View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea specialistica in Tecnologie Informatiche

Tesi di laurea

Skeleton Data Parallel per OcamlP3l

Nicoletta Triolo

Relatore	Controrelatore
Prof.ssa Susanna Pelagatti	Dott.ssa Laura Ricci

Anno Accademico 2008/2009

Indice generale

Introduzione **vii**

1 Gli skeleton **1**

1.1 Gli skeleton: la definizione 1

1.2 Gli skeleton task parallel 3

1.2.1 Pipeline 3

1.2.2 Farm 4

1.3 Gli skeleton data parallel 6

1.3.1 Map 6

1.3.2 Reduce 7

1.3.3 Stencil fisso 9

1.4 Gli skeleton di controllo 12

1.4.1 Loop 12

1.5 Modello analitico dei costi 13

1.6 Sommario 15

2 Gli skeleton nella letteratura **17**

2.1 Ambiente di programmazione 17

2.1.1 Linguaggi per la programmazione parallela strutturata 17

2.1.2 Librerie di skeleton 21

2.2 Implementazione 22

2.2.1 Template 22

2.2.2 Data flow 23

2.2.3 Architettura e supporto alle comunicazioni 25

2.3 Sistemi esistenti 26
2.3.1 P^3L 26
2.3.2 eSkel 28
2.3.3 Skeleton in MetaOcaml 31
2.3.4 SkeTo 33
2.3.5 BlockLib 35
2.3.6 Muskel 39
2.3.7 Assist 40
2.4 Conclusioni 44

3 Gli skeleton in OcamlP3l **49**
3.1 OcamlP3l: skeleton task parallel 50
3.1.1 Pipeline 50
3.1.2 Farm 51
3.2 OcamlP3l: skeleton data parallel 51
3.2.1 Mapvector 51
3.2.2 Reducevector 52
3.3 OcamlP3l: skeleton di controllo 53
3.3.1 Loop 53
3.3.2 Seq 54
3.3.3 Parfun 55
3.3.4 Pardo 56
3.4 Annidamento di skeleton 57
3.5 I colori 58

4 Ocamlp3l: implementazione **61**
4.1 I p3ltree 61
4.2 Il modello d'esecuzione 65
4.3 Dai p3ltree ai template 67
4.4 I template 86
4.5 Inserimento di nuovi skeleton 91
4.6 Il ruolo delle lambda-astrazioni 92

5 Modifica degli skeleton data parallel esistenti **95**

5 .1 Mapvector 95

5.2 Reducevector 100

6 Map: un nuovo skeleton data parallel **105**

6.1 Il punto di vista dell'utente 106

6.1.1 Distribuzione dei dati in P^3L 106

6.1.2 Distribuzione dei dati in OcamlP3l 108

6.1.3 Bigarray 109

6.1.4 Processori virtuali/reali 112

6.1.5 Iterazione 113

6.1.6 Un esempio completo 115

6.2 Implementazione 117

6.2.1 Algoritmo di distribuzione 117

6.2.2 Modello di calcolo 121

6.2.3 Scambio dati e calcolo della condizione di terminazione 124

6.2.4 Dai p3ltree ai template 125

6.2.5 I template 129

6.3 Esempi 141

6.3.1 Mandelbrot 141

6.3.2 Prodotto di matrici 144

6.3.3 Game of life 146

7 Risultati sperimentali **149**

8 Conclusioni e sviluppi futuri **157**

.

Introduzione

La tesi si inserisce nell'ambito della programmazione a skeleton: uno skeleton è una particolare struttura di un algoritmo parallelo con la quale possono essere parallelizzate molte classi di problemi sequenziali; uno skeleton è dunque uno schema algoritmico parametrico rispetto alla funzione sequenziale da eseguire in parallelo.

Dall'analisi del problema sequenziale è possibile derivare il particolare skeleton, tra alcuni ben conosciuti, che esegue un'applicazione parallela funzionalmente equivalente a quella sequenziale ma in generale caratterizzata da prestazioni migliori.

Come vedremo gli skeleton possono essere suddivisi in due categorie: task parallel, in cui il parallelismo deriva dall'applicazione contemporanea di una funzione su più elementi di una sequenza di input, e data parallel, in cui il parallelismo deriva dall'applicazione contemporanea di una funzione su parti diverse dello stesso elemento, una struttura dati decomponibile, di una sequenza di input.

Vedremo anche che gli skeleton possono essere utilizzati per raggiungere due principali obiettivi, spesso in antitesi tra loro, e sono da un lato le prestazioni massime dall'altro la facilità con la quale il programmatore può esprimere un'applicazione parallela anche riutilizzando codice preesistente pur ottenendo miglioramenti prestazionali.

Negli ultimi vent'anni il concetto di skeleton ha avuto molto successo in ambito accademico e sono stati proposti numerosi sistemi che offrono skeleton sia task che data parallel e che si propongono di raggiungere uno dei due principali obiettivi appena visti. In particolare in letteratura sono stati proposti sia linguaggi per la programmazione parallela strutturata, in cui gli skeleton sono offerti come costrutti

nativi di tali linguaggi, sia delle librerie, in cui gli skeleton sono offerti come funzioni o metodi.

La nostra tesi verte sul sistema *OCamlP3l*, una libreria in linguaggio OCaml, che deriva da uno dei primi linguaggi per la programmazione parallela strutturata, P^3L, sviluppato presso il Dipartimento di Informatica dell'Università di Pisa. Il lavoro è stato volto principalmente all'introduzione di uno skeleton data parallel che permettesse di parallelizzare computazioni di struttura più complessa rispetto agli skeleton già esistenti nel sistema. Tali computazioni sono denominate stencil e tendono a trovarsi in applicazioni iterative in cui la condizione di terminazione spesso dipende dal risultato ottenuto ad ogni iterazione; lo skeleton introdotto permette di esprimere anche queste applicazioni.

La tesi è organizzata come segue:

- nel primo capitolo diamo una definizione di skeleton e definiamo alcuni concetti che serviranno nel corso della tesi; vediamo inoltre la semantica informale di alcuni skeleton conosciuti sia task che data parallel e una breve introduzione al modello dei costi di tali skeleton;

- nel secondo capitolo vediamo lo stato dell'arte della letteratura sugli skeleton, dando prima una lettura generale ai diversi filoni seguiti e poi una panoramica di alcuni sistemi esistenti con alcuni esempi di utilizzo degli stessi; infine concludiamo con una valutazione critica dei sistemi visti;

- nel terzo capitolo vediamo gli skeleton offerti all'utente dalla libreria *OCamlP3l* prima del contributo della tesi; in particolare diamo una visione di come l'utente li deve utilizzare al fine di esprimere un'applicazione parallela;

- nel quarto capitolo vediamo l'architettura del sistema, il modello di esecuzione e i dettagli implementativi; infine tracciamo le linee guida per l'inserimento di nuovi skeleton nel sistema;

- nel quinto capitolo vediamo le modifiche apportate all'implementazione degli skeleton appartenenti alla categoria data parallel, che avevano in precedenza

una implementazione task parallel funzionalmente equivalente, per renderli effettivamente data parallel;

- nel sesto capitolo vediamo il nuovo skeleton introdotto prima dal punto di vista dell'utente, e quindi i tipi di dato introdotti e la sintassi che l'utente deve usare e poi dal punto di vista degli algoritmi che esegue e dell'implementazione; infine diamo alcuni esempi di codice utente relativi ad alcune applicazioni note;
- nel settimo capitolo mostriamo i risultati sperimentali ottenuti per il nuovo skeleton;
- nell'ottavo capitolo diamo le conclusioni circa il lavoro svolto, gli obiettivi raggiunti, le limitazioni e alcuni sviluppi futuri.

Capitolo 1

Gli skeleton

1.1 Gli skeleton: la definizione

Secondo la definizione data da Murray Cole in [Col91] uno skeleton è una forma di parallelismo che calcola una certa funzione, è uno schema generico, indipendente dalla funzione che si vuole calcolare. E' lo scheletro di un algoritmo parallelo, la struttura di una computazione parallela instanziabile con qualunque funzione particolare.

In termini funzionali, uno skeleton è una funzione di ordine superiore parametrica nella funzione sequenziale da calcolare. Quest'ultima definizione rende il concetto di skeleton particolarmente adatto all'implementazione in un linguaggio funzionale, anche se è chiaramente adattabile ad un linguaggio imperativo.

Data una funzione $f\colon A \rightarrow B$ e un modulo P che accetta in ingresso uno stream di valori a_i di tipo A e restituisce uno stream di valori di output $f(a_i)$ di tipo B, un' implementazione parallela P' del modulo P, con grado di parallelismo n, funzionalmente equivalente, ma caratterizzata da prestazioni migliori, è costituita da n processi (*processori virtuali*) comunicanti tra loro, e può essere ricavata applicando un certo skeleton.

I processori virtuali sono i processi che necessitano per una parallelizzazione "ideale", cioè considerando il modulo P isolato dal contesto in cui si trova: lo stream di

ingresso infinitamente veloce e costi di comunicazione tra nodi nulli. I processori virtuali dunque eseguono la computazione a grana minima possibile relativamente al problema in esame. Il grado di parallelismo n è detto *grado di parallelismo ideale.* Il concetto di grado di parallelismo ideale si adatta anche al caso di modulo P non isolato, indicando quindi il grado di parallelismo che consente di raggiungere le prestazioni migliori considerati il *tempo di interarrivo* dello stream di ingresso (tempo che trascorre tra l'arrivo di due elementi consecutivi dello stream) e i costi di comunicazione non nulli.

Il grado di parallelismo ideale nella pratica non è sempre raggiungibile per cause materiali (mancanza di nodi di elaborazione) o per motivi di prestazioni (possibile introduzione di colli di bottiglia dovuti proprio al numero di processori); si rende quindi spesso necessario passare ad un grado di parallelismo n' minore di n, e in questo caso si parla di *processori reali.* Assume una grande importanza l'utilizzo di determinate strategie per associare nel modo migliore i processori virtuali ai processori reali, allocandoli sui nodi fisici che eseguiranno effettivamente la computazione.

Per derivare una possibile parallelizzazione di un certo modulo sequenziale è buona norma ragionare in termini di processori virtuali ed in seguito passare ad una rappresentazione del modulo parallelo in termini di processori reali riducendo il grado di parallelismo ed eventualmente aumentando la grana del calcolo.

Questi concetti saranno più chiari in seguito, una volta presentati nel dettaglio gli skeleton più comuni. Come descritto in [Pel93], le forme di parallelismo, e quindi gli skeleton, si suddividono in due classi:

- *task (o stream) parallel*: il parallelismo è raggiunto tramite l'applicazione della funzione f su differenti elementi dello stream di input; a questa classe appartengono ad esempio gli skeleton *pipeline* e *farm.*
- *data parallel*: il parallelismo è raggiunto tramite l'applicazione della funzione f su parti differenti dello stesso elemento dello stream di input; a questa classe appartengono ad esempio gli skeleton *map* e *reduce.*

Per loro natura gli skeleton task parallel hanno senso solo su stream (per questo

sono detti anche stream parallel) mentre gli skeleton data parallel possono essere utilizzati con vantaggio anche in assenza di stream.

Buona parte delle computazioni sequenziali possono essere parallelizzate tramite una composizione o l'iterazione dei pochi skeleton sopra citati e che vedremo dettagliatamente nelle sezioni 1.2, 1.3 e 1.4. Infatti una importante caratteristica degli skeleton è la possibilità di essere annidati tra loro, creando una combinazione di uno o più skeleton. Un'applicazione parallela è dunque un albero di skeleton, in cui le foglie rappresentano la computazione sequenziale da eseguire per ogni dato di ingresso.

