi contengono dati (non messaggi) dello stesso contesto e $y > y'$;
- il blocco Q contiene un dato locale di z e P contiene un messaggio per z;
- entrambi i blocchi sono messaggi per lo stesso processore ma i mittenti sono distinti e $x > x'$.

È necessario quindi dello spazio aggiuntivo per l'etichetta di ogni blocco e, supponendo che le dimensioni di un blocco e di una chiave siano $\Theta(1)$ word [15], la memoria supplementare per ogni contesto è $\Theta(\mu)$. Per le ipotesi iniziali tale memoria è disponibile all'interno di $\tilde{\mu}$. Anche l'algoritmo di ordinamento richiede $\Theta(\frac{\mu N}{2^i})$ locazioni contigue libere. Se il contesto $\tilde{\mu}$ ha una dimensione opportuna, tale memoria è ottenibile comprimendo all'inizio del cluster i contesti originali etichettati. La funzione `PackAndTag(`C`)` estrae tale spazio e contemporaneamente inserisce le chiavi: prima i contesti μ vengono riallocati alla fine del cluster spostando iterativamente i blocchi e contemporaneamente inserendo le chiavi; poi i contesti etichettati vengono riportati all'inizio del cluster. Al termine dell'ordinamento i messaggi saranno consegnati correttamente per le proprietà della chiave, ma sarà necessario eliminare le chiavi ed allineare i contesti come definito nella Fig. 4.1. La procedura `UnpackAndUntag(`C`)`, l'inverso di `PackAndTag(`C`)`, ristabilirà il layout originale, eventualmente inserendo dei blocchi vuoti per allineare i contesti.

La correttezza dell'intero algoritmo è garantita dal verificarsi del seguente invariante dove con *round* intendiamo una singola iterazione del ciclo *while* della riga 2.2:

Invariante 4.1. *All'inizio di ogni round, il cluster da simulare C è s-ready.*

All'interno della dimostrazione si considerano programmi per D-BSP validi anche quelli eventualmente privi dello 0-superstep finale. Dimostriamo la validità dell'invariante per induzione sul numero di processori N. Per $N = 1$ la validità è evidente. Supponiamo sia vero per $N < N'$ e dimostriamo la validità per $N = 2N'$. All'inizio della simulazione ogni i_s-cluster è evidentemente 1-ready. Supponiamo per ipotesi che nell's-esimo round C sia s-ready e dimostriamo che nel successivo ciclo il cluster C' da simulare è $s+1$-ready. Durante l's-esimo round il cluster C esegue il superstep s (riga 2.5) e quindi al termine del round è $s+1$-ready. Se $i_{s+1} \geq i_s$ il cluster C' è contenuto in C e dunque al termine del ciclo è $s+1$-ready. Se, invece, $i_{s+1} < i_s$, l'algoritmo esegue i rimanenti $2^{i_s - i_{s+1}} - 1$ i_s-cluster e sia $\tilde{C}$ uno di questi. $\tilde{C}$ è r-ready e i superstep compresi tra r (incluso) ed s (escluso) sono di indice maggiore di i_s, altrimenti il cluster C non sarebbe s-ready perché C e $\tilde{C}$ appartengono allo stesso i-cluster con $i \leq i_s$. L'algoritmo esegue i rimanenti passi che formano un programma $\tilde{\mathcal{P}}$ per un D-BSP con $\frac{N}{2^{i_s+1}} < N$ nodi. Per l'ipotesi induttiva su N, terminata la simulazione di $\tilde{\mathcal{P}}$, $\tilde{C}$ sarà $s+1$ ready. Terminati tutti i possibili $\tilde{C}$, il successivo cluster C' sarà $s+1$-ready perché composto da nodi $s+1$-ready.

L'esattezza della simulazione è evidente: per ipotesi il cluster dell'ultimo round è sempre uno 0-cluster e per l'invariante è $|\mathcal{P}|$-ready; dopo l'esecuzione dell'ultimo superstep, ogni processore sarà $|\mathcal{P}|+1$-ready e $\mathcal{P}$ completato.

□

4.3 Analisi temporale

Teorema 4.2. *Sia $\mathcal{P}$ un programma per CD-BSP$(N, \boldsymbol{g}', Z', L', t')$ con contesto μ e k_i i-superstep. La sua simulazione su un CD-BSP$(1, \boldsymbol{g}, Z, L, t)$, ovvero un CM$(Z, L, t)$, richiede:*

$$T_{Sim}^{RAM}(\mathcal{P}) = O\left(N \sum_{i=0}^{\log N} k_i \left(\tau + \mu \log \mu \frac{N}{2^i}\right)\right), \tag{4.6}$$

$$Q_{Sim}(\mathcal{P}, Z, L) = O\left(1 + \frac{\mu N}{L} + N \sum_{i=0}^{\lambda-1} k_i \left(Q_\tau^*(Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right)\right), \tag{4.7}$$

$$\begin{aligned} T_{Sim}(\mathcal{P}, Z, L, t) = O\Bigg(\left(1 + \frac{\mu N}{L}\right) t \; &+ N \sum_{i=0}^{\log N} k_i \left(\tau + \mu \log \mu \frac{N}{2^i}\right) + \\ &+ N \sum_{i=0}^{\lambda-1} k_i \left(Q_\tau^*(Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right) t\Bigg) \end{aligned} \tag{4.8}$$

dove $Q^*_\tau(Z, L)$ e τ indicano rispettivamente il numero di capacity miss e la complessità RAM di un singolo processore e:

$$\tilde{\mu} = \Theta(\mu) \text{ } (\tilde{\mu} \text{ contiene un contesto } \mu \text{ e lo spazio aggiunto per la simulazione}),$$
$$\lambda = \min \left\{ \{\log N + 1\} \cup \{0 \leq i \leq \log N \ : \ Z - \tilde{\mu}\frac{N}{2^i} \geq 0\} \right\} \tag{4.9}$$

Dim. Analizziamo inizialmente il costo della simulazione nel modello RAM di un superstep C^i. La procedura `PackAndTag(`C^i`)` (e analogamente `UnpackAndUntag(`C^i`)`) è composta da due cicli *while* ognuno dei quali legge e sposta al più $O\left(\tilde{\mu}\frac{N}{2^i}\right)$ word sequenzialmente e quindi:

$$T^{RAM}_{Pack}(C^i) = \ O\left(\tfrac{\tilde{\mu}N}{2^i}\right) = O\left(\tfrac{\mu N}{2^i}\right). \tag{4.10}$$

La chiave può essere costruita in tempo costante perché si ottiene dalla concatenazione di tre valori noti: il mittente, il destinatario e l'indice del blocco.
Il costo della procedura `Communication(`C^i`)` dipende dalle due funzioni appena accennate e dall'algoritmo di ordinamento scelto; due algoritmi ottimi sono quelli presentati in [18]; infatti essi minimizzano il numero di miss e il tempo RAM. Poiché l'ordinamento avviene su $\Theta\left(\mu\frac{N}{2^i}\right)$ blocchi di dimensione $\Theta(1)$ che compongono i contesti μ, la complessità è:

$$T^{RAM}_{Ord}(C^i) = O\left(\mu\frac{N}{2^i}\log\mu\frac{N}{2^i}\right). \tag{4.11}$$

Perciò in totale, se il costo dell'unpack equivale a quello del pack, `Communication(`C^i`)` richiede:

$$T^{RAM}_{Comm}(C^i) = T^{RAM}_{Ord}(C^i) + 2T^{RAM}_{Pack}(C^i) = O\left(\mu\frac{N}{2^i}\log\mu\frac{N}{2^i}\right). \tag{4.12}$$

Il costo di `Compute(`C^i`)` segue la seguente elementare ricorrenza, dove τ è il costo della computazione locale di un processore:

$$\begin{aligned} T^{RAM}_{Comp}(C^i) &= \begin{cases} O(\tau) & \text{se } i = \log N \\ 2T(C^{i+1}) + \Theta(1) & \text{altrimenti} \end{cases} \\ &= O\left(\tau\frac{N}{2^i}\right). \end{aligned} \tag{4.13}$$

Quindi la simulazione di un superstep richiede tempo:

$$\begin{aligned} T^{RAM}_{ClSim}(C^i) &= T^{RAM}_{Comp}(C^i) + T^{RAM}_{Comm}(C^i) \\ &= O\left(\tau\frac{N}{2^i} + \mu\frac{N}{2^i}\log\mu\frac{N}{2^i}\right) \end{aligned} \tag{4.14}$$

Il primo addendo non è necessariamente trascurabile rispetto al secondo perché dipende dal particolare programma $\mathcal{P}$ simulato. In totale un programma $\mathcal{P}$ richiede tempo RAM pari a:

$$T^{RAM}_{Sim}(\mathcal{P}) = O\left(N\sum_{i=0}^{\log N} k_i\left(\tau + \mu\log\mu\frac{N}{2^i}\right)\right) \tag{4.15}$$

Analizziamo ora il numero di miss necessari per la simulazione di $\mathcal{P}$. Questi dipendono non linearmente dalla taglia Z della cache:

- $\tilde{\mu}N \leq Z$

  Poiché gli N contesti sono adiacenti e la cache ha dimensione $Z \geq \tilde{\mu}N$, ogni blocco della cache verrà caricato una ed una sola volta e quindi si verificheranno solo cold miss:

  $$Q_{Sim}(\mathcal{P}) = O\left(1 + \frac{\tilde{\mu}N}{L}\right) = O\left(1 + \frac{\mu N}{L}\right) \tag{4.16}$$

- $\tilde{\mu} \leq Z < \tilde{\mu}N$

  Sia $0 < \lambda \leq \log N$ l'indice del più grande cluster che può essere totalmente contenuto in una cache di dimensione Z. Se $\lambda = 0$ si ottiene il caso precedente. $\mathcal{P}$ può essere scomposto in sottoprogrammi massimali composti da soli superstep minori di λ o non minori.