Nelle sezioni 1.2, 1.3 e 1.4 introdurremo gli skeleton più comuni descrivendone informalmente la semantica e daremo loro una rappresentazione grafica (*topologia*) sotto forma di grafo orientato. I nodi del grafo rappresentano i processori virtuali, mentre gli archi orientati rappresentano i canali di comunicazione tra i nodi.

1.2 Gli skeleton task parallel

1.2.1 Pipeline

Date $f_1 : A_0 \rightarrow A_1$, $f_2 : A_1 \rightarrow A_2$,..., $f_n : A_{n-1} \rightarrow A_n$ e x_i elemento di tipo A_0 dello stream di ingresso, per ogni x_i , lo skeleton *pipeline* $(f_1, f_2, \ldots, f_n)\ x_i$ calcola la funzione $F : A_0 \rightarrow A_n$, restituendo uno stream di tipo A_n di elementi:

$$y_i = F\ x_i = f_n(\ldots(f_2(f_1(x_i))\ldots).$$

Come si vede una pipeline è una composizione di funzioni; è costituita da n stadi in cui lo stadio *j-esimo* corrisponde alla funzione *j-esima* che compone F.

In Fig. 1 è mostrato un pipeline di n stadi: il primo stadio ha in ingresso uno stream di elementi x_i , vi applica f_1 e invia il risultato al secondo stadio; ogni stadio intermedio (stadio_j) applica la sua funzione (f_j) all'elemento che riceve dallo stadio precedente (stadio_{j-1}) e invia il risultato (y^j_i) allo stadio successivo (stadio_{j+1}). L'ultimo stadio riceve il risultato dal penultimo stadio, vi applica f_n , e invia il risultato finale (quindi il risultato di $F\ x_i$) sullo stream di uscita.

Il parallelismo deriva dal fatto che, su stream e a regime, i diversi stadi della pipeline eseguono contemporaneamente la funzione f_j a cui corrispondono su un diverso elemento x_i dello stream di input.

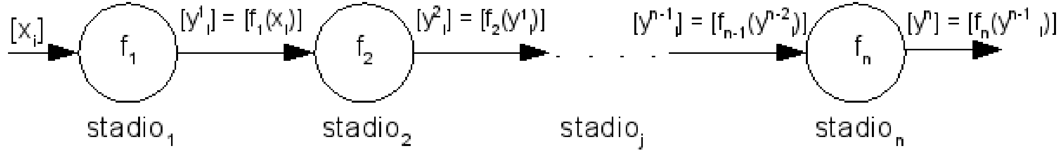

Fig. 1: Un pipeline di n stadi che calcola la funzione F. Con la notazione [m] si intende uno stream di elementi m

Vediamo un esempio di computazione (su stream) parallelizzabile con una pipeline. Vogliamo applicare la funzione

$$F = ((x + 2) * 6) / 8 \quad \text{(eq.1)}$$

ad uno stream di numeri interi.

In generale una sua possibile parallelizzazione è data da un pipeline di 3 stadi:

- il primo stadio calcola la funzione $f_1(x) = x + 2$;
- il secondo stadio calcola la funzione $f_2(x) = x * 6$;
- il terzo stadio calcola la funzione $f_3(x) = x / 8$.

1.2.2 Farm

Dati $f : A \to B$, funzione pura (senza stato), e x_i elemento di tipo A dello stream di ingresso, per ogni x_i , lo skeleton *farm* $f\ x_i$ calcola $f\ x_i$, restituendo uno stream di tipo B di elementi:

$$y_i = f(x_i).$$

In Fig. 2 è mostrata la topologia dello skeleton farm; un farm è costituito da un processore virtuale detto emettitore, n processori virtuali detti worker, e un processore

virtuale detto collettore. L'emettitore riceve gli elementi dello stream di ingresso e si occupa di assegnare ognuno ad un diverso worker. Sugli n worker è replicata la funzione f e ognuno di essi la esegue sull'elemento che riceve dall'emettitore. Infine il collettore riceve i risultati delle elaborazioni dei diversi worker e li inoltra su uno stream di uscita.

Il parallelismo deriva dal fatto che, a regime, la funzione f è applicata contemporaneamente a n elementi dello stream dagli n worker.

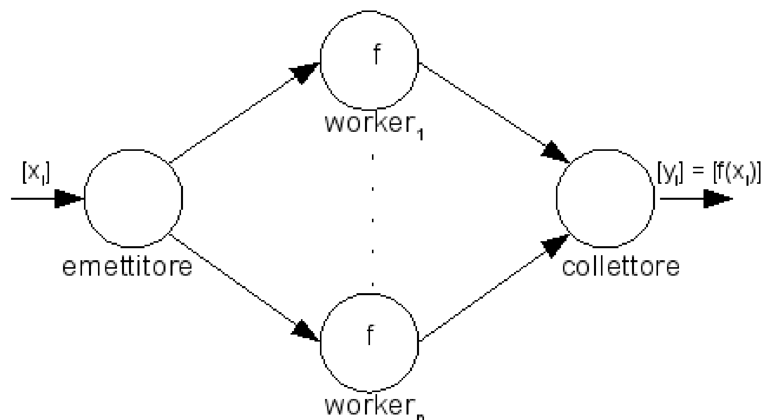

Fig. 2: farm con n worker che applicano la funzione f

L'emettitore, per la distribuzione dei task, può seguire diverse strategie, ad esempio:

- round robin, ovvero distribuire i task ciclicamente dal worker 1 al worker n
- on demand, ovvero distribuire i task ai worker liberi che ne fanno richiesta

Queste due strategie si differenziano in termini di bilanciamento del carico: la strategia on demand è sicuramente la migliore da questo punto di vista, in caso di elevata varianza di calcolo, perché minimizza il numero di worker inattivi durante il calcolo.

Supponiamo di voler calcolare la funzione *eq.1* ancora ad uno stream di numeri interi. La funzione, in quanto pura, si presta ad una parallelizzazione con paradigma farm. Notiamo che, mentre per derivare una parallelizzazione di F con paradigma pipeline è necessario conoscere la struttura della funzione da calcolare (composizione

di funzioni), per derivare una parallelizzazione di F con paradigma farm si potrebbe anche non conoscerla.

Nel caso in questione ogni worker eseguirà la funzione $F = ((x + 2) * 6) / 8$ sull'elemento dello stream ricevuto dall'emettitore e invierà il risultato al collettore.

1.3 Gli skeleton data parallel

1.3.1 Map

Dati $f : A \rightarrow B$, e uno stream di ingresso di vettori $X_i = [x_1, \ldots, x_m]$ di componenti di tipo A, per ogni X_i, lo skeleton *map f* X_i calcola $F : A^m \rightarrow B^m$, restituendo uno stream di tipo B^m di elementi:

$$Y_i = F\ X_i = [f(x_1), \ldots, f(x_m)].$$

In Fig. 3 vediamo una rappresentazione grafica dello skeleton map.

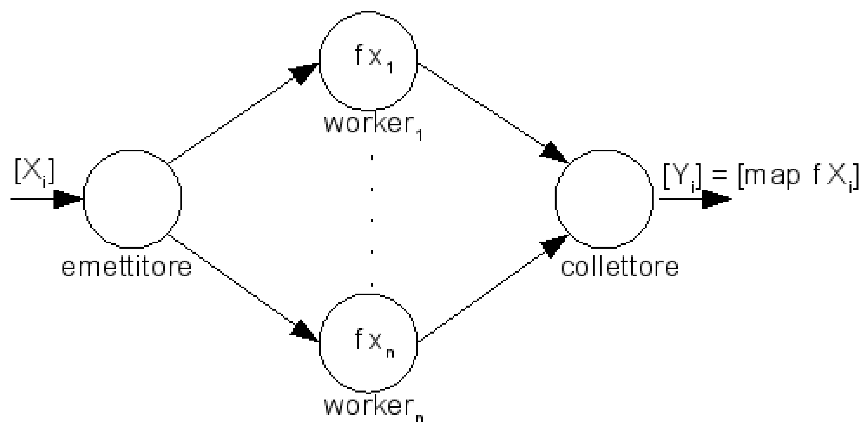

Fig. 3: Lo skeleton map con n = m worker che applica la funzione f a tutti gli elementi dei vettori X in ingresso. La notazione map f X ha il significato consueto dei linguaggi di programmazione funzionali

La topologia del map mostrata in Fig. 3 può ricordare quella del farm, ma in questo caso si ha partizionamento del dato (vettore) in ingresso tra i diversi worker e replicazione della funzione f che viene applicata ai diversi elementi di X_i. Il parallelismo

deriva dal fatto che la funzione f viene applicata contemporaneamente ai diversi elementi del vettore in ingresso.

Idealmente, come in Fig. 3, se il vettore ha lunghezza m, il map è composto da $m + 2$ processori virtuali: l'emettitore, il collettore e m worker, ognuno contenente un elemento del vettore. In questo caso l'emettitore ha il compito di suddividere ogni vettore dello stream di input e distribuire ogni elemento ad un worker diverso. Ogni worker applica la funzione f al proprio elemento e invia il risultato al collettore. Il collettore ha il compito di ricomporre gli elementi ricevuti dai worker in un vettore di output che invia sullo stream di uscita. Se i processori reali sono $n < m$ ogni processore reale riceve dall'emettitore una partizione di ampiezza m / n ed applica la funzione f ad ogni elemento della stessa.

Supponiamo di voler calcolare la stessa funzione *eq. 1* a tutti gli elementi dei vettori di interi che costituiscono uno stream di ingresso. In questo caso ogni worker applica la funzione $F = ((x + 2) * 6) / 8$ ad ogni elemento della partizione che detiene e invia la partizione dei risultati al collettore.

1.3.2 Reduce

Dati $f: A * A \rightarrow A$ e uno stream di ingresso di vettori $X_i = [x_1,...., x_m]$ di componenti di tipo A, per ogni X_i , lo skeleton *reduce* $f\ X_i$ calcola $F: A^m \rightarrow A$, restituendo un stream di tipo A di elementi:

$$y_i = F\ X_i = f\ x_1\ (f\ x_2\ (...(f\ x_m)...))).$$

La funzione f è un operatore associativo binario. La proprietà associativa permette di poter usare alcuni particolari schemi di parallelizzazione. Ad esempio in Fig. 4 vediamo uno schema (logicamente) ad albero per poter calcolare F in parallelo.