  Consideriamo uno i-superstep con $0 \leq i < \lambda$. In `PackAndTag(`C^i`)` i puntatori l e m nei due cicli *while* scandiscono il contenuto del cluster, word dopo word, un numero costante di volte, cosicché ad ogni accesso in memoria verranno caricati in cache i dati delle successive $\Theta(L)$ richieste. Il numero di miss è quindi:

  $$Q_{Pack}(C^i, Z, L) = O\left(\frac{\mu N}{2^i L}\right). \tag{4.17}$$

  Il numero di miss per `Communication(`C^i`)` è dato dalla procedura precedente, dall'analoga `UnpackAndUntag(`C^i`)` e dall'ordinamento; se, come specificato in precedenza, si utilizza uno dei due algoritmi ottimi, il numero di miss vale:

  $$Q_{Ord}(C^i, Z, L) = O\left(\frac{\mu N \log \mu \frac{N}{2^i}}{2^i L \log Z}\right) \tag{4.18}$$

  e poiché $\mu \frac{N}{2^i} = \Omega(Z)$:

  $$Q_{Comm}(C^i, Z, L) = 2Q_{Pack}(C^i, Z, L) + Q_{Ord}(C^i, Z, L) = O\left(\frac{\mu N \log \mu \frac{N}{2^i}}{2^i L \log Z}\right). \tag{4.19}$$

  L'esecuzione dei contesti è ricorsiva e il numero di miss per un i-cluster è dato dal numero di miss generato dai due $(i+1)$-sottocluster. Un λ-cluster richiede invece $O\left(\frac{\mu N}{2^\lambda L}\right) = O\left(\frac{Z}{L}\right)$ (cold) miss e quindi vale la seguente ricorrenza:

  $$\begin{aligned} Q_{Comp}(C^i, Z, L) &= \begin{cases} O\left(\frac{Z}{L}\right) & \text{se } i = \lambda \\ 2Q_{Comp}(C^{i+1}, Z, L) + \Theta(1) & \text{altrimenti} \end{cases} \\ &= O\left(\frac{\mu N}{2^i L}\right). \end{aligned} \tag{4.20}$$

Iterando la ricorrenza si ottiene $Q_{Comp}(C^i, Z, L) = 2^k Q_{Comp}(C^{i+k}, Z, L) + \Theta(2^k)$. Il caso base si raggiunge quando $k = \lambda - i$ e quindi:

$$Q_{Comp}(C^i, Z, L) = O\left(\frac{2^\lambda Z}{2^i L}\right) = O\left(\frac{\mu N 2^\lambda Z}{\mu N 2^i L}\right) = O\left(\frac{\mu N}{2^i L}\right)$$

Sommando i contributi si ottiene che la simulazione di un i-cluster, con $0 \leq i < \lambda$, incorre in $Q_{ClSim}(C^i, Z, L)$ miss, dove:

$$Q_{ClSim}(C^i, Z, L) = Q_{Comp}(C^i, Z, L) + Q_{Comm}(C^i, Z, L) = O\left(\frac{\mu N \log \mu \frac{N}{2^i}}{2^i L \log Z}\right). \quad (4.21)$$

Consideriamo ora i superstep di indice non minore di λ e sia $\mathcal{P}'$ un sottoprogramma massimale di $\mathcal{P}$ composto da soli superstep di indice non minore di λ. $\mathcal{P}'$ può essere visto come un programma (ad eccezione dello 0-superstep finale) per un CD-BSP con $\frac{N}{2^\lambda}$ nodi. Poiché la cache ha dimensione $Z = \Omega\left(\frac{\mu N}{2^\lambda}\right)$, tale sottoprogramma può essere eseguito incorrendo in $O\left(\frac{\mu N}{2^\lambda L}\right)$ (cold) miss. Tale valore è però trascurabile perché vale il seguente lemma:

Lemma 4.3. *Per ogni sottoprogramma $\mathcal{P}'$ di $\mathcal{P}$ composto da soli superstep non minori di λ, il numero di miss è ammortizzato dal successivo superstep di $\mathcal{P}'$ in $\mathcal{P}$ di indice minore di λ.*

Dim. Sia $k < \lambda$ l'indice del primo superstep che segue $\mathcal{P}'$; la sua simulazione richiede per ogni k-cluster $\Omega\left(\frac{\mu N}{L 2^k}\right)$ miss (il cluster deve essere almeno letto). L'algoritmo prima di simulare il k-cluster esegue $\mathcal{P}'$ per i $2^{\lambda-k}$ λ-cluster contenuti. Ogni *round* richiede $O\left(\frac{\mu N}{2^\lambda L}\right)$ miss e quindi in totale $O\left(\frac{\mu N}{2^k L}\right)$ miss, trascurabili rispetto al numero di miss per la successiva simulazione del k-cluster. □

Il numero di miss dell'intera simulazione è dominato dai soli i-superstep con $i < \lambda$ e, siccome vi sono 2^i i-cluster, vale:

$$Q_{Sim}(\mathcal{P}, Z, L) = O\left(N \sum_{i=0}^{\lambda-1} k_i \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right). \quad (4.22)$$

- $\tilde{\mu} > Z$

  Il contesto non può essere contenuto nella cache e il numero di miss non è più indipendente dall'algoritmo simulato. Nei casi precedenti un contesto era sempre contenuto totalmente nella cache e i (cold) miss dell'esecuzione erano ammortizzati dagli i-superstep con $i < \lambda$ per il Lemma 4.3. Ora un contesto non può più essere contenuto in cache e la computazione locale può influire sul comportamento della simulazione se il numero di miss che genera è asintoticamente maggiore dei cold miss. Poiché nessun cluster è contenuto totalmente in cache, per le comunicazioni vale la (4.19). Il costo

della computazione invece segue una ricorrenza diversa da quella definita in (4.20) perché quest'ultima ipotizzava che fosse $\mu \leq Z$:

$$Q_{Comp}(C^i, Z, L) = \begin{cases} O(Q_\tau(Z, L)) & \text{se } i = \log N \\ 2Q_{Comp}(C^{i+1}, Z, L) + \Theta(1) & \text{altrimenti} \end{cases}$$
$$= O\left(\frac{N}{2^i} Q_\tau(Z, L)\right). \tag{4.23}$$

Sommando i vari contributi si ottiene che un i-superstep richiede:

$$Q_{ClSim}(C^i, Z, L) = O\left(\frac{N}{2^i}\left(Q_\tau(Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right)\right). \tag{4.24}$$

Sommando i contributi dei vari superstep:

$$Q_{Sim}(\mathcal{P}) = O\left(N \sum_{i=0}^{\log N} k_i \left(Q_\tau(Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right)\right). \tag{4.25}$$

Per unire i tre risultati trovati definiamo con $Q^*_\tau(Z, L)$ il numero di *capacity* miss durante l'esecuzione di un contesto e con λ:

$$\lambda = \min\left\{\{\log N + 1\} \cup \{0 \leq i \leq \log N \ : \ Z - \tilde{\mu}\frac{N}{2^i} \geq 0\}\right\};$$

Nei primi due casi elencati si ha $Q^*_\tau(Z, L) = 0$. Il risultato finale è dunque[2]:

$$Q_{Sim}(\mathcal{P}, Z, L) = O\left(1 + \frac{\mu N}{L} + N \sum_{i=0}^{\lambda-1} k_i \left(Q^*_\tau(Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right)\right). \tag{4.26}$$

Infatti nelle ipotesi del primo caso si ottiene $\lambda = 0$ e $Q^*_\tau(Z, L) = 0$, sostituendo si ottiene (4.16); nel terzo caso si avrebbe $\lambda = \log N + 1$ e quindi la (4.25); nel secondo si ottiene un valore di λ compreso tra 0 e $\log N + 1$ (esclusi) e $Q^*_\tau(Z, L) = 0$, semplificando si trova la (4.22). Sommando il tempo richiesto dai miss, ovvero $Q_{Sim}(\mathcal{P}, Z, L) * t$, con il tempo RAM (4.15), si ottiene la (4.8). □

4.4 Simulazione su M processori

Vogliamo ora simulare un programma per un modello CD-BSP($N, \mathbf{g}', Z', L', t'$), detto guest, su un CD-BSP($M, \mathbf{g}, Z, L, t$), detto host, con $M < N$, M e N potenze di due. Distribuiamo uniformemente i processori del guest, cosicché ogni nodo dell'host ospiti un $\log M$-cluster,

[2]Definiamo $\sum_{i=k}^{h}(\ldots) = 0$ se $k > h$.

ovvero $\frac{N}{M}$ nodi. Inoltre, per ogni nodo dell'host, distribuiamo i contesti come nel Paragrafo 4.2. Infine allochiamo altre $\Theta\left(\mu\frac{N}{M}\right)$ locazioni contigue, che chiameremo H, adiacenti ai contesti del guest.

Un programma $\mathcal{P}$ può essere scomposto in una serie di sottoprogrammi alternativamente costituiti da soli superstep di indice non minore di $\log M$ e da soli superstep di indice strettamente minore di $\log M$. La simulazione di $\mathcal{P}$ varia per questi due tipi di sottoprogrammi.

Nel secondo caso uno i-cluster è distribuito su due o più nodi dell'host e uno i-superstep del guest corrisponde ad un i-superstep dell'host. La simulazione si compone dei seguenti passi: ogni nodo dell'host esegue gli $\frac{N}{M}$ contesti che contiene; raccoglie $O\left(\mu\frac{N}{M}\right)$ messaggi in H destinati a processori del guest non contenuti nel nodo; li invia attraverso la rete; distribuisce con l'ordinamento i messaggi "interni" e quelli appena ricevuti. Quindi la simulazione può essere vista come un programma per CD-BSP$(M, \mathbf{g}, Z, L, t)$, dove ogni processore deve eseguire i contesti e distribuire messaggi. Analizziamo la complessità di tale simulazione.

Lo spostamento in H dei messaggi da spedire nella rete può essere eseguito in tempo $O\left(\mu\frac{N}{M}\right)$ e $O\left(1+\frac{\mu N}{LM}\right)$ miss, basta infatti una scansione lineare dei contesti e dello spazio H. La comunicazione attraverso la rete richiede tempo $O\left(\mu\frac{N}{M}g_i\right)$ (ogni nodo può inviare al più $\Theta(\mu)$ messaggi) e si può supporre che i messaggi in ingresso siano memorizzati in H. La comunicazione interna, ovvero la distribuzione con il packing, l'ordinamento e l'unpacking dei messaggi appena giunti e dei messaggi interni al nodo dell'host, richiede tempo $O\left(\mu\frac{N}{M}\log\mu\frac{N}{M}\right)$ e $O\left(1+\frac{\mu N}{LM}+\frac{\mu N\log\frac{\mu N}{M}}{ML\log Z}\right)$ miss perché la cache è totalmente invalidata in seguito all'utilizzo della rete. In maniera simile a quanto svolto per il caso $M=1$, si ottiene che uno i-superstep, $0 \le i < \log M$, richiede:

$$T^{RAM}_{ClSim}(C^i) = O\left(\tau\frac{N}{M}+\frac{\mu N}{M}\log\frac{\mu N}{M}\right)$$

$$Q_{ClSim}(C^i, Z, L) = O\left(Q^*_\tau(Z,L)\frac{N}{M}+1+\frac{\mu N}{ML}+\frac{\mu N\log\mu\frac{N}{M}}{ML\log Z}\right)$$

$$T_{ClSim}(C^i, Z, L) = O\left(\tau\frac{N}{M}+\frac{\mu N}{M}\log\frac{\mu N}{M}+\frac{N}{M}\left(Q^*_\tau(Z,L)+\frac{\mu\log\mu\frac{N}{M}}{L\log Z}\right)t+\frac{\mu N}{M}g_i\right)$$

Nella seconda e terza equazione con $Q^*_\tau(Z,L)$ si intende il numero di capacity miss nell'esecuzione di un contesto di $\mathcal{P}$. Nella terza equazione, unione delle prime due, invece, la quantità $1+\frac{\mu N}{ML}$ è coperta dalle comunicazioni per quanto affermato in (4.3). Se un sottoprogramma

contiene k_i i-superstep, con $0 \leq i < \log M$, allora la simulazione di questi richiede:

$$T'^{RAM}_{Sim}(\mathcal{P}) = O\left(\frac{N}{M}\sum_{i=0}^{\log M-1} k_i\left(\tau + \mu\log\mu\frac{N}{M}\right)\right)$$

$$Q'_{Sim}(\mathcal{P}, Z, L) = O\left(\sum_{i=0}^{\log M-1} k_i\left(\frac{N}{M}Q^*_\tau(Z, L) + 1 + \frac{\mu N}{M} + \frac{\mu N\log\mu\frac{N}{M}}{ML\log Z}\right)\right)$$

$$T'_{Sim}(\mathcal{P}) = \frac{N}{M}O\left(\sum_{i=0}^{\log M-1} k_i\left(\tau + \mu\log\frac{\mu N}{M} + \left(Q^*_\tau(Z, L) + \frac{\mu\log\mu\frac{N}{M}}{L\log Z}\right)t + \mu g_i\right)\right)$$

Nel caso di un sottoprogramma composto da soli superstep non minori di $\log M$, ogni processore dell'host può ospitare uno o più i-cluster: ogni nodo dell'host simulerà in parallelo con gli altri un programma per CD-BSP con $\frac{N}{M}$ processori[3]. Infatti uno i-superstep può essere visto come uno $(i - \log M)$-superstep di un CD-BSP con $\frac{N}{M}$ processori. Quindi un sottoprogramma con k_i i-superstep, $\log M \leq i < \lambda$ (λ è sempre l'indice del più grande cluster che può essere contenuto in cache) richiede:

$$T''^{RAM}_{Sim}(\mathcal{P}) = O\left(\frac{N}{M}\sum_{i=\log M}^{\log N} k_i\left(\tau + \log\frac{\mu N}{2^i}\right)\right),$$

$$Q''_{Sim}(\mathcal{P}, Z, L) = O\left(1 + \frac{\mu N}{ML} + \frac{N}{M}\sum_{i=\log M}^{\lambda-1} k_i\left(Q^*_\tau(Z, L) + \frac{\mu\log\mu\frac{N}{2^i}}{L\log Z}\right)\right),$$

$$T_{Sim}''(\mathcal{P}, Z, L) = O\left(\left(1 + \frac{\mu N}{ML}\right)t + \frac{N}{M}\sum_{i=\log M}^{\log N} k_i\left(\tau + \mu\log\mu\frac{N}{2^i}\right) + \right.$$
$$\left. + \frac{N}{M}\sum_{i=\log M}^{\lambda-1} k_i\left(Q^*_\tau(Z, L) + \frac{\mu\log\mu\frac{N}{2^i}}{L\log Z}\right)t\right)$$

Sommando i contributi dei vari sottoprogrammi si ottiene il seguente teorema analogo al Lemma di Brent per computazioni PRAM [8]:

Teorema 4.4. *Sia $\mathcal{P}$ un programma per CD-BSP$(N, \boldsymbol{g}, Z', L', t')$ con contesto μ, tempo RAM τ e $Q^*_\tau(Z, L)$ capacity miss per ogni computazione locale. La sua simulazione su una*

[3]Non sono esattamente programmi per CD-BSP perché non è garantita l'esistenza di uno 0-superstep finale.

macchina CD-BSP$(M, \boldsymbol{g}, Z, L, t)$ *richiede:*

$$T^{RAM}(\mathcal{P}) = O\left(\frac{N}{M}\sum_{i=0}^{\log N} k_i\left(\tau + \mu \log \mu M_i\right)\right) \tag{4.27}$$

$$Q(\mathcal{P}, Z, L) = O\left(1 + \frac{\mu N}{ML} + \sum_{i=0}^{\max\{\log M, \lambda\}-1} k_i\left(\frac{N}{M}Q^*_\tau(Z, L) + 1 + \frac{\mu N}{LM} + \frac{\mu N}{LM}\frac{\log \mu M_i}{\log Z}\right)\right) \tag{4.28}$$

$$T(\mathcal{P}, Z, L) = O\left(\frac{N}{M}\sum_{i=0}^{\log N} k_i\left(\tau + \mu \log \mu M_i\right) + \left(1 + \frac{\mu N}{ML}\right)t + \right.$$
$$\left. + \frac{N}{M}\sum_{i=0}^{\max\{\log M, \lambda\}-1} k_i\left(Q^*_\tau(Z, L) + \frac{\mu \log \mu M_i}{L \log Z}\right)t + \sum_{i=0}^{\log M - 1} k_i \mu g_i\right) \tag{4.29}$$

dove:

$$M_i = \min\left\{\frac{N}{2^i}, \frac{N}{M}\right\} \tag{4.30}$$

$$\lambda = \min\left\{\{\log N + 1\} \cup \{0 \leq i \leq \log N \ : \ Z - \tilde{\mu}\frac{N}{2^i} \geq 0\}\right\} \tag{4.31}$$

4.5 Conclusioni

La procedura di simulazione presentata non si basa su alcun parametro della cache, né sulla banda del modello simulato. Quindi se l'algoritmo parallelo per CD-BSP non si basa su alcun parametro della cache, la sua simulazione sarà cache-oblivious. In particolare un algoritmo fine-grained per (C)D-BSP è cache-oblivious poiché il contesto ha dimensione $\Theta(1)$ e quindi la computazione locale incorre sempre e solo in cold miss, indipendentemente dai parametri della cache.