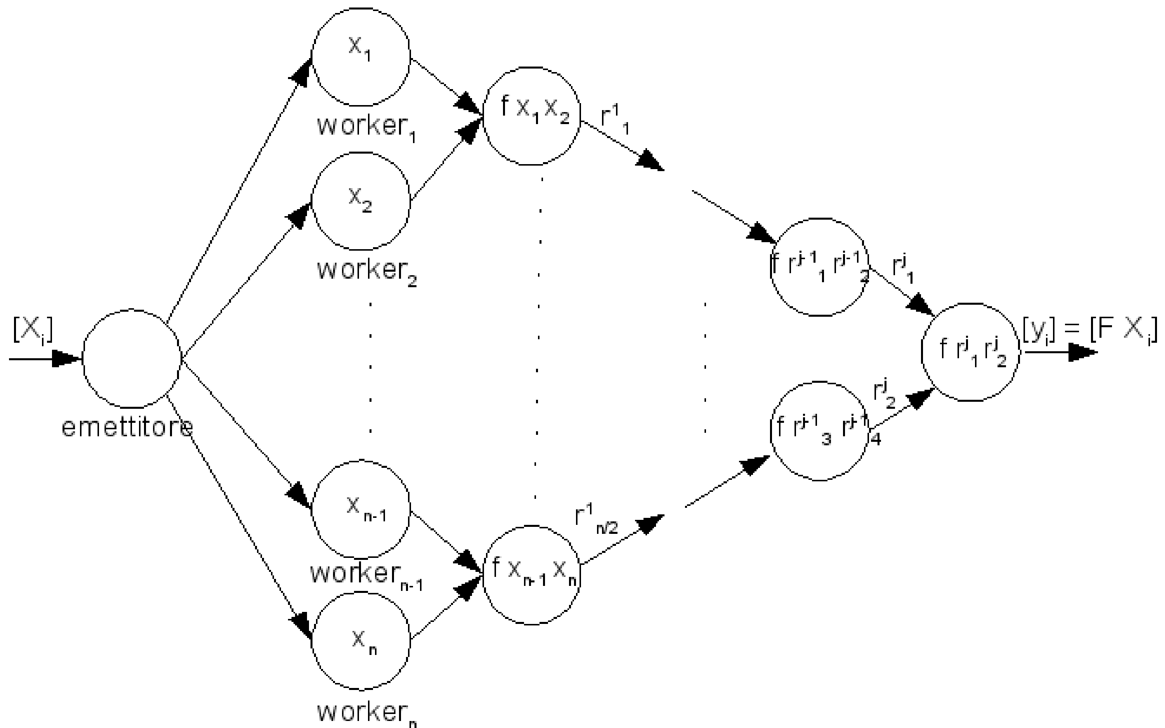

Fig. 4: Lo skeleton reduce: un albero binario costituito da 2n - 1 nodi più l'emettitore (caso n = m). Con j si indica la profondità dell'albero ovvero lg n - 1, con $r^p{}_q$ si intende il risultato parziale del q-esimo nodo del p-esimo livello dell'albero

In Fig. 4 è rappresentato il caso ideale dello schema ad albero in cui l'emettitore, come nello skeleton map, distribuisce gli m elementi del vettore X_i a $n = m$ worker che costituiscono le foglie dell'albero. I worker inviano i loro dati al livello successivo dell'albero; i livelli intermedi dell'albero calcolano i risultati parziali (indicati in figura con $r^p{}_q$, con cui si intende il risultato parziale del q-esimo nodo del p-esimo livello dell'albero) di f sui dati inviati dai figli e li inviano ai nodi padri. Il nodo radice calcola il risultato definitivo a partire dai risultati ricevuti dai figli e lo inoltra sullo stream di uscita.

Il calcolo di $F\ X_i$ procede dunque attraverso $lg_2\ n$ passi corrispondenti ai livelli dell'albero; al passo p i nodi che eseguono calcolo relativo all'elemento X_i sono gli $n/2^p$ nodi del livello p. Considerando la computazione su stream, a regime, tutti i nodi eseguono calcolo; ma ogni livello eseguirà la computazione relativa ad un diverso elemento dello stream di ingresso.

Il parallelismo, per ogni X_i , deriva dall'esecuzione contemporanea al passo p della funzione f sugli $n/2^p$ nodi intermedi del livello p.

Nel caso realistico in cui n sia minore di m, ogni worker inizialmente incapsula una partizione di X_i, di ampiezza m / n , ed esegue il *fold* della funzione f sulla propria partizione prima di consegnare il risultato parziale ai successivi livelli dell'albero.

1.3.3 Stencil fisso

Dati $f : A^{2h+1} \rightarrow B$, e uno stream di ingresso di vettori $X_i = [x_1,...., x_m]$ di componenti di tipo A, per ogni X_i , lo skeleton *stencil fisso* calcola $F : A^m \rightarrow B^m$, restituendo uno stream di tipo B^m di elementi:

$$Y_i = F X_i = [f(x_1, x_{1-h}, x_{1-h+1},\dots, x_{1+h-1}, x_{1+h}),\dots, f(x_m, x_{m-h}, x_{m-h+1},\dots, x_{m+h-1}, x_{m+h})].$$

dove h è un intero.

Ogni elemento dell'array di uscita è calcolato in funzione del corrispondente elemento dell'array di input e di alcuni $(2 * h)$ elementi vicini.

In Fig. 5 vediamo uno stencil fisso nel caso in cui $h = 1$ e i processori virtuali sono $n = m$. Il generico processore virtuale worker_j incapsula l'elemento x_j dell'array di ingresso e calcola y_j (*owner computes rule*); ai fini del calcolo della funzione $f(x_j , x_{j-1} , x_{j+1})$ il worker i-esimo deve ricevere dai due vicini, il worker (j-1)-esimo e il worker (j+1)-esimo, gli elementi che essi detengono; il worker_j , prima di riceve i due elementi vicini deve aver inviato a sua volta il proprio elemento ai due vicini, affiché anch'essi possano calcolare f. Questo scambio di elementi tra worker avviene tramite canali di comunicazione tra di essi, mostrati in figura per il generico worker_j. Il termine stencil si riferisce proprio alla configurazione dei canali di comunicazione tra i worker.

Per chiarificare vediamo lo pseudo-codice del generico worker: utilizziamo, a meno di alcuni dettagli, il formalismo di linguaggio a scambio di messaggi (di seguito LC) presente in [Van09]. I canali di comunicazione sono rappresentati da variabili di tipo `channel`; con la primitiva `send (msg, ch)` si effettua l'invio del messaggio

`msg` sul canale `ch`; con la primitiva `receive (msg, ch)` si effettua la ricezione del messaggio `msg` sul canale `ch`. Nello pseudo-codice vediamo tre array di canali, uno rappresentante i canali tra emettitore e i worker per la distribuzione degli elementi di Xi (`emitter`), un'array dei canali di comunicazione verso nord (`north`), e un array di canali di comunicazione verso sud (`south`). In particolare il j-*esimo* processore virtuale `worker[j]` ha due canali in ingresso, uno da nord (`north[j]`) e uno da sud (`south[j-1]`), e due canali in uscita, uno verso nord (`north[j-1]`) e uno verso sud (`south[j]`). La configurazione dei canali è mostrata anche in Fig. 5.

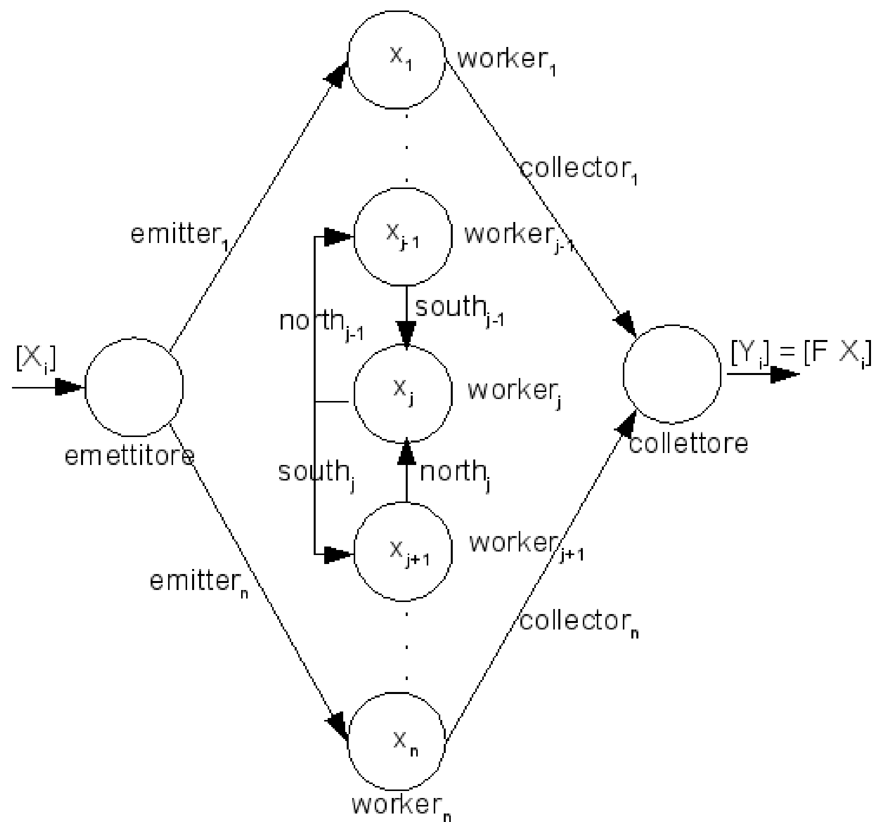

Fig. 5: Stencil fisso con h = 1 e n = m. Sono mostrati anche i nomi dei canali come etichette sugli archi che li rappresentano.

```
1 channel [N] emitter; //array dei canali tra workers ed
     emettitore
2 channel [N] collector; //array dei canali tra workers ed
     collettore
```

```
3 channel [N] north; channel [N] south; //array di canali
    per lacomunicazione tra worker
4 worker[j] ::{
5  channel in emitter[j], north[j], south[j-1];
6  channel out north[j-1], south[j], collector[j];
7  receive (xj, worker[j]); //ricezione dell'elemento dall'
     emettitore
8  send (xj, north[j-1]);//invio al vicino a nord
9  send(xj,  south[j]); //invio al vicino a sud
10 receive(xj-1,  north[j]); //ricezione dal vicino a nord
11 receive(xj+1,  south[j-1]); //ricezione dal vicino a sud
12 send ((f (xj, xj-1, xj+1) , collector[j]); //invio del
     risultato al collettore
13 }
```

Anche nel caso dello stencil fisso nella pratica i processori reali sono $n < m$, per cui ogni worker incapsula una partizione di X_i ampia m / n e invia (e riceve) ai vicini gli elementi di bordo della partizione (il primo e l'ultimo nel caso $h = 1$).

Generalmente computazioni di tipo stencil si trovano all'interno di un ciclo; se gli indici degli elementi da scambiare dipendono in modo statico dall'indice dell'iterazione si parla di *stencil variabile*, in quanto la configurazione delle comunicazione varia ad ogni iterazione (lo skeleton reduce è una computazione di tipo stencil variabile); se gli indici degli elementi da scambiare dipendono in modo dinamico dall'indice dell'iterazione (secondo una funzione non predicibile staticamente) di parla di *stencil dinamico* (a differenza degli altri due che sono detti *stencil statici*).

Molte volte la condizione di terminazione del ciclo dipende dai valori dell'array globale risultati ad ogni iterazione, ovvero la condizione di terminazione è spesso espressa da una reduce dei risultati di ogni iterazione.

1.4 Gli skeleton di controllo

Oltre alle forme di parallelismo vere e proprie viste sopra esistono degli skeleton che hanno una semantica sequenziale, ma che sono usati per controllare e comporre gli skeleton stream e data parallel canonici. Un esempio classico è lo skeleton che itera l'esecuzione di altri skeleton.

1.4.1 Loop

Dati $f : A \rightarrow A$, $cond : A \rightarrow bool$, e x_i elemento dello stream di ingresso di tipo A, per ogni x_i , lo skeleton *loop f cond* x_i calcola la funzione $F = f^{\,k} :$ A $\rightarrow$ A , restituendo lo stream di tipo A di elementi

$$y_i = f^{\,k}\, x_i = repeat\ x_i := f\ x_i\ until\ not\ cond\ x_i$$

ovvero calcola ripetutamente (a partire da x_i) la funzione f al risultato dell'iterazione precedente , finché la funzione *cond* , applicata al risultato corrente non restituisce *true.* Con $f^{\,k}$ si indica la ripetizione della funzione f per k volte; in questo caso il numero k è dato dalle iterazioni che occorre ripetere per raggiungere la condizione di terminazione del ciclo.