Capitolo 5

Estensioni

Nel capitolo precedente è stata presentata una simulazione di validità generale per portare algoritmi per CD-BSP su un modello CM (o su un CD-BSP con un numero minore di processori). Di seguito verranno proposte delle simulazioni *ad hoc* che permetteranno di ottenere algoritmi sequenziali più efficienti rispetto a quelli ottenibili con la procedura del capitolo precedente. Infatti la simulazione delle comunicazioni con l'ordinamento non utilizza certe regolarità nei pattern di comunicazione degli algoritmi paralleli.

Questo capitolo è così suddiviso: nel Paragrafo 5.1 verranno presentate le simulazioni *ad hoc*; nel Paragrafo 5.2 verranno confrontati i risultati tra i due tipi di simulazioni; nel Paragrafo 5.4 verrà analizzata la simulazione di algoritmi per D-BSP($N, \mathbf{g}$) su un processore con $\log N$ livelli di cache; infine nel Paragrafo 5.3 verrà analizato il concetto di regolarità.

5.1 Casi particolari di pattern di comunicazione

Sia $\mathcal{M}$ un algoritmo che permetta di simulare le comunicazioni di un certo problema $\mathcal{P}$. Se $\mathcal{M}$ richiede $O\left(1 + \frac{\mu N}{L2^i}\right)$ miss, $O\left(\mu \frac{N}{2^i}\right)$ tempo RAM e $O\left(\mu \frac{N}{2^i}\right)$ spazio ausiliario per distribuire i messaggi di un i-cluster, allora chiameremo simulazione *ad hoc* la procedura di simulazione ottenuta da quella descritta nel capitolo precedente sostituendo `Communication(`C^i`)` con $\mathcal{M}$. Vale il seguente teorema:

Teorema 5.1. *Sia $\mathcal{P}$ un algoritmo per CD-BSP($N, \boldsymbol{g}', Z', L', t'$) con contesto μ, tempo RAM τ e $Q^*_\tau(Z, L)$ capacity miss per computazione locale; sia $\mathcal{M}$ un algoritmo che permetta di simulare le comunicazioni di $\mathcal{P}$ in un i-cluster con $Q_{\mathcal{M}}(C^i, Z, L) = O\left(1 + \frac{\mu N}{L2^i}\right)$ miss, $T^{RAM}_{\mathcal{M}}(C^i) = O\left(\mu \frac{N}{2^i}\right)$ tempo RAM e $S_{\mathcal{M}}(C^i) = O\left(\mu \frac{N}{2^i}\right)$ spazio ausiliario. Allora la simulazione di $\mathcal{P}$ su un CD-BSP($1, \boldsymbol{g}, Z, L, t$), utilizzando $\mathcal{M}$ per la simulazione dei messaggi,*

richiede:

$$T_{Sim}^{RAM}(\mathcal{P}) = O\left(N \sum_{i=0}^{\log N} k_i \tau\right), \tag{5.1}$$

$$Q_{Sim}(\mathcal{P}, Z, L) = O\left(1 + \frac{\mu N}{L} + N \sum_{i=0}^{\lambda-1} k_i \left(Q_\tau^*(Z, L) + \frac{\mu}{L}\right)\right), \tag{5.2}$$

$$T_{Sim}(\mathcal{P}, Z, L, t) = O\left(\left(1 + \frac{\mu N}{L}\right) t + N\left(\sum_{i=0}^{\log N} k_i \tau + \sum_{i=0}^{\lambda-1} k_i \left(Q_\tau^*(Z, L) + \frac{\mu}{L}\right) t\right)\right) \tag{5.3}$$

dove:

$$\lambda = \min\left\{\{\log N + 1\} \cup \{i : Z - \tilde{\mu}\frac{N}{2^i} \geq 0\}\right\}, \tag{5.4}$$

$$\tilde{\mu} = \Theta(\mu).$$

Dim. L'algoritmo è analogo a quello presentato nel capitolo precedente per la simulazione generica a patto che: 1) $\tilde{\mu}$ sia di dimensione tale da poter generare con un *eventuale* packing (contenuto in $\mathcal{M}$) $S_{\mathcal{M}}(C^i)$ spazio, e 2) di sostituire la procedura `Communication(`C^i`)` con $\mathcal{M}$`(`C^i`)`. Anche la dimostrazione della simulazione è analoga a quella presentata nel capitolo precedente. La (4.14) diventa:

$$\begin{aligned} T_{ClSim}^{RAM}(C^i) &= T_{Comp}^{RAM}(C^i) + T_{\mathcal{M}}^{RAM}(C^i) \\ &= O\left(\tau\frac{N}{2^i} + \mu\frac{N}{2^i}\right) = O\left(\tau\frac{N}{2^i}\right) \end{aligned} \tag{5.5}$$

perché $\tau = \Omega(\mu)$. La (4.21) invece diventa:

$$Q_{ClSim}(C^i, Z, L) = Q_{Comp}(C^i, Z, L) + Q_{\mathcal{M}}(C^i, Z, L) = O\left(\frac{\mu N}{2^i L}\right). \tag{5.6}$$

Modificando di conseguenza le successive equazioni si ottiene la tesi. □

5.2 Applicazioni

Applichiamo ora i teoremi 4.2 e 5.1 a tre degli algoritmi più diffusi in letteratura: moltiplicazione di matrici, trasformata discreta di Fourier e ordinamento.

5.2.1 La moltiplicazione di matrici

Per la moltiplicazione di matrici utilizziamo l'algoritmo descritto nel Paragrafo 3.1.1 con $N = 2^{2k}$. Il contesto è $\Theta(1)$, infatti ogni processore alloca un elemento per ognuna delle due

matrici da moltiplicare e per il risultato, $\tau = \Theta(1)$ e $Q^*_\tau(Z, L) = 0$. $\mathcal{P}$ è composto da solo $2j$-superstep con $j = 0 \dots \frac{\log N}{2}$ e $k_{2j} = \Theta(2^j)$. Sostituendo si ottiene:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, Z, L) &= O\left(1 + \frac{\mu N}{L} + N \sum_{j=0}^{\lfloor \frac{\lambda-1}{2} \rfloor} 2^j \frac{\mu \log \mu \frac{N}{2^{j+1}}}{L \log Z}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N}{L \log Z} \sum_{j=0}^{\lfloor \frac{\lambda-1}{2} \rfloor} 2^j \log \frac{N}{2^j}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N 2^{\frac{\lambda}{2}} \log N}{L \log Z}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N\sqrt{N}}{L\sqrt{Z}} \frac{\log N}{\log Z}\right). \end{aligned}$$

perché se $\lambda > 0$ allora $\frac{N}{2^\lambda} = \Theta(Z)$, mentre se $\lambda = 0$ allora per definizione $\tilde{\mu} N \leq Z$ e $\frac{N 2^{\frac{\lambda}{2}} \log N}{L \log Z} = O(\frac{N}{L})$. Per quanto riguarda la complessità temporale, da (4.6) si ottiene:

$$\begin{aligned} T^{RAM}_{Sim}(\mathcal{P}) &= O\left(N \sum_{j=0}^{\frac{\log N}{2}} 2^{j+1} \mu \log \mu \frac{N}{2^j}\right) \\ &= O\left(N \sum_{j=0}^{\frac{\log N}{2}} 2^j \log \frac{N}{2^j}\right) \\ &= O\left(N\sqrt{N} \log N\right). \end{aligned}$$

Se i contesti, ordinati rispetto all'indice del processore, rappresentano le matrici in bit-interleaved layout, allora le comunicazioni in un i-cluster C, che consistono in $\Theta(1)$ scambi tra coppie di $i+2$-cluster di C, si possono effettuare tramite scambio di sottomatrici; ma, per le proprietà del layout, ogni sottomatrice è allocata in un blocco unico e lo scambio di dati richiede tempo lineare, $\Theta\left(1 + \frac{\mu N}{L 2^i}\right)$ miss e nessuno spazio supplementare. Possiamo quindi applicare il Teorema 5.1:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, Z, L) &= O\left(1 + \frac{\mu N}{L} + \frac{\mu N}{L} \sum_{j=0}^{\lfloor \frac{\lambda-1}{2} \rfloor} 2^j\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N}{L} 2^{\frac{\lambda}{2}}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N\sqrt{N}}{L\sqrt{Z}}\right). \end{aligned}$$

Per la complessità RAM:

$$T_{Sim}^{RAM}(\mathcal{P}) = O\left(N \sum_{j=0}^{\frac{\log N}{2}} 2^{j+1}\right) = O\left(N\sqrt{N}\right)$$

Per quanto affermato nel Paragrafo 2.2.1, tale risultato è ottimo, mentre la simulazione generica risulta asintoticamente maggiore dell'ottimo di un fattore logaritmico, che deriva dall'ordinamento effettuato ad ogni fase di comunicazione.