In Fig. 6 diamo una rappresentazione grafica di questo skeleton.

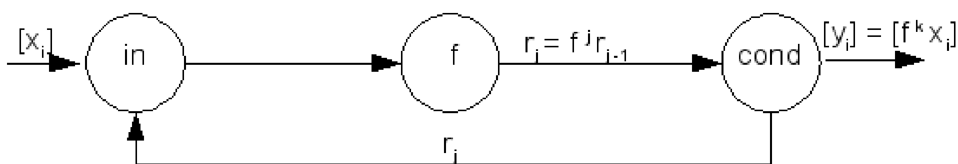

Fig. 6: Un loop che calcola F x_i : il modulo in riceve gli elementi dello stream e li inoltra a f per il calcolo della prima iterazione; cond riceve i risultati parziali r e se cond r = true invia il risultato sullo stream di uscita; altrimenti lo invia a in, che riceve da cond con priorità massima, e rinvia a f per l'iterazione successiva

Il modulo che calcola f può essere a sua volta parallelizzato secondo uno degli skeleton già visti stream o data parallel.

1.5 Modello analitico dei costi

Ad ogni skeleton è possibile associare un modello che descrive le sue prestazioni; un'equazione che esprime un'approssimazione del *tempo di completamento* del calcolo che lo skeleton esegue su tutti gli elementi dello stream (se presente). Questo modello è espresso in termini di parametri che dipendono dall'architettura concreta su cui lo skeleton è eseguito.

Conoscendo il modello dei costi e riuscendo a determinare una predizione sulle prestazioni degli skeleton è possibile:

- determinare il grado di parallelismo ottimo;
- valutare quale skeleton scegliere tra più possibili skeleton adatti alla parallelizzazione di uno stesso modulo sequenziale;
- determinare, applicando i concetti della teoria delle code [Kle76], l'esistenza di *colli di bottiglia* tra i processori virtuali (ovvero processori virtuali più lenti degli altri) che compongono uno skeleton e applicare un'ulteriore parallelizzazione a tali processori.

Oltre al tempo di completamento esistono altri importanti indicatori delle prestazioni. In caso di skeleton operanti su stream, è importante conoscere il *tempo di servizio*, ovvero il tempo che trascorre tra l'inizio dell'elaborazione di due elementi successivi dello stream. In altri casi può interessare conoscere la *latenza*, ovvero il tempo necessario ad elaborare il singolo elemento dello stream.

E' importante anche poter confrontare il tempo si servizio del modulo parallelizzato con il tempo di servizio della versione sequenziale da cui deriva, esprimendolo sotto forma di rapporto tra i due: si parla in questo caso di *scalabilità*, che indica di quanto viene migliorata la computazione sequenziale tramite la parallelizzazione.

I processori virtuali/reali sono connessi tra loro da dei canali; attraverso questi canali i processori virtuali/reali cooperano secondo un modello a scambio di messaggi. Ai fini della predizione delle prestazioni è importante conoscere la *latenza di comunicazione* dei messaggi su questi canali; la latenza di comunicazione varia in base a

come sono implementati fisicamente i canali (dall'architettura concreta), ma è caratterizzabile analiticamente rispetto all'architettura astratta. Indichiamo con *Tsend (M)* il tempo che occorre per trasferire un messaggio di lunghezza M tra due processori virtuali; possiamo scrivere la seguente equazione che caratterizza la latenza di comunicazione sull'architettura atratta.:

$$Tsend(M) = Tsetup + M * Ttrasm$$

Quest'equazione, che rappresenta la latenza di comunicazione, indica che il trasferimento di un messaggio tra due processori consta di una parte fissa e di una parte che dipende linearmente dalla lunghezza del messaggio. I parametri *Tsetup* e *Ttrasm* dipendono di volta in volta dall'architettura concreta su cui lo skeleton è eseguito e dal supporto di comunicazioni che si utilizza su essa .

Vediamo l'esempio di modello dei costi dello skeleton pipeline mostrato graficamente in Fig. 1, ripreso da [Pel93].

Assumiamo che:

- lo stream di ingresso sia composto da N elementi di lunghezza *M;*
- la dimensione dei valori di uscita da tutti gli stadi sia sempre M;
- il tempo di interarrivo degli elementi dello stream sia arbitrariamente basso.

Indichiamo con Tf_j il tempo di elaborazione della funzione f_j eseguita dallo stadio_j.

Il tempo di servizio della pipeline, indicato con T_s , in caso non si possa sovrapporre il calcolo alle comunicazioni, è dato da:

$$T_s = max\ (Tf_j) + Tsend\ (M)$$

La latenza della pipeline, indicata con L , è data da:

$$L = Tf_1 + Tsend(M) + Tf_2 + Tsend(M) + ... + Tf_n + Tsend(M)$$

Il tempo di completamento della pipeline è composto da un tempo *transitorio* di riempimento della pipeline, coincidente col passaggio attraverso tutti gli stadi della pipeline del primo elemento dello stream (la latenza), e dalla successiva elaborazione parallela degli altri elementi dello stream:

$$T_c = L + (N\text{-}1) * T_s$$

La scalabilità, indicata con s, è :

$$s = max(Tf_i) \,/\, T_F$$

dove T_F esprime il tempo di servizio della funzione sequenziale composta dai singoli Tf_i

Nonostante sia un aspetto importantissimo degli skeleton il modello dei costi non sarà trattato nel corso della tesi.

1.6 Sommario

Abbiamo fin qui visto la definizione di skeleton sia stream che data parallel, alcuni degli skeleton più comuni dando loro una semantica informale e una rappresentazione grafica. Abbiamo accennato al modello dei costi, e visto come sia una caratteristica molto importante degli skeleton.

Nel prossimo capitolo vedremo come gli skeleton abbiano da subito avuto un enorme successo in ambito accademico e le direzioni che hanno preso i diversi sistemi che li implementano nel corso degli anni. Dopo una discussione generale vedremo anche alcuni esempi particolari.

Capitolo 2

Gli skeleton nella letteratura

Dal 1989 in poi in letteratura troviamo diversi esempi di sistemi che consentono di scrivere programmi paralleli usando gli skeleton. I sistemi esistenti si differenziamo in base all'ambiente di programmazione offerto, all'implementazione che viene data agli skeleton e alle architetture a cui sono indirizzate, in particolare sul supporto alle comunicazioni che utilizzano.

2.1 Ambiente di programmazione

In letteratura si distinguono due tendenze relative all'ambiente di programmazione offerto dai diversi sistemi: da una parte esistono linguaggi per la programmazione parallela strutturata, che offrono gli skeleton come costrutti primitivi; dall'altra esistono librerie, che espongono gli skeleton tramite particolari interfacce.

2.1.1 Linguaggi per la programmazione parallela strutturata

In [Col91] gli skeleton sono proposti come il fondamento teorico di un nuovo paradigma di programmazione.

L'idea dei linguaggi di programmazione a skeleton è quella di fornire al programmatore dei costrutti, corrispondenti alle diverse forme di parallelismo, analoghi ai costrutti primitivi dei linguaggi di programmazione classici, come un *for* o un *if* .

Gli skeleton diventano cotrutti primitivi del linguaggio e sono l'unico modo per esprimere computazioni parallele; per questo si parla di programmazione parallela strutturata.

Ad esempio, volendo scrivere un programma parallelo costituito da un farm i cui worker calcolano la funzione f il programmatore deve solo scrivere un codice del tipo:

```
1 farm seq f
```

Volendo comporre più skeleton, ad esempio per ottenere un farm i cui worker sono costituiti da una pipeline di due stadi che calcolano rispettivamente le funzioni *f1* e *f2*, il programmatore deve scrivere un codice del tipo:

```
1 farm
2  pipe
3   stage1: seq f1
4   stage2: seq f2
5  endpipe
6 endfarm
```

Notiamo che è presente una parola chiave, in questo caso `seq`, che rappresenta uno skeleton sequenziale, la foglia dell'albero di skeleton, che racchiude il calcolo effettivo da espletare in sequenziale (i worker di un farm o stadi di una pipeline non ulteriormente parallelizzati).

Allo stesso modo con cui un programma sequenziale è scritto in termini dei pochi costrutti tipici dei linguaggi di programmazione classici, un programma parallelo è ora scritto in termini dei costrutti che rappresentano gli skeleton.

Con i costrutti-skeleton è possibile programmare un'applicazione parallela tramite una combinazione di skeleton primitivi, rendendo più facile lo sviluppo di applicazioni che necessitano di alte prestazioni: il programmatore si deve curare dei dettagli dell'implementazione effettiva al livello dei processi.

Il programmatore è solo tenuto a conoscere gli skeleton e la loro semantica, a saper derivare da una computazione sequenziale una sua parallelizzazione secon-

do una composizione di pochi skeleton notevoli ed esprimerla sotto forma di albero di skeleton.

Il compilatore (o l'interprete) del linguaggio genera il codice al livello dei processi che costituisce l'applicazione parallela, ed in molti casi è dotato di un modello dei costi che permette di ricavare il grado di parallelismo senza che debba essere dichiarato esplicitamente.

Ad esempio, il costrutto `farm seq f` produce un codice reale simile allo pseudo-codice riportato di seguito che rappresenta l'emettitore, il worker e il collettore del farm; utilizziamo lo stesso formalismo usato in 1.3.3. I canali `disp` e `collector` sono canali asimmetrici in uscita dal worker rispettivamente verso l'emettitore e il collettore; sul canale `disp` il worker invia il proprio indice indicando la propria disponibilità a ricevere un task (strategia on demand), sul canale `collector` il worker invia il risultato del suo calcolo; i task sono inviati dall'emettitore al worker sui canali dell'array `workers`. Per semplicità non consideriamo la condizione di terminazione esplicita, ma ogni processo cicla infinitamente.

```
1  channel workers[N];
2  emitter:: {
3   channel in disp; channel in stream_in;
4   channel out workers[*];
5   while (true) do {
6    receive (stream_in, task); //ricezione elemento dello
        stream
7    receive (disp, i); //ricezione disponibilita' dal
        worker (strategia on demand)
8    send (workers[i], task); //invia il task al worker
        disponibile
9   }
10 }
11 worker[i]:: {
12  channel in workers[i];
```

```
13  channel out disp, collector;
14  while (true) {
15   send (disp, i); //invia disponibilita' all'emettitore
16   receive (workers[i], task); //riceve task dall'
        emettitore
17   send (collector, (f task)); //applica f al task
        ricevuto e invia al collettore
18  }
19 }
20 collector::{
21  channel in collector;
22  channel out stream_out;
23  while(true) {
24   receive (coll, result); //riceve risultato da un worker
25   send (stream_out, result); //invia risultato sullo
        stream di uscita
26  }
27 }
```

Una possibile implementazione in linguaggio C in cui i processi si scambiano messaggi tramite socket unix richiederebbe molte linee di codice; in particolare la gestione dei canali di comunicazione, in questo caso i socket, costituisce buona parte del codice. Tutto questo codice può essere espresso tramite il solo costrutto `farm`. E' del tutto evidente il vantaggio che ne può trarre il programmatore in termini di produttività.