5.2.2 La trasformata discreta di Fourier

L'algoritmo descritto nel Paragrafo 3.1.2 su $N = 2^{2k}$ processori richiede per ogni nodo $\Theta(2^j)$ superstep del tipo $\left(1 - \frac{1}{2^j}\right)\log N$, $j = 0 \ldots \log\log\sqrt{N}$. Ogni nodo è costituito da un contesto $\Theta(1)$, richiede tempo di computazione $\Theta(1)$ e $Q_\tau^*(Z, L) = 0$ capacity miss, quindi la (4.7) diventa:

$$\begin{aligned}
Q_{Sim}(\mathcal{P}, Z, L) &= O\left(1 + \frac{\mu N}{L} + N \sum_{j=0}^{\lceil \log \frac{\log N}{\log N - \lambda + 1} \rceil} 2^j \frac{\mu \log \frac{\mu N}{2^{\left(1-\frac{1}{2^j}\right)\log N}}}{L \log Z}\right) \\
&= O\left(1 + \frac{N}{L} + \frac{N}{L \log Z} \sum_{j=0}^{\lceil \log \frac{\log N}{\log N - \lambda + 1} \rceil} 2^j \log N^{\frac{1}{2^j}}\right) \\
&= O\left(1 + \frac{N}{L} + \frac{N}{L \log Z} \sum_{j=0}^{\lceil \log \frac{\log N}{\log N - \lambda + 1} \rceil} \log N\right) \\
&= O\left(1 + \frac{N}{L} + \frac{N \log N}{L \log Z} \log \frac{\log N}{\log N - \lambda}\right) \\
&= O\left(1 + \frac{N}{L} + \frac{N \log N}{L \log Z} \log \frac{\log N}{\log Z}\right).
\end{aligned}$$

La (4.6) invece diventa:

$$\begin{aligned}
T_{Sim}^{RAM}(\mathcal{P}) &= O\left(N \sum_{j=0}^{\log\log N} 2^{j+1} \log \frac{\mu N}{2^{\left(1-\frac{1}{2^j}\right)\log N}}\right) \\
&= O\left(N \sum_{j=0}^{\log\log N} 2^j \frac{1}{2^j} \log N\right) \\
&= O\left(N \log N \sum_{j=0}^{\log\log N} 1\right) \\
&= O\left(N \log N \log\log N\right).
\end{aligned}$$

Anche per la trasformata di Fourier le comunicazioni possono essere simulate in tempo e miss lineare: supponiamo che i contesti del C^i rappresentino una matrice in row-major di dimensione $\sqrt{\frac{N}{2^i}} \times \sqrt{\frac{N}{2^i}}$ (μ è di dimensione trascurabile $\Theta(1)$); ogni riga rappresenta un $j = \frac{1}{2}(\log N + i)$-cluster. Nella comunicazione ogni h-esimo j-cluster scambia il k-esimo contesto con l'h-esimo contesto del k-esimo j-cluster, $0 \leq k, h < \sqrt{\frac{N}{2^i}}$, ma questa non è altro che la trasposizione di una matrice che può essere eseguita in tempo lineare, con $O\left(1 + \frac{\mu N}{2^i L}\right)$ miss e nessuno spazio lineare. Applicando il teorema:

$$\begin{aligned} Q(\mathcal{P}, Z, L) &= O\left(1 + \frac{\mu N}{L} + \frac{\mu N}{L} \sum_{j=0}^{\lceil \log \frac{\log N}{\log N - \lambda + 1} \rceil} 2^j\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N}{L} \frac{\log N}{\log N - \lambda}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N}{L} \frac{\log N}{\log Z}\right) \end{aligned}$$

Per la complessità RAM:

$$\begin{aligned} T^{RAM}(\mathcal{P}) &= O\left(\mu N \sum_{j=0}^{\log \log N} 2^j\right) \\ &= O\left(N \log N\right) \end{aligned}$$

La simulazione generica restituisce un algoritmo non ottimo sia nel numero di miss sia nella complessità RAM; la simulazione *ad hoc*, invece, permette di raggiungere l'ottimo in entrambi i valori (la simulazione coincide esattamente con l'algoritmo presentato in [17]). È interessante notare che lo slowdown dell'algoritmo generico è minore rispetto a quello calcolato per la moltiplicazione.

Analizziamo ora il semplice algoritmo parallelo ottenuto direttamente dal FFT dag (vedi Paragrafo 3.1.2). Ogni i-superstep viene eseguito $\Theta(1)$ volte da ogni processore, perciò:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, Z, L) &= O\left(1 + \frac{\mu N}{L} + N \sum_{i=0}^{\lambda-1} \frac{\mu \log \frac{\mu N}{2^i}}{L \log Z}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N}{L \log Z} \sum_{i=0}^{\lambda-1} \log \frac{N}{2^i}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N \log N}{L \log Z} \lambda\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N \log N}{L \log Z} \log \frac{N}{Z}\right) \end{aligned}$$

Per il tempo RAM, invece:

$$\begin{aligned} T_{Sim}^{RAM}(\mathcal{P}) &= O\left(N \sum_{i=0}^{\log N} \mu \log \mu \frac{N}{2^i}\right) \\ &= O\left(N \sum_{i=0}^{\log N} \log \frac{N}{2^i}\right) \\ &= O\left(N \log^2 N\right) \end{aligned}$$

Anche in questo caso è possibile distribuire i messaggi con un semplice algoritmo: se i contesti, ordinati rispetto all'indice del processore, rappresentano il vettore, allora le comunicazioni in un i-cluster $C = C_1C_2$ consistono nello scambio dei contesti tra C_1 e C_2. Questa operazione richiede tempo $O\left(\mu\frac{N}{2^i}\right)$, $O\left(1 + \frac{\mu N}{L2^i}\right)$ miss e nessuno spazio aggiuntivo poiché ogni cluster occupa un blocco unico di memoria. Siamo quindi nelle ipotesi del Teorema 5.1:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, Z, L) &= O\left(\left(1 + \frac{\mu N}{L}\right) + N \sum_{i=0}^{\lambda-1} \frac{\mu}{L}\right) \\ &= O\left(\left(1 + \frac{N}{L}\right) + \frac{N}{L} \log \frac{N}{Z}\right) \\ T_{Sim}^{RAM}(\mathcal{P}) &= O\left(N \sum_{i=0}^{\log N} \mu\right) \\ &= O\left(N \log N\right) \end{aligned}$$

Il numero di miss della simulazione ad hoc non è ottimo, ma l'aumento è dovuto solo all'algoritmo parallelo scelto dacché la complessità RAM è ottima. È interessante notare che i due algoritmi paralleli per la FFT risultano ottimi per il D-BSP(N, x^α), mentre solo il primo è ottimo per il D-BSP$(N, \log x)$.

5.2.3 L'ordinamento

Per sottolineare ulteriormente le caratteristiche della simulazione generica e di quella ad hoc, analizziamo il comportamento dell'algoritmo parallelo di ordinamento con le procedure di simulazione.

Il k-sort con bitonic sort e $k = \Theta(1)$, descritto nel Paragrafo 3.1.3, richiede $\Theta(i + 1)$ i-superstep; il contesto ha dimensione $\Theta(1)$, la computazione locale ha complessità $\Theta(1)$ e

non vi sono capacity miss. Il numero di miss della simulazione è quindi:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, Z, L) &= O\left(1 + \frac{\mu N}{L} + N\sum_{i=0}^{\lambda-1}(i+1)\frac{\mu \log \frac{\mu N}{2^i}}{L \log Z}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N}{L \log Z}\sum_{i=0}^{\lambda-1} i \log \frac{N}{2^i}\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N \log N}{L \log Z}\lambda^2\right) \\ &= O\left(1 + \frac{N}{L} + \frac{N \log N}{L \log Z}\log^2 \frac{N}{Z}\right) \end{aligned}$$

Mentre la complessità temporale vale:

$$\begin{aligned} T_{Sim}^{RAM}(\mathcal{P}) &= O\left(N\sum_{i=0}^{\log N} i\ \mu \log \mu \frac{N}{2^i}\right) \\ &= O\left(N\sum_{i=0}^{\log N} i \log \frac{N}{2^i}\right) \\ &= O\left(N \log^3 N\right) \end{aligned}$$

Evidentemente, non è efficiente simulare le comunicazioni di un algoritmo di ordinamento con l'ordinamento. Se però i contesti, ordinati rispetto all'indice del processore, rappresentano il vettore da ordinare, allora le comunicazioni in un i-cluster $C = C_1C_2$ consistono nello scambio dei contesti tra C_1 e C_2. Questa operazione richiede tempo $O\left(\mu\frac{N}{2^i}\right)$, $O\left(1 + \frac{\mu N}{L2^i}\right)$ miss e nessuno spazio aggiuntivo. Siamo quindi nelle ipotesi del Teorema 5.1:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, Z, L) &= O\left(\left(1 + \frac{\mu N}{L}\right) + N\sum_{i=0}^{\lambda-1} i\frac{\mu}{L}\right) \\ &= O\left(\left(1 + \frac{N}{L}\right) + \frac{N}{L}\log^2 \frac{N}{Z}\right) \\ T_{Sim}^{RAM}(\mathcal{P}) &= O\left(N\sum_{i=0}^{\log N} i\mu\right) \\ &= O\left(N \log^2 N\right) \end{aligned}$$

Con la simulazione generica non si ottiene un algoritmo di ordinamento efficiente poiché le comunicazioni vengono eseguite da un algoritmo di ordinamento sequenziale. Con la simulazione ad hoc, invece, è possibile ricavare un algoritmo sequenziale quasi-ottimo: in questo caso, però, il risultato non ottimo è imputabile all'algoritmo parallelo scelto poiché la complessità RAM coincide con il numero di nodi del grafo computazionale di bitonic sort.

5.3 Simulazioni regolari

In questa tesi è stato utilizzato il modello a cache ideale con tecnica di rimpiazzo ottima che non è implementabile in pratica. In [17] è stato dimostrato che se un algoritmo è regolare (vedi Definizione 2.1) allora il numero di miss tra una cache ideale e una cache LRU non cambia asintoticamente.

È possibile enunciare il concetto di regolarità per un modello CD-BSP: un algoritmo per CD-BSP è regolare se il numero di miss per computazione locale $Q_\tau(Z, L)$ è regolare, ovvero $Q_\tau(Z, L) = O(Q_\tau(2Z, L))$. Un algoritmo fine-grained $\mathcal{P}$ per CD-BSP o per D-BSP è banalmente regolare poiché il contesto μ di $\mathcal{P}$ ha dimensione costante e dunque incorre sempre in $O(1 + \frac{\mu}{L}) = O(1)$ miss indipendentemente da Z.

Gli algoritmi per il modello a cache ideale ottenuti dalla simulazione (sia con ordinamento che ad hoc) non sono necessariamente regolari: ad esempio un algoritmo parallelo con $\log N$ k-superstep incorre solo in cold miss se $Z = \frac{\tilde{\mu}N}{2^k}$, oppure in almeno $O\left(\frac{\mu N}{L} \log N\right)$ se la cache viene dimezzata. La regolarità delle simulazioni dipende dal numero di superstep, in particolare possiamo enunciare i seguenti teoremi:

Teorema 5.2. *Condizione sufficiente per ottenere un algoritmo sequenziale regolare da una simulazione con ordinamento o ad hoc di un algoritmo regolare $\mathcal{P}$ per CD-BSP è che $\forall i$, $0 < i \leq \log N$, $k_i = O(k_{\tilde{i}})$ per un certo $\tilde{i} < i$ e $k_0 = O(1)$, dove k_i indica il numero di i-superstep in $\mathcal{P}$.*

Dim. Di seguito con $\lambda(Z)$ rappresenteremo l'indice λ definito precedentemente, sottolineando la sua dipendenza da Z.
L'ipotesi di regolarità vale anche per il numero di capacity miss, dacché:

$$\begin{aligned} Q_{Sim}(\mathcal{P}, 2Z, L) &= O\left(1 + \frac{\mu N}{L} + N \sum_{i=0}^{\lambda(2Z)-1} k_i \left(Q^*_\tau(2Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log 2Z}\right)\right) \\ &= O\left(1 + \frac{\mu N}{L} + N \sum_{i=0}^{\lambda(2Z)-1} k_i \left(Q^*_\tau(Z, L) + \frac{\mu \log \mu \frac{N}{2^i}}{L \log Z}\right)\right) \end{aligned} \tag{5.7}$$

Innanzitutto consideriamo il caso in cui $Z < \frac{\tilde{\mu}}{2}$: sia con una cache di dimensione Z che $2Z$ il contesto non può essere contenuto in cache e quindi $\lambda(Z) = \lambda(2Z) = \log N + 1$ e per (5.7) $Q_{Sim}(\mathcal{P}, Z, L) = O(Q_{Sim}(\mathcal{P}, 2Z, L))$.
Sia $\frac{\tilde{\mu}}{2} \leq Z < \tilde{\mu}$: il contesto può essere totalmente mantenuto in una cache di dimensione $2Z$ e quindi $Q_{Sim}(\mathcal{P}, 2Z, L)$ vale

$$Q_{Sim}(\mathcal{P}, 2Z, L) = O\left(N \sum_{i=0}^{\log N-1} k_i \frac{\mu \log \mu \frac{N}{2^i}}{L \log 2Z}\right).$$

Ma poiché:

$$Q^*_\tau(Z,L) = O(Q^*_\tau(2Z,L)) = O\left(1+\frac{\mu}{L}\right) = O\left(\frac{\mu \log \mu\frac{N}{2^i}}{L\log Z}\right)\ \forall\ 0 \le i \le \log N,$$

$$k_{\log N} = O(k_{\tilde{i}}),\ \tilde{i} < \log N,$$

$$\frac{\mu \log \mu\frac{N}{2^{\log N}}}{L\log Z} = O\left(\frac{\mu \log \mu\frac{N}{2^{\tilde{i}}}}{L\log Z}\right),$$

si ottiene:

$$Q_{Sim}(\mathcal{P},Z,L) = O\left(N\sum_{i=0}^{\log N} k_i\left(Q^*_\tau(Z,L) + \frac{\mu\log\mu\frac{N}{2^i}}{L\log Z}\right)\right) = O(Q_{Sim}(\mathcal{P},2Z,L)).$$

Sia ora $\tilde{\mu} \le Z < \frac{\tilde{\mu}N}{2}$: $Q_{Sim}(\mathcal{P},Z,L)$ e $Q_{Sim}(\mathcal{P},2Z,L)$ differiscono asintoticamente per il solo termine $Nk_{\lambda-1}\frac{\mu\log\frac{\mu N}{2^i}}{L\log Z}$ perché $\lambda(2Z) = \lambda(Z) - 1$, ma esso è asintoticamente trascurabile:

$$Nk_{\lambda-1}\frac{\mu\log\frac{\mu N}{2^{\lambda-1}}}{L\log Z} = O\left(Nk_{\tilde{i}}\frac{\mu\log\frac{\mu N}{2^{\lambda-1}}}{L\log Z}\right) = O\left(Nk_{\tilde{i}}\frac{\mu\log\frac{\mu N}{2^{\tilde{i}}}}{L\log Z}\right).$$

Sia ora $\frac{\tilde{\mu}N}{2} \le Z < \tilde{\mu}N$: si ottiene che $\lambda(Z) = 1$, mentre $\lambda(2Z) = 0$; dimezzando la cache, il numero di miss aumenta di $O\left(Nk_0\frac{\mu\log\mu N}{L\log Z}\right)$, ma $Z = \Theta(\mu N)$ e quindi tale termine è $O\left(\frac{\mu N}{L}\right) = O(Q(\mathcal{P},2Z,L))$.
Infine sia $Z \ge \tilde{\mu}N$: tutta la simulazione è già contenuta pienamente in cache e quindi $Q_{Sim}(\mathcal{P},Z,L) = O(Q_{Sim}(\mathcal{P},2Z,L)) = O\left(1+\frac{\mu N}{L}\right)$.
È analoga la dimostrazione per le simulazioni ad hoc. □

Teorema 5.3. *Condizione necessaria e sufficiente per ottenere un algoritmo sequenziale regolare da una simulazione ad hoc di un algoritmo regolare $\mathcal{P}$ per CD-BSP è che $\sum_{i=0}^{j} k_i = O(\sum_{i=0}^{j-1} k_i)$ $\forall i\ 0 < i \le \log N$ e $k_0 = O(1)$.*

Dim. La dimostrazione della sufficienza è analoga a quella precedente, che risulta un caso particolare del presente teorema. La necessità deriva dalla definizione di regolarità applicata al caso in cui $\tilde{\mu} \le Z < \tilde{\mu}N$ $(\lambda(2Z) = \lambda(Z) - 1)$:

$$Q_{Sim}(\mathcal{P},Z,L) = O(Q_{Sim}(\mathcal{P},2Z,L))$$

$$O\left(N\sum_{i=0}^{\lambda(Z)-1} k_i\frac{\mu}{L}\right) = O\left(N\sum_{i=0}^{\lambda(2Z)-1} k_i\frac{\mu}{L}\right) \iff O\left(\sum_{i=0}^{\lambda(Z)-1} k_i\right) = O\left(\sum_{i=0}^{\lambda(Z)-2} k_i\right)$$

□

Gli algoritmi paralleli per la moltiplicazione di matrici, la trasformata discreta di Fourier e l'ordinamento verificano le ipotesi del Teorema 5.2; ad esempio per la moltiplicazione di matrici si ha che $k_{2j} = O(2^j)$ e $k_{2j+1} = 0$, $0 \le j \le \frac{\log N}{2}$, quindi $k_{2j} = O(k_{2(j-1)})$, $k_{2j+1} = O(k_{2j})$ e $k_0 = O(1)$. Le simulazioni di questi algoritmi sono quindi regolari e i risultati trovati nel Paragrafo 5.2 valgono anche per cache con politica di rimpiazzo LRU.