L'approccio dei linguaggi per la programmazione parallela strutturata si rivela molto efficiente, e negli anni sono stati proposti molti sistemi sotto forma di:

- linguaggi ad hoc (P^3L cite12)

- linguaggi di coordinamento che permettono la scrittura del codice sequenziale in diversi linguaggi di programmazione (SkIe [BDPV99], ASSIST [Van02])

– estensioni per la programmazione parallela di linguaggi esistenti (Eden [LOmP05], estensione di Haskell).

D'altra parte questa modalità si rivela poco flessibile per il programmatore che è limitato da un certo numero di skeleton standard che non riescono ad esprimere forme di parallelismo più complesse.

2.1.2 Librerie di skeleton

A causa dei limiti emersi nell'ambito dei linguaggi per la programmazione strutturata si è diffusa la tendenza ad utilizzare gli skeleton sotto forma di librerie.

Le librerie permettono al programmatore lo sviluppo di propri skeleton utilizzabili per la parallelizzazione di applicazioni più specifiche, pena la perdita di efficienza propria dei linguaggi compilati a skeleton.

In questo caso gli skeleton sono instanziati, non tramite costrutti primitivi del linguaggio, ma attraverso chiamate di funzioni, o creazione di oggetti, a seconda del linguaggio con cui sono implementate. Tali funzioni hanno come parametro altri skeleton e la funzione di base da calcolare. Anche nel caso di librerie è possibile che queste siano provviste di un modello dei costi che evita al programmatore di dover esprimere esplicitamente il grado di parallelismo di ogni skeleton; viceversa il gado di parallelismo è un ulteriore parametro della funzione che restituisce lo skeleton.

Usando una libreria di skeleton, se vuole ottenere la stessa applicazione della sezione 2.1.1 (farm di pipeline), il programmatore deve invocare una serie di funzioni, come in:

```
1 farm ( pipeline( [ seq(f1); seq(f2) ] ) )
```

dove `farm` e `pipeline` sono delle funzioni che prendono in ingresso altri skeleton; in questo caso ipotizziamo che la generica libreria in caso di pipeline prenda in ingresso una lista di skeleton, nell'esempio di tipo `seq`, che rappresentano gli stadi della pipeline. Anche per le librerie di skeleton valgono le stesse considerazioni fatte per

i linguaggi per la programmazione strutturata circa il risparmio di codice da parte del programmatore.

Nei sistemi che adottano questo modello di programmazione si rende necessaria l'introduzione di nuovi skeleton, con una semantica sequenziale, inseribili nella categoria degli skeleton di controllo, che hanno la funzione di coordinare la creazione e l'esecuzione degli skeleton paralleli. Questi particolari skeleton di controllo sostituiscono il compilatore nella fase di generazione della rete di processi applicativi che eseguono lo skeleton; vedremo esempi di questi skeleton relativamente a *OCamlP3l* nei paragrafi 3.3.3 e 3.3.4.

2.2 Implementazione

In letteratura si sono imposte negli anni due modalità principali di implementazione di skeleton: la modalità basata su template, più naturale e consistente nella trasformazione di uno skeleton (o di un albero di skeleton) in una rete di processi ad esso isomorfa, e la modalità basata su grafi data flow, secondo la quale gli skeleton sono trasformati in istruzioni data flow la cui esecuzione è demandata ad un insieme di interpreti.

2.2.1 Template

Dal punto di vista dell'implementazione degli skeleton si è da subito imposto un modello basato su *template*, che deriva in modo naturale dalla defininizione stessa di skeleton.

Secondo questa strategia l'albero degli skeleton che rappresenta l'applicazione è trasformato nella rete di processi applicativi che lo rappresenta; i processori virtuali sequenziali (i worker di un farm o gli stadi di una pipeline) sono instanziati con la funzione da calcolare. La rete dei processi applicativi creata è isomorfa alla rete di processori virtuali presenti nella rappresentazione grafica che abbiamo dato agli skeleton nel capitolo 1.

La topologia della rete è sempre la stessa, ovvero per ogni skeleton vengono creati gli stessi processi con gli stessi canali di comunicazione, ma è parametrica nella computazione sequenziale e nel grado di parallelismo.

Più precisamente, ogni processore virtuale - emettitore o collettore di un farm/map, worker di map/farm, stadio di una pipeline - ha una (o più d'una) implementazione predeterminata, ma parametrica, e quindi adattabile a diverse applicazioni esprimibili attraverso lo stesso skeleton.

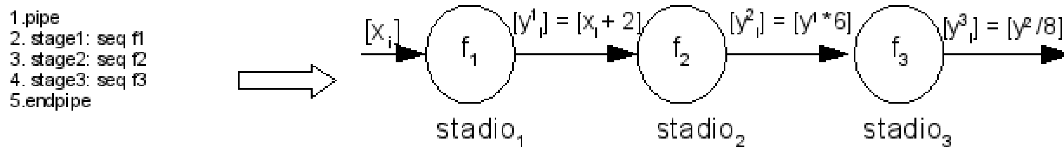

Fig. 7: Trasformazione del codice che rappresenta una pipeline in una rete di processi applicativi che lo implementa

In Fig. 7 è mostrata la trasformazione di uno skeleton pipeline in una rete di processi applicativi che lo rappresenta; più precisamente si tratta della parallelizzazione dell'esempio *eq.1* in 1.2.1. A partire dal codice avente la stessa sintassi introdotta in 2.1.1 otteniamo un grafo di processi applicativi composto da tre nodi connessi tra di loro tramite canali di comunicazione opportuni.

2.2.2 Data flow

Più di recente, in [Dan99], è stata proposto un nuovo modello, basato su grafi data-flow, secondo il quale l'albero degli skeleton è trasformato un un grafo di istruzioni data flow.

Per ogni skeleton `seq` è generata un'istruzione data flow; quando tutti i dati in ingresso sono disponibili l'istruzione è pronta per l'esecuzione (diventa un'istanza del grafo data flow). A questo scopo esiste un insieme di interpreti di istruzioni data flow, analoghi ai worker di un farm; l'istruzione viene assegnata ad uno di questi

interpreti per l'effettiva esecuzione. Su ogni interprete è in esecuzione un gestore delle istanze dei grafi data flow (contiene l'insieme delle istanze di grafo ricevute per l'esecuzione), un gestore della memoria, cioè il gestore di una cache di dati locali e l'interprete vero e proprio.

In Fig. 8 è mostrata un'implementazione data flow dell'esempio visto in 2.2.1.

Secondo i sostenitori di tale modello il vantaggio di tale concezione sta nella possibilità del supporto di adattarsi al contesto di esecuzione, ovvero nella sua dinamicità. Il modello basato su template prevede una allocazione ed un dimensionamento statico dei nodi di elaborazione che non può essere modificato a tempo d'esecuzione, oltre che una sottoutilizzazione di certi nodi in certe fasi di calcolo. Un supporto data flow consente di poter seguire strategie di schedulazione di istruzioni a interpreti che permettano un adattamento dinamico alle condizioni che si rilevano a tempo d'esecuzione, come la varianza del calcolo, l'indisponibilità di un certo nodo fisico, i fallimenti della rete.

D'altra parte il modello data flow si rivela poco adatto a computazioni non su stream rispetto al modello basato su template.

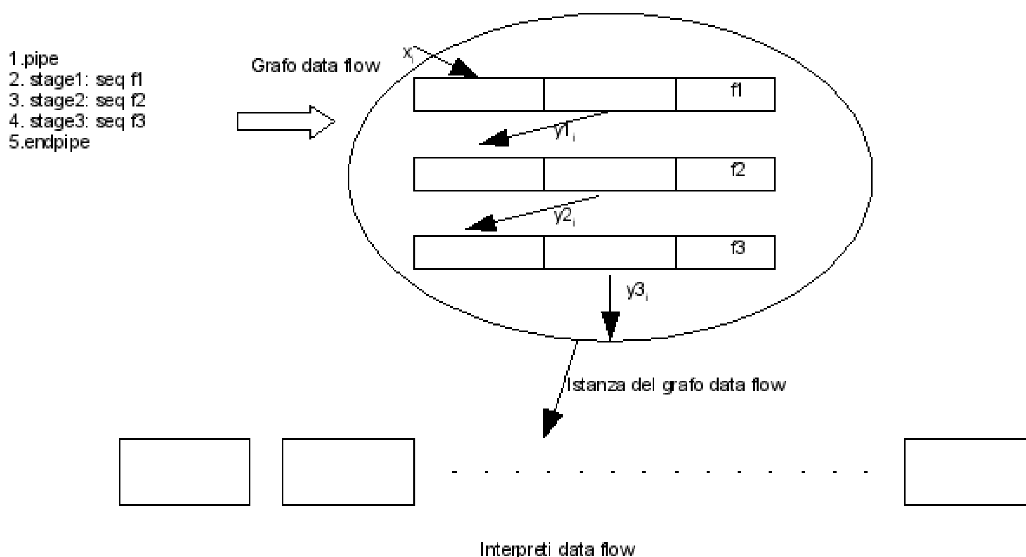

Fig. 8: Trasformazione del codice che rappresenta una pipeline in un grafo di istruzioni data flow in cui ogni istanza del grafo viene assegnata per l'esecuzione ad un interprete data flow

2.2.3 Architettura e supporto alle comunicazioni

I processori virtuali che rappresentano l'architettura astratta degli skeleton devono essere associati ad una architettura concreta in cui vengano fatti corrispondere a dei processori reali; uno skeleton (in generale un albero di skeleton) è trasformato in una rete di processi operanti su una certa architettura formata da nodi fisici di elaborazione.

Tutti i sistemi esistenti sono pensati per funzionare su determinate architetture, in particolare si differenziano come architetture obiettivo di un certo linguaggio o di una libreria per la programmazione parallela:

- architetture a memoria distribuita: cluster di PC interconnessi da una LAN (o WAN) in cui i processi comunicano tramite librerie di comunicazione ad alto livello preesistenti, come librerie Message Passing Interface (MPI), Java RMI, socket Unix.

- architetture a memoria condivisa, processori multicore con supporto alle comunicazioni dedicato a basso livello.

Nel seguito vedremo alcuni esempi di sistemi esistenti documentati in letteratura e verranno caratterizzati in base alle differenziazioni finora delineate in modo generale in questa sezione e in quelle precedenti e verranno forniti dei piccoli esempi di utilizzo da parte dell'utente. I sistemi che vedremo si differenziano in base alla loro caratteristica di essere linguaggi per la programmazione parallela strutturata o librerie di skeleton, in tutti e due i casi per avere un'implementazione basata su template o su grafi data flow, e per avere come architetture obiettivo cluster o multiprocessori o entrambi.

2.3 Sistemi esistenti

2.3.1 P^{3}L

Uno dei primi sistemi che seguirono la definizione di Cole fu P^{3}L [Pel93] [BDO+95] (Pisa Parallel Programming Language) sviluppato presso il Dipartimento di Informatica dell'Università di Pisa nei primi anni '90 .

Si tratta di un linguaggio per la programmazione parallela strutturata implementato in C/C++; la computazione sequenziale è espressa C, e le comunicazioni tra processi del supporto avvengono tramite MPI; è il primo esempio che si trova in letteratura a supportare l'annidamento di skeleton.

Il compilatore ha un'implementazione basata su template ed è dotato di un modello dei costi analitico che permette di effettuare la scelta del grado di parallelismo a tempo di compilazione.

Gli skeleton offerti come costrutti di programmazione, annidabili tra loro, sono: pipe, farm, map (e stencil fisso), reduce, loop e seq (che incapsula la computazione sequenziale).