5.4 Simulazione su $\log N$ livelli di cache

Per sottolineare la forte correlazione tra le località della cache e la località di un modello parallelo, utilizziamo come host una macchina con una gerarchia spinta di $\log N$ livelli di cache e una memoria RAM, come guest un D-BSP (consideriamo quindi un modello con solo località di comunicazione). Utilizziamo come gerarchia di memorie il modello descritto nel Paragrafo 2.2.6. Con Z_j, L_j e t_j, $0 \leq j \leq \log N - 1$, indichiamo rispettivamente la dimensione totale, la lunghezza del blocco e il tempo di accesso alla cache $j+1$ (o alla memoria se $j = \log N - 1$) della cache j. Con Z, L e t indichiamo, invece, il vettore contenente i singoli valori per ogni cache rispettivamente della dimensione, taglia del blocco e tempo di accesso. Riguardo alla simulazione di un programma per D-BSP si può enunciare quanto segue:

Teorema 5.4. *Sia $\mathcal{P}$ un algoritmo fine-grained per D-BSP$(N, \boldsymbol{g})$ e sia $\mathcal{M}$ un algoritmo cache-oblivious che permetta di simulare le comunicazioni di un i-cluster con $O\left(1 + \frac{\mu N}{L 2^i}\right)$ miss, tempo RAM $O\left(\mu \frac{N}{2^i}\right)$ e spazio ausiliario $O\left(\mu \frac{N}{2^i}\right)$. Se la simulazione di $\mathcal{P}$ è regolare, allora la simulazione di $\mathcal{P}$ su un processore con $\log N$ livelli di cache di dimensione $Z_j \geq 2Z_{j-1}$ e $Z_0 = \Omega(\tilde{\mu})$ richiede:*

$$Q_j(\mathcal{P}, Z_j, L_j) = O\left(\frac{\mu N}{L_j} \sum_{i=0}^{\log N - j - 1} k_i\right) \qquad \forall\, 0 \leq j \leq \log N - 1 \tag{5.8}$$

$$T^{RAM}(\mathcal{P}) = O\left(N \sum_{i=0}^{\log N} k_i \tau\right) \tag{5.9}$$

$$T(\mathcal{P}, Z, L, t) = O\left(N \sum_{i=0}^{\log N} k_i \tau + N \sum_{j=0}^{\log N - 1} \frac{\mu t_j}{L_j} \sum_{i=0}^{\log N - j - 1} k_i\right) \tag{5.10}$$

Inoltre se $g_i = \sum_{j=0}^{j=\log N - i - 1} \frac{t_j}{L_j}$ e $Z_j = \tilde{\mu} \frac{N}{2^{\log N - j}}$ si ottiene un analogo al lemma di Brent, ovvero:

$$T(\mathcal{P}, Z, L, t) = \Theta(N\tilde{T}) \tag{5.11}$$

dove $\tilde{T}$ indica il tempo totale dell'algoritmo parallelo.

Dim. La simulazione di $\mathcal{P}$ è regolare, quindi l'utilizzo di una politica LRU non aumenta asintoticamente il numero di miss. Per il Lemma 2.9, il numero di miss nel livello i è pari al numero di miss nel caso in cui vi sia solo la cache i. Poiché per costruzione $Z_i \geq \tilde{\mu} \frac{N}{2^{\log N - i}}$, risulta che $\lambda = \log N - i$ e, quindi, la tesi (5.8). La complessità RAM è trasparente alla cache e quindi vale la (5.1).
Se le dimensioni delle cache sono esattamente $Z_j = \tilde{\mu} \frac{N}{2^{\log N - j}}$, per le tre equazioni precedenti vale l'o-piccolo e quindi, se si raccolgono in (5.10) i vari k_i, si ottiene:

$$N \sum_{i=0}^{\log N} k_i \tau + \mu N \sum_{i=0}^{\log N - 1} k_i \sum_{j=0}^{\log N - i - 1} \frac{t_j}{L_j} = N\Theta\left(\sum_{i=0}^{\log N} k_i \tau + \mu \sum_{i=0}^{\log N - 1} k_i g_i\right).$$

L'argomento del Θ è il tempo di esecuzione dell'algoritmo su un D-BSP e quindi si ottiene un analogo al lemma di Brent per il modello PRAM [8]. □

Si ha quindi uno slowdown lineare quando il tempo di comunicazione in un i-cluster è pari al costo *medio* della lettura di un blocco della cache $\log N - i$.

Lemma 5.5. *Dato un modello D-BSP(N, $\boldsymbol{g}$) con g_i non crescente con i è sempre possibile costruire una gerarchia di cache tali che $g_i = \sum_{j=0}^{\log N-i-1} \frac{t_j}{L_j}$ (pesatura su $\boldsymbol{g}$).*

Dim. Per $i = \log N$ si ottiene sempre $g_{\log N} = 0$ per l'indice negativo ad apice della sommatoria, invece per un generico $0 \leq i < \log N$ si ottiene la ricorrenza:

$$g_i = g_{i+1} + \frac{t_{\log N-i-1}}{L_{\log N-i-1}}$$
$$\Delta_i = g_i - g_{i+1} = \frac{t_{\log N-i-1}}{L_{\log N-i-1}}$$

Poiché il parametro di banda g_i è non crescente con i, $\Delta_i \geq 0$ e se poniamo $L_{\log N-i-1} = 2L_{\log N-i-2} = 2^{\log N-i-1}L_0$ si ottiene $t_{\log N-i-1} = 2^{\log N-i-1}\Delta_i L_0$. Con questa impostazione è il tempo t_i che rispecchia l'andamento dell'inverso della banda, ovvero se tra un i-cluster ed un $i+1$-cluster non vi è differenza di banda ($\Delta_i = 0$), la cache associata ad $i+1$ è fittizia perché il tempo di accesso tra la cache $\log N - i$ e $\log N - i - 1$ è nullo. □

Sia ad esempio $g_i = \alpha \log \frac{N}{2^i}$, se poniamo $t_j = 2^j \alpha L_0$ si ottiene la pesatura cercata. Invece per $g_i = \left(\frac{N}{2^i}\right)^\alpha$ si ha $t_j = 2^\alpha 2^{j(1+\alpha)}(1 - 2^{-\alpha})L_0$.

Definiamo come bandwidth-oblivious un algoritmo parallelo che risulti ottimo per tutti i possibili parametri di banda di comunicazione $\mathbf{g}$ non crescenti ($g_i \geq g_{i+1}$). Dal lemma e dal teorema precedenti è possibile ricavare il seguente corollario:

Corollario 5.6. *Sia $\mathcal{P}$ un algoritmo per D-BSP($N, \boldsymbol{g}$) che verifichi le ipotesi del Teorema 5.4 e la cui simulazione su un unico processore con una cache ideale comporti un algoritmo cache-oblivious ottimo e regolare, allora $\mathcal{P}$ è banda-oblivious per tutti i modelli D-BSP con parametro di banda g_i non crescente all'aumentare di i.*

Dim. Supponiamo per assurdo che esista un algoritmo $\mathcal{P}'$ con contesto μ' tale che per un certo modello di D-BSP(N, **g'**) $\tilde{T}_{P'} < \tilde{T}_P$. Simuliamo questo algoritmo su un un modello di cache ideale gerachico con $Z_i = \tilde{\mu}' \frac{N}{2^i}$ e pesato su **g'**; il tempo di simulazione è $N\tilde{T}_{P'} < N\tilde{T}_P$, ma questo è assurdo perché l'algoritmo ottenuto simulando P è cache-oblivious ottimo, ovvero il tempo totale è sempre ottimo indipendentemente dalla gerarchia di cache sottostante perché causa il minor numero di miss in ogni livello. □

Capitolo 6

Conclusioni

Nei due capitoli precedenti sono state presentate due strategie di simulazione di algoritmi paralleli su modelli dotati di gerarchie di cache: la prima, totalmente generale, vale per qualsiasi algoritmo parallelo, anche nel caso in cui le comunicazioni vengano determinate a tempo di esecuzione; la seconda, invece, si basa sul particolare pattern di comunicazione dell'algoritmo e perciò è necessario conoscere anticipatamente il programma per poter determinare un metodo efficiente per simulare lo scambio di messaggi.