Vediamo l'esempio già visto nella sezione 2.1.1 di farm di pipeline in questo linguaggio:

```
1  farm main in (int a, int b)  out(int c)
2   pipe w in (int a, int b) out (int c)
3    seq  f1 in (a , b) out (int c1)
4    ${ /* codice C */ }$
5    end seq
6    seq f2 in (c1) out (c)
7    ${  /* codice C */}$
8    end seq
9   end pipe
10 end farm
```

Il costrutto principale è il `farm`, denominato `main`, il cui worker `w` è di tipo pipeline, ed è composto da due stadi sequenziali. Vediamo che per ogni skeleton è necessario indicare esplicitamente il tipo dei parametri di input - `in (int a, int b)` - e di output - `out (int c)`. Il codice sequenziale è inserito tra i simboli `${` e `}$`.

E' di particolare interesse per lo svolgimento della tesi la sintassi introdotta da P3L per indicare la distribuzione dei vettori/matrici sui processori virtuali, organizzati in un vettore di m dimensioni, e la scrittura del codice sequenziale in termini dei dati in ingresso ai processori virtuali.

Più precisamente è possibile indicare la distribuzione dei dati (vettori o matrici bidimensionali) utilizzando opportunamente gli indici dei vettori/matrici (con la sintassi `*i`).

Ad esempio, volendo calcolare in parallelo il prodotto tra due matrici n X n è possibile distribuire a n^2 processori virtuali una riga della prima e una colonna della seconda ottenendo da ognuno l'elemento corrispondente della matrice di output. Il codice P3L relativo al prodotto di due matrici 10 x 10 è il seguente:

```
1  seq prod in (int a[10], int b[10]) out (int z)
2  ${
3   z= 0;
4   for (int k = 0; k<10; k++) z = z + a[k]*b[k]; //prodotto
        riga per colonna
5  }$
6
7  map prodmatrix in (int a[10][10], int b[10][10]) out (int
       c[10][10])
8  nworker 2,2
9   prod in (a[*i][], b[][*j]) out (c[*i][*j])
10 end map
```

In particolare la sintassi

```
in (a[*i][], b[][*j]) out (c[*i][*j])
```

indica che ogni elemento di indice i , j della matrice di output è calcolato in termini della i-esima riga della matrice `a` e della j-esima colonna della matrice `b`; `*i` e `*j` sono dei quantificatori universali.

Il generico processore virtuale i , j (in tutto sono n^2 e cioè 10^2) ha in ingresso la riga i della matrice `a` e la colonna j della matrice `b` (quindi due vettori) e restituisce l'elemento i , j della matrice `c` di output.

Da notare che la funzione sequenziale `prodmatrix` è scritta in termini dei parametri in ingresso al generico processore virtuale, cioè due vettori, e restituisce un intero.

Con la clausola

```
nworker 2,2
```

si indica che la matrice processori virtuali è associata a una matrice di worker reali più piccola (2x2 anziché 10x10), ma con lo stesso numero di dimensioni. In questo caso ogni processore reale ingloba una sotto-matrice di 5 x 5 processori virtuali e opera su blocchi di cinque righe di `a` e cinque colonne di `b`.

2.3.2 eSkel

Eskel [MCGH05] (Edimburgh Skeleton Library) è una libreria per la programmazione parallela sviluppata presso l'Università di Edimburgo. La prima versione fu sviluppata nel 2002 proprio da Murray Cole, la seconda, alla quale ci riferiamo, dallo stesso Cole e il suo gruppo di ricerca.

Si tratta di una libreria C con implementazione basata su template; le comunicazioni tra processi avvengono tramite MPI e offre solo due skeleton: Pipeline e Deal (farm con emettitore round robin), ottenuti tramite due funzioni omonime; altri skeleton data parallel e il farm classico, presenti nella prima versione, non sono ancora supportati.

Rispetto alla prima versione e ad altre librerie esistenti si pone come obiettivo una maggiore flessibilità; vuole offrire al programmatore la possibilità di poter decidere, attraverso alcuni parametri degli skeleton (ed altre funzioni), le modalità di interazione e l'annidamento tra skeleton.

Oltre alla possibilità di annidare in modo "classico" e statico due skeleton, eSkel permette di poter definire degli annidamenti "transitori" ed interazioni tra skeleton "esplicite" [BC05].

Ad esempio, è possibile definire una pipeline in cui un particolare stadio può, a seconda di alcune condizioni definite esplicitamente a programma e valutate a tempo d'esecuzione, non inoltrare il risultato allo stadio successivo - e si parla in questo caso di interazione esplicita - oppure può inoltrarlo ad un "sotto-pipeline" - e si parla in questo caso di annidamento transitorio.

Questa flessibilità si traduce inevitabilmente in una maggiore complessità della libreria e del codice sequenziale che l'utente deve scrivere.

Il codice utente, infatti, non si limita alla sola computazione "secca" sul dato in ingresso e alla dichiarazione degli skeleton, ma contiene l'esplicita programmazione degli aspetti sopra citati; è prevista da parte del programmatore l'adozione (e comprensione) di un particolare modello dei dati e di esecuzione.

In particolare ogni interazione tra skeleton avviene tramite lo scambio di dati atomici, di diverso numero a seconda della semantica dello skeleton (in realtà essendo implementati solo i due skeleton citati sopra si scambiano sempre un unico atomo). Gli atomi sono contenuti in una struttura denominata `eskel_molecule_t` e acceduti nel codice tramite questa. Ogni stadio di `Pipeline` e worker di `Deal` riceve in ingresso e restituisce in output un `eskel_molecule_t`.

Inoltre l'allocazione dei nodi agli stadi della pipeline o ai worker è determinato a programmazione tramite il valore della funzione `myrank`, con pattern di utilizzo simile a quella del *pid* di un processo.

Il valore stabilito in base al `rank`, i puntatori alle funzioni che rappresentano gli stadi di una pipeline o i worker di un deal, le modalità di interazione per ogni stadio o worker, il numero di stadi o worker, la dimensione di input e output sono alcuni dei parametri da passare alle funzioni `Deal` e `Pipeline`.

Vediamo come definire un pipeline di due stadi che applicano le funzioni `f1` e `f2` agli elementi dello stream di ingresso:

```
1  //codice del primo stadio
2  eSkel_molecule_t * FirstStage (eSkel_molecule_t *in) {
3   ((int*) in->data[0])[0] = f1 (((int) in->data[0])[0]);
4
5   return in;
6  }
7  //codice del secondo stadio
8  eSkel_molecule_t * SecondStage (eSkel_molecule_t *in) {
9   ((int*) in->data[0])[0] = f2 (((int) in->data[0])[0]);
10
11  return in;
12 }
13 //applicazione principale
14 int main (int argc, char *argv[])
15 {
16  .... //dichiarazione e preparazione di alcuni parametri
       dello skeleton
17
18  Imode_t imodes[STAGES]= {IMPL, IMPL}; //modalita' di
       esecuzione degli stadi del pipeline
19
20  //gli stage del pipeline
21  eSkel_molecule_t *(*stages[STAGES])(eSkel_molecule_t *)
       = {FirstStage, SecondStage};
22
23  MPI_Init(&argc, &argv); SkelLibInit();
24  inputs = ....//generazione degli input
25  //associazione dello stadio del pipeline al processo in
       esecuzione
26  if (myrank()<2) mystagenum = 0;
27  else mystagenum = 1;
```

```
28   results = .... // creazione buffer di uscita
29   // chiamata dello skeleton Pipeline
30   Pipeline (STAGES, imodes, stages, mystagenum, BUF, ... ,
         ... , (void *) inputs, INPUTSZ, INPUTNB, (void *)
       results, INPUTSZ, ... , INPUTNB*INPUTSZ, ...);
31   MPI_Finalize();   return 0;
32  }
```

Il programma principale è contenuto nella funzione `main` (linee 14-32); le prime linee di codice della funzione sono dedicate alla preparazione dei parametri da passare alla funzione `Pipeline` (chiamata alla linea 30), ad esempio l'array che contiene le modalità di interazione tra gli stadi (linea 18), in questo caso implicita (`IMPL`) e l'array di stadi del pipeline che contiene le funzioni che li rappresentano (linea 21). Il numero ordinale dello stadio del pipeline (variabile `mystagenum`) è deciso tramite la funzione `myrank` (linee 26-27) e costituisce anch'esso un parametro della funzione `Pipeline`. Tra gli altri parametri vi sono il numero totale degli stadi (`STAGES`), l'array `input` che rappresenta lo stream di iingressoi, il buffer per i risultati (`outputs`), la dimensione del singolo elemento (`INPUTSZ`), e il numero degli elementi di input (`INPUTNB`). I due stadi della pipeline sono espressi dalle funzioni `FirstStage` e `SecondStage` (linee 1-13); vediamo che ognuna ha come parametro in ingresso un`eSkel_molecule_t *` , chiamato `in`, che contiene, all'interno del campo `data`, il singolo elemento dello stream; una volta svolte le elaborazioni su tale elemento viene memorizzato nella `struct in` e restituito.

Da notare la complessità e la lunghezza del codice per esprimere uno skeleton semplice come una pipeline di due stadi.

2.3.3 Skeleton in MetaOcaml

In [SF08] è mostrato come applicare i principi della metaprogrammazione e della valutazione parziale alla programmazione a skeleton. In particolare è definito un linguaggio, chiamato SKL, implementato da una libreria scritta in MetaOcaml [Met]

, un'estensione di OCaml per la programmazione multistage; le comunicazioni tra processi avvengono tramite MPI.

In generale i linguaggi multistage, introdotti da Walid Taha in [Tah99] , consentono la generazione ed esecuzione di programmi, offrendo al programmatore la possibilità di trattare esplicitamente, tramite costrutti del linguaggio, la valutazione parziale e la specializzazione del codice.

Questi linguaggi permettono di scrivere codice altamente generico che non risenta di particolari overhead tramite costrutti appositi per la generazione, compilazione ed esecuzione di codice specializzato in base a valori calcolabili in momenti diversi (stage) della computazione.

In particolare in MetaOCaml esistono tre operatori : brackets (`.< >.`), che incapsulano un frammento di programma; escape (`.~`) , che ingloba un frammento di programma in un frammento più grande; run (`!.`), che compila ed esegue un programma (valutando i frammenti contenuti fra brackets).

La libreria SKL, nel parsare l'albero di skeleton, sfrutta questi operatori per produrre codice specializzato una volta per tutte per ogni nodo fisico che compone un particolare skeleton senza che la specializzazione debba avvenire a tempo d'esecuzione ogni volta che un certo frammento di codice deve essere eseguito. Sostanzialmente i template implementativi sono ottenuti tramite valutazione parziale. Come vedremo in seguito, lo stesso scopo viene raggiunto in OcamlP3l tramite marshalling di chiusure.

Gli skeleton paralleli offerti da SKL sono la pipeline e il farm; ad esempio, per ottenere un farm i cui worker sono costituiti da un pipeline di due stadi l'utente deve scrivere questo semplice codice:

```
1 let prod () = ... (* funzione che genera i dati *)
2 let f1 x = .... (* funzione calcolata dal primo stadio
    del pipeline interno al farm *)
3 let f2 x = .... (* funzione calcolata dal secondo stadio
    del pipeline interno al 5.farm *)
4 let cons x = ...(* funzione che consuma i dati *)
5 let prog =
```

```
6   Pipe [seq prod; (Farm ( 5, Pipe [seq f1; seq f2])); seq
        cons] in
7 run prog;;
```

vediamo che il programma principale (linea 6) è una pipeline i cui stadi iniziale e finale generano lo stream di ingresso al farm e consumano lo stream di uscita dal farm; il `farm` è una funzione che come parametri ha il numero dei worker e lo skeleton che rappresenta il worker; la pipeline prende in ingresso una lista di skeleton che rappresentano i suoi stadi (in questo caso `seq`).

2.3.4 SkeTo

SkeTo (Skeletons in Tokyo) è una libreria C++ basata su MPI, sviluppata dal 2005 presso l'università di Tokyo presentata in [MIEH06].

Questo sistema costituisce un'applicazione della teoria degli algoritmi costruttivi allo sviluppo di programmi paralleli basati su strutture dati quali liste, matrici ed alberi.

Seguendo la teoria degli algoritmi costruttivi, in generale, la struttura dei programmi deriva dalla struttura dei dati che manipola. Tali strutture dati hanno una precisa definizione algebrica e per ognuna è definito matematicamente un pattern di computazione, detto omomorfismo. L'omomorfismo gode di certe proprietà che permettono la costruzione a partire da esso di altre regole che rappresentano i passi della computazione da eseguire sulla particolare struttura dati.

Questa teoria è applicabile agli skeleton data parallel in quanto possono essere visti come computazioni parallele su determinate strutture dati; uno skeleton è un'omomorfismo.

Secondo questo approccio la libreria definisce degli skeleton data parallel di base - map, reduce, scan (parallel prefix) - operanti su strutture dati ben definite costruttivamente; permette inoltre l'estensibilità degli skeleton base, da parte dell'utente, in base al principio di componibilità tra omomorfismi, e quindi tra skeleton preesistenti.

SkeTo tratta tre tipi di strutture dati “parallele”: le liste, le matrici (di due dimen-

sioni), e gli alberi. Per ognuna di queste è implementata una specifica struttura dati definita in modo costruttivo e gli skeleton operanti su di essa.

In particolare sono definite le strutture *DList* (implementata con la classe *dist_list*), *DMatrix* (implementata con la classe *dist_matrix*), *DTree* (implementata con la classe *dist_tree*).

A titolo d'esempio mostriamo la definizione costruttiva di una *DList* :

```
DList a = Empty | Singleton a | DList a ++ DList a
```

secondo la quale una lista di elementi di tipo *a* è o vuota, o composta da un singolo elemento o data dalla concatenazione di due liste dello stesso tipo. Quest'ultima definizione permette di definire la struttura dati "parallela": l'operatore associativo di composizione (++), non determina l'ordine di costruzione di una lista; quindi è possibile applicare un'omomorfismo ricorsivamente alle sottoliste che compongono la lista e poi combinare il risultato.

Per ognuna delle tre strutture dati esiste una classe che contiene i diversi skeleton come suoi metodi statici; ogni skeleton di relativo ad una diversa struttura è implementato come un template che differisce dagli altri per la distribuzione dei dati e l'organizzazione delle comunicazioni tramite MPI.

Vediamo un esempio di applicazione parallela per il calcolo della somma degli elementi contenuti in una lista usando lo skeleton reduce.

```
1 struct Sum: public skeleton::binary_function<double,
     double, double> {
2  double operator()( double x, double y ) const { return x
       + y; }
3 } add;
4 const double init = 0.0;
5 int SketoMain( int argc, char **argv ){
6  //genera lista lunga size secondo il generatore gen
7  dist_list<double> * data = new dist_list<double>(gen,
      size);
```

```
8    double sum = list_skeletons::reduce(sum, init, data);
9    //stampa a video del risultato
10   skeleton::cout << "sum: "  << sum << std::endl;
11 }
```

Il programma principale è contenuto nella funzione `SketoMain` (linee 5-11); la lista data è di tipo `dist_list` ed è riempita passando un generatore di numeri casuali e la sua lunghezza (linea 7); lo skeleton reduce relativo ad una `dist_list` è un metodo statico della classe `list_skeletons` (linea 8). Notiamo che le funzioni sequenziali da passare agli skeleton sono implementate estendendo delle *struct* parametriche, `unary_function` o `binary_function`, a seconda che abbiano uno o due parametri; in questo esempio la somma è una struct chiamata `Sum` che estende `binary_function` istanziando i parametri di tipo con `double` (linea 1). Questa scelta è stata fatta al fini di eliminare overhead dovuti a chiamate di funzioni, che ci sarebbero ad esempio in caso di passaggio di puntatori a funzione, affidandosi invece all'inlining effettuato dal compilatore sulle struct.

Parte importante della libreria è un ottimizzatore [MKI+04], implementato in OpenC++ (un preprocessore C++), che applica agli skeleton annidati presenti nel codice delle regole di riscrittura che li fondono al fine di eliminare overhead dovuti alla creazione di strutture dati intermedie. Ad esempio se uno skeleton reduce segue uno skeleton map, è possibile fondere questi due in un unico skeleton. In questo nuovo skeleton ogni worker prima applica la funzione della map a tutti gli elementi dell'array in ingresso e poi esegue i passi di reduce ai risultati, anziché procedere ad una nuova ridistribuzione dei dati.

2.3.5 BlockLib

BlockLib [AEK08] è una recente libreria C di skeleton indirizzata all'architettura del multiprocessore CELL [paIR]. Il multiprocessore CELL, che vediamo in Fig. 9, ha un'architettura a memoria condivisa, costituita da un processore PowerPC (PPE, power processing element), che coordina 8 processori vettoriali RISC (SPE, syner-

gistic processing element), dotati ognuno di una piccola memoria locale; tutti i processori sono interconnessi tra loro e alla memoria esterna principale da una struttura bus.

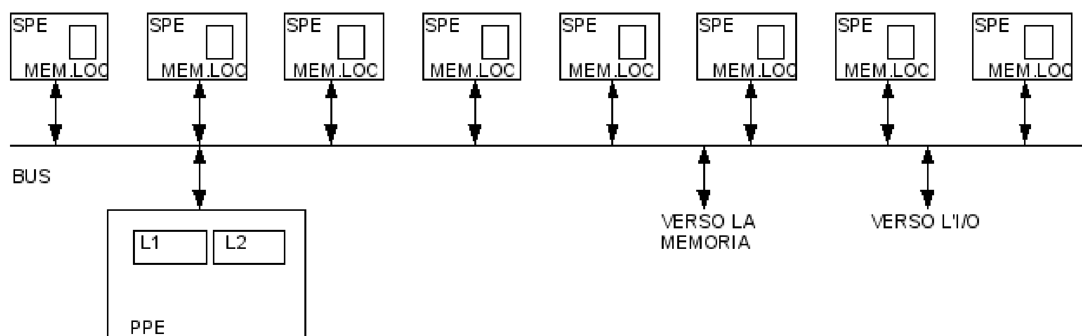

Fig. 9: Architettura di un processore CELL: 8 processori SPE slave con piccola memoria locale e un processore PPE master con cache di primo e secondo livello. Un bus collega i processori tra loro e alla memoria esterna

La gestione della memoria condivisa e i trasferimenti tra e verso i processori costituisce il punto critico di un qualsiasi supporto alla programmazione di questa architettura.

BlockLib per alcuni aspetti di gestione della memoria si poggia sull'ambiente NestedStep, un supporto a run time del modello bulk synchronous parallel (BSP) e la sincronizzazione è basata su segnali (implementati con registri speciali presenti su tutti i processori).

Il modello BSP si basa sulla nozione di "superstep"; in un superstep il calcolo e la comunicazione sono disgiunti. Ogni processo effettua i calcoli che può eseguire sui dati locali e, se necessita di dati appartenenti ad altri processi, la comunicazione avviene solo dopo che è finita la fase di calcolo su tutti i processi. Quando tutte le comunicazioni hanno avuto fine può iniziare il prossimo superstep.

BlockLib fornisce quattro skeleton data parallel, ovvero map, reduce, e due loro varianti: map-reduce: una combinazione di map e reduce in successione caratterizzata da prestazioni maggiori rispetto ad una combinazione di map e reduce; e map-with-overlap, uno stencil fisso.

Per avere prestazioni migliori la funzione sequenziale deve essere fornita dall'utente, non rispetto al singolo elemento dell'array, ma rispetto ad un blocco di elementi dell'array.

Ad esempio, volendo sommare in parallelo tutti gli elementi di un array usando lo skeleton reduce l'utente deve scrivere il seguente codice:

```
1  float sum (float *x, int dim) { /*somma gli elementi di x
       di dimensione dim*/
2   float sum = 0;
3   for (int i = 0; i<n; i++){
4    sum+= x[i];
5   }
6   return sum;
7  }
8  /*dichiarazione dell'array x di input e della dimensione
      N del blocco*/
9  .......
10 float result = sDistReduceLocalFunc(&sum, x, N);
```

Come si vede (linee 1-7) la funzione `sum` somma gli elementi di un array (un blocco dell'array globale) e viene passata alla funzione che implementa lo skeleton (`sDistReduceLocalFunc`) tramite puntatore a funzione (linea 10). E' comunque possibile esprimere la funzione `sum` come funzione operante su soli due parametri `float`, ma questa va passata ad una funzione che rappresenta lo skeleton diversa.

Al fine di avere ulteriori vantaggi sulle prestazioni la funzione sequenziale andrebbe scritta dall'utente usando delle particolari ottimizzazioni specializzate per l'architettura del CELL; per evitare all'utente di scriverle "a mano" è fornito, oltre che una libreria predefinita di funzioni di base e più avanzate, un linguaggio per la definizione di funzioni implementato tramite macro C. Sono fornite delle macro parametriche che rappresentano gli skeleton implementati, una per ogni skeleton, per ogni molteplicità di array in ingresso, per ogni tipo di dato atomico. Le macro accettano tra i parametri altrettante macro che rappresentano le funzioni da calcolare; attraverso la

macro espansione effettuata dal compilatore C viene generato il codice sequenziale ottimizzato.

Ad esempio lo skeleton map che opera su due array di ingresso è rappresentato dalla macro:

```
1 DEF_MAP_TWO_FUNC_S(name,res,cmacro1...cmacron,omacro1...
    omacrom)
```

dove *name* è il nome della funzione che verrà generata , *res* è il nome del risultato della funzione; con *cmacro1...cmacron* si intende una sequenza di macro costanti, con *omacro1...omacrom* una sequenza di macro che rappresentano le funzioni da calcolare. Volendo calcolare la funzione *f x1 x2 = (x1 + x2) * 5* a tutti gli elementi di due array di ingresso possiamo scrivere qualcosa di simile a questo:

```
1 DEF_MAP_TWO_FUNC (map_f , res,
2  BL_SCONST(c , 5.0f), //costante 5
3  BL_SADD (r1, x1, x2) //macro per la somma di due float
4  BL_SMUL(res, r1, c)) //macro per la moltiplicazione di
     due float
5 //chiamata della funzione ottimizzata creata
6 map_f (array1, array2, arrayres, N);
```

Con la macro `BL_SCONST (key, val)` si dichiarano delle costanti, in questo caso dichiariamo la costante 5. Con una macro del tipo `BL_Sfun (r, a, b)` si definisce una funzione binaria i cui parametri di ingresso sono a e b e il cui risultato è r; in questo caso usiamo le macro per la somma (`BL_SADD`) e per la moltiplicazione (`BL_SMUL`). Dall'espansione di queste macro si ottiene il codice ottimizzato per il calcolo della funzione *f* su tutti gli elementi dei due array di ingresso; l'ottimizzazione consiste, all'interno delle funzioni sequenziali, nell'utilizzo di istruzioni che effettuano certe operazioni vettoriali ad hoc per il processore CELL .