La prima simulazione utilizza l'ordinamento per distribuire i messaggi tra processori. L'utilizzo dell'ordinamento è una scelta obbligata: le comunicazioni sono una rete di permutazioni che nel caso peggiore richiede tanti miss quanti l'ordinamento [4], pertanto la simulazione *generale* non può essere ulteriormente migliorata. La simulazione di alcuni algoritmi D-BSP per tre problemi tradizionali (moltiplicazione di matrici, trasformata di Fourier, ordinamento) non offre algoritmi per CM ottimi né nel numero di miss né nella complessità RAM: gli algoritmi ottenuti sono comunque maggiori dell'ottimo solo di un fattore $O(\log \mu N)$ per il tempo e $O\left(\frac{\log \mu N}{\log Z}\right)$ per i miss[1].

Per poter migliorare la simulazione è necessario analizzare il pattern di comunicazione dell'algoritmo: l'ordinamento, infatti, non distingue tra comunicazioni con particolari regolarità e comunicazioni prive di struttura. Di questo, appunto, si occupano le simulazioni *ad hoc*, con cui è possibile ottenere dei risultati ottimi per il modello a cache ideale. In queste simulazioni è necessario conoscere a priori il pattern di comunicazione per poter implementare una strategia per la distribuzione dei messaggi che richieda tempo RAM e numero di miss lineari (rispettivamente $\Theta\left(\mu\frac{N}{2^i}\right)$ e $\Theta\left(\frac{\mu N}{L2^i}\right)$ per un i-cluster). Proprio perché non è possibile costruire una rete di permutazioni lineare nel numero di miss, non è possibile progettare un algoritmo efficiente per ogni pattern; inoltre è chiaro che non è realizzabile una simulazione ad hoc per algoritmi paralleli con pattern di comunicazione determinati a tempo di esecuzione.

Nelle simulazioni ad hoc, l'utilizzo di una sola cache "nasconde" parzialmente la gestione efficiente dei cluster: se λ è l'indice del cluster più grande che può essere contenuto in cache,

[1] In questo capitolo viene analizzato solo il caso in cui $Z = \Omega(\mu)$ e $Z = O(\mu N)$.

non esiste alcun motivo per preferire un $i+1$-superstep ad un i-superstep con $i \neq \lambda$. Infatti il numero di miss e il tempo RAM per simulare un i-cluster sono esattamente il doppio di quelli richiesti da un $i+1$-superstep. In realtà il buon utilizzo dei cluster viene premiato con il variare della dimensione della cache: se con una cache di dimensione Z, che può contenere fino ad un λ-cluster, non vi è differenza tra l'utilizzo di un λ ed un $\lambda+1$ superstep, con una cache di dimensione $\frac{Z}{2}$ tale differenza conta. Infatti, un $i+1$-cluster può essere contenuto in cache, ma non un i-cluster, quindi aumenta inevitabilmente il numero di miss. L'utilizzo efficiente dei cluster sembra essere una condizione necessaria, ma non sufficiente, per ottenere algoritmi ottimi *cache-oblivious*.

I modelli HMM e BT differiscono rispetto al modello a cache ideale per questo motivo: nei primi generalmente si studia l'andamento degli algoritmi per un numero limitato di bande (solitamente x^α e $\log x$); nel secondo, invece, i parametri Z, L e t sono totalmente arbitrari a meno della proprietà di *tall* cache. Non è possibile usare un unico modello di banda per progettare algoritmi che abbiano uno slowdown lineare sul CM come avviene per i modelli HMM e BT [16]. Infatti, per ottenere uno slowdown lineare sono richieste le bande **g** e **g'** per le simulazioni rispettivamente generiche e ad hoc, dove:

$$g_i = \begin{cases} \Theta\left(\log \frac{\mu N}{2^i}\right) & \text{se } \lambda \leq i < \log N \\ \Theta\left(\log \frac{\mu N}{2^i} + \frac{t \log \frac{\mu N}{2^i}}{L \log Z}\right) & \text{altrimenti;} \end{cases} \tag{6.1}$$

$$g'_i = \begin{cases} \Theta(1) & \text{se } \lambda \leq i < \log N \\ \Theta\left(\frac{t}{L}\right) & \text{altrimenti} \end{cases} \tag{6.2}$$

Una buona approssimazione di un modello *effective* si potrebbe ottenere con il modello D-BSP$(N, \log x)$, infatti g_i ha un andamento principalmente logaritmico e l'appiattimento del parametro di banda g'_i per $i < \lambda$ è approssimato abbastanza fedelmente dal logaritmo. Un esempio a favore di questa scelta è dato dalla trasformata di Fourier. Nei capitoli precedenti sono stati presentati due algoritmi per la DFT: il primo è la semplice esecuzione del DFT dag ($\Theta(1)$ superstep di tipo i, $0 \leq i < \log N$); il secondo si basa sulla divisione del dag in $\sqrt{N}$ sottografi da $\sqrt{N}$ nodi (2^i superstep di tipo $(1 - \frac{1}{2^i}) \log N$, $0 \leq i < \log \log N$). Nel D-BSP(N, x^α) entrambe le versioni sono ottime, mentre nel D-BSP$(N, \log x)$ solo il secondo è ottimo; anche nelle simulazioni si ottiene che il secondo è migliore del primo, dunque il parametro di banda logaritmico sembrerebbe avere una maggior effectivness della polinomiale.

Il principale problema del passaggio da (C)D-BSP a CM è che l'appiattimento del costo di comunicazione all'interno di un cluster non si riconduce necessariamente ad un appiattimento del costo di accesso alla memoria. Infatti, nel D-BSP il concetto di località di comunicazione scompare all'interno di un cluster. La località di riferimento, invece, scompare solo quando un cluster può essere contenuto pienamente in cache; in caso contrario, il tempo di accesso ad un dato varia a seconda che questo sia in cache oppure in memoria.

Un punto di unione tra i due modelli è la correlazione tra i concetti bandwith-oblivious e cache-oblivious ottimo: se la simulazione di un programma $\mathcal{P}$ restituisce un algoritmo cache-oblivious ottimo, allora $\mathcal{P}$ è bandwith-oblivious.

La simulazione presentata può essere utilizzata anche in modelli di cache con tecniche di rimpiazzo non ottime a patto che gli algoritmi paralleli da simulare soddisfino certe caratteristiche, non particolarmente restrittive, sul numero di superstep.

Bibliografia

[1] *www.cs.ucl.ac.uk/staff/s.bhatti/teaching/1b10/notes/1b10-03-memory.pdf.*

[2] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[6] Gianfranco Bilardi, Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. On effetiveness of D-BSP as a bridging model of parallel computation. In *Proceedings of the International Conference of Computational Science*, volume 2074 of *Lectures Notes in Computer Science*, pages 579–588, 2001.

[7] Gianfranco Bilardi and Franco Preparata. Processor-time tradeoff under bounded-speed message propagation: Part I, upper bounds. *Theory Computation Systems*, 30:523–546, 1997.

[8] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–208, 1974.

[9] Yi-Jen Chiang, Micheal T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroffm, and Jeffery Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, San Francisco, California, USA, 1995.

[10] James C. Cooley and John W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT Press, 2 edition, 2001.

[12] Pilar De la Torre and Clyde P. Kruskal. Submachine locality in the bulk synchronous settinf. In *Proceedings of the EUROPAR 96*, volume 1124 of *Lecture Notes in Computer Science*, pages 352–358, Lyon, France, August 1996.

[13] Frank Dehne, Wolfang Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse grained parallel algorithms. *Algotihmica*, 36(2):97–122, April 2003.

[14] Frank Dehne, David Hutchinson, Anil Maheshwari, and Wolfang Dittrich. Bulk synchronous parallel algorithms for the external memory model. *Theory Computation Systems*, 35(6):567–597, 2002.

[15] Carlo Fantozzi. *A Computational model for parallel and hierarchical machines*. PhD thesis, Università di Padova, 2003.

[16] Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. Translating submachine locality into locality of reference. *Journal of Parallel and Distributed Computing, Special resume on 18th IPDS*, In print, 2005.

[17] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–297, New York, NY, October 1999.

[18] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT Department of Electrical Engineering and Computer Science, June 1999.

[19] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, 2002.

[20] Jop F. Sibeyn and Micheal Kaufmann. BSP-like external-memory computation. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 229–240, 1999.

[21] Daniel D. Sleator and Robert E. Tarjan. Amortized Efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[22] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] Uzi Vishkin. Can parallel algorithm enhance serial implementation? *Communication of the ACM*, 39(9):88–91, 1996.