2.3.6 Muskel

Muskel è una libreria di skeleton Java, sviluppata dal 2007 presso il Dipartimento di Informatica dell'Università di Pisa [AMP07].

E' un'applicazione del modello implementativo data-flow e le comunicazioni tra processi avvengono tramite Java RMI.

Gli skeleton offerti sono Pipeline e Farm, e sono implementati con classi omonime; l'oggetto istanza di una di queste classi è una particolare istanza dello skeleton che rappresenta. Il costruttore di uno skeleton ha come parametro gli skeleton che lo compongono(al minimo uno skeleton sequenziale); lo skeleton sequenziale è una classe fornita dall'utente che estende l'interfaccia *Skeleton.*

Vediamo un semplice esempio di applicazione parallela composta da un farm i cui worker sono pipeline di due stadi:

```
1  ....
2 Skeleton prog = new Farm (new Pipeline (f1, f2));
3 Manager exec = new Manager();
4 manager.setProgram(prog);
5 manager.setContract(new ParDegree(5));
6 ...
7 manager.eval();
8 ...
```

Vediamo che l'albero di skeleton è un oggetto di tipo `Skeleton` costruito a partire da costruttori annidati (linea 2); il costruttore `Pipeline` prende un numero indeterminato di parametri di tipo `Skeleton`; `f1` e `f2` sono istanze di due sottochlssi di `Skeleton` (definite in precedenza dall'utente), che rappresentano le funzioni calcolate dai due stadi del pipeline.

Gli skeleton sono coordinati a tempo d'esecuzione dall'oggetto `Manager` (linea 3), a cui viene associato l'albero di skeleton `prog` (linea 4); il grado di parallelismo è definito esplicitamente tramite l'oggetto `ParDegree` (linea 5); l'esecuzione parte invocando il metodo `eval` di `Manager` (linea 7).

Di particolare importanza è la possibilità fornita dalla libreria di poter introdurre nuovi skeleton disegnando il grafo data-flow che lo rappresenta (anche tramite interfaccia grafica) e estendendo una particolare classe `ParCompute`. Tramite questo meccanismo è stato recentemente introdotto lo skeleton data parallel map.

2.3.7 Assist

Assist [Van02] è un linguaggio di programmazione parallela, sviluppato dal 2002 presso il Dipartimento di Informatica dell'Università di Pisa.

Questo sistema adotta una concezione di skeleton diversa da quella classica: con skeleton non si intende una particolare forma di parallelismo come quelle discusse nel capitolo 1, ma una forma di parallelismo generica. Si pone come obiettivo la possibilità di esprimere le classiche forme di parallelismo allo stesso modo di forme di parallelismo, sia task che data parallel, caratterizzate da una struttura irregolare e dinamica, anche con comportamento non deterministico, ottenendo prestazioni elevate.

Per questo linguaggio sono stati implementati i supporti a tempo d'esecuzione indirizzati a diverse architetture esistenti, sia per cluster che per multiprocessori a memoria condivisa (come il CELL); il codice sequenziale può essere espresso in C, C++ o Fortran.

Un programma ASSIST ha la struttura di grafo i cui nodi sono moduli sequenziali o paralleli (rappresentati da dei costrutti appositi) e gli archi sono astrazioni di stream. I moduli sequenziali sono i componenti di base di un grafo, hanno uno stato interno e si attivano al momento dell'arrivo di un dato sullo stream di ingresso. Il modulo parallelo (*parmod*) è il costrutto che permette di esprimere le diverse forme di parallelismo.

I concetti su cui si basa un modulo parallelo sono molto simili a quelli su cui si basava P^3L , ma a differenza di quest'ultimo non fornisce costrutti ad alto livello per gli skeleton notevoli del capitolo 1. In particolare il concetto di processore virtuale e di topologia sono qui esplicitati con una particolare sintassi, allo stesso modo dei

concetti di stream di input e stream di output. In più aggiunge i concetti di stato interno e di oggetti esterni. Il template implementativo è in questo caso il grafo di moduli, alcuni a loro volta paralleli e composti da una certa topologia di processori virtuali, ricavato a dalla compilazione del programma.

Al fine di spiegare quanto detto mostreremo l'esempio (documentato nel tutorial Assist 1.3) del prodotto di matrici riportato anche per P^3L nel paragrafo 2.3.1. Come già detto al fine di parallelizzare l'algoritmo più semplice per moltiplicare due matrici n X n (riga per colonna) sono necessari n^2 processori virtuali, ognuno dei quali opera su una riga della prima matrice e una colonna della seconda producendo l'elemento corrispondente della matrice di uscita. Il programma consiste in un grafo di quattro moduli, due sequenziali per produrre lo stream di matrici di ingresso, uno sequenziale per consumare lo stream di matrici di uscita, uno parallelo per effettuare la moltiplicazione:

```
1  generic main()
2  {
3    stream long[N][N] a; //prima matrice di input
4    stream long[N][N] b; //seconda matrice di input
5    stream long[N][N] c; //matrice di output
6    prod (output_stream a); //modulo sequenziale produttore
         stream della prima matrice
7    prod (output_stream b); //modulo sequenziale produttore
         stream della seconda matrice
8    prodmatrix (input_stream a, b output_stream c); //
        modulo parallelo per il prodotto di matrici
9    cons (input_stream c); //modulo sequenziale consumatore
         della matrice di output
10 }
```

Vediamo che il programma inizia con la dichiarazione degli stream e col loro tipo (linee 3-5); di seguito vengono richiamati i moduli che compongono il grafo; il collegamento tra i nodi del grafo è esplicitato dalle parole chiave `input_stream` e

`output_stream` nei parametri dei moduli (che sono gli stream dichiarati in precedenza): per esempio la parola chiave `output_stream` davanti al parametro a nel modulo `prod` (linea 6) e la parola chiave `input_stream` davanti allo stesso parametro del modulo `prodmatrix` indica un arco orientato da `prod` a `prodmatrix` che rappresenta lo stream di matrici `a`.

Tralasciamo la definizione dei moduli sequenziali e vediamo una versione semplificata del modulo parallelo `prodmatrix`:

```
1  parmod prodmatrix (input_stream long a[N][N], long b[N][N
     ], output_stream long c[N][N]) {
2   //sezione topologia
3   topology array [i:N][j:N] Pv;
4   //sezione stato interno ...
5   attribute long C[N][N] scatter C[*i][*j] onto Pv[i][j];
6   //sezione input
7   do input_section {
8    guard: on , , a && b {
9     a[*i0][*j0] scatter to Pv[i0][j0];
10    b[*i1][*j1] scatter to Pv[i1][j1];
11   }
12  } while (true)
13 //sezione processori virtuali
14  virtual_processors {
15   elab (in guard out ris) {
16    VP i, j { //calcolo per ogni processore virtuale
17     ...
18     rowxcol (in a[i][], b[][j] out C[i][j]);
19     ...
20     assist_out(ris, C[i][j]); //invia il risultato dell'
         elaborazione sullo stream di uscita
21    }
```

```
22    }
23  }
24   //sezione output
25   output_section {
26    collects ris from ALL Pv[i][j] {
27     int temp_c[N][N];
28     ... //codice per ricomposizione della matrice di ouput
29     assist_out(c, temp_c);
30    }
31   }
32  }
33   //procedura sequenziale in C++
34   proc rowxcol(in long A[M], long B[M] out long C)
35   $c++{
36    // prodotto riga per colonna
37    register long count=0;
38    for (register int k=0; k<M; ++k)
39     count += A[k]*B[k];
40    C = count;
41  }c++$
```

Nella sezione topologia, con la parola chiave `topology` si dichiara la disposizione dei processori virtuali, dando un nome a tutto l'insieme; in questo caso sono disposti ad array bidimensionale `N x N` e si dichiarano le variabili che si usano per indicizzarli, in questo caso `i` e `j` (linea 3). Nella sezione stato interno, con la parola chiave `attribute` si dichiara lo stato interno del modulo e la strategia di distribuzione sui processori virtuali; in questo caso è indicata dalla parola chiave `scatter`, con la quale si indica che ogni elemento di `c` è distribuito al corrispondente processore virtuale (la corrispondenza è stabilita dagli indici `i` e `j`) (linea 5). Per altri casi sono definite altre strategie, come `broadcast`, `on_demand`, `multicast`. Nella sezione di input, con la parola chiave `input_section` si introduce la sezione in cui vengono definite

le guardie e la eventuale strategia di distribuzione dei dati in ingresso. La guardia indica la condizione al verificarsi della quale una certa elaborazione parallela può avere inizio; in questo caso il modulo parallelo si attiva alla ricezione di entrambe le matrici di ingresso (linee 7-8). In altri casi è possibile esprimere condizioni non deterministiche (comando alternativo del modello CSP). Nella sezione processori virtuali, con la parola chiave `virtual_processor` si introduce la sezione in cui è definita l'elaborazione di ciascun processore virtuale (è possibile anche esprimere un'elaborazione diversa per ogni processore virtuale assegnando dei valori particolari agli indici `i , j`); in questo caso l'elaborazione parallela è denominata `elab`, prende come parametri la guardia che la attiva e il risultato (`ris`) che deve ritornare (linea 15). In seguito (linea 16) si definisce l'elaborazione di ciascun processore virtuale dell'insieme `VP` precedentemente dichiarato; in questo caso per ogni valore di `i` e `j` viene effettuata la stessa elaborazione sequenziale `rowxcol` (linea 18) che prende in ingresso una riga di `a` e una colonna di `b` (con la sintassi molto simile a quella di P^3L) e restituisce l'elemento corrispondente di `c` (la procedura sequenziale è definita alle linee 34-41). Nella sezione output, con la parola chiave `output_section` si introduce la sezione in cui viene definita la strategia per raccogliere gli elementi calcolati dai processori virtuali, in questo caso (linea 26) si stabilisce che si devono ricevere (`collects`) i risultati parziali (`ris`) da tutti i processori virtuali (`from ALL Pv[i][j]`). Il comando `assist_out` indica l'invio del valore calcolato sullo stream di uscita (composto dai diversi `ris`).
Di particolare importanza sono due costrutti, che nell'esempio non abbiamo visto, che permettono di esprimere stencil fissi e variabili e una condizione di terminazione globale in caso di calcoli iterativi. Questi costrutti sono chiamati rispettivamente `for` e `while`, analoghi agli omonimi costrutti dei linguaggi imperativi, ma hanno una semantica parallela e sono utilizzabili in una sezione `virtual_processor`.

2.4 Conclusioni

In generale lo scopo della parallelizzazione di un'applicazione è aumentare le prestazioni rispetto alla versione sequenziale funzionalmente equivalente. Idealmente un'appli-

cazione parallelizzata con grado di parallelismo n, dovrebbe avere prestazioni n volte migliori rispetto alla stessa applicazione sequenziale. Nella pratica questioni implementative, come copie di dati, e le comunicazioni tra processi, provocano degli overhead tali da non poter raggiungere le prestazioni ideali, se non addirittura tali da avere un deterioramento rispetto al caso sequenziale.

Si rende necessario un bilanciamento tra i costi di calcolo e i costi di comunicazione e gli altri costi (che non sussistono nel caso sequenziale) tale da poter arrecare vantaggio alla parallelizzazione effettuata. Per questo alcuni skeleton si rivelano più adatti a calcoli di grana grossa piuttosto che grana fine. In generale gli skeleton data parallel, in assenza di stream, sono adatti per calcoli a grana grossa e alto grado di parallelismo, mentre quelli stream parallel sono adatti anche a calcoli di grana medio-fine (a parità di condizioni). Si noti che dei sistemi visti, quelli che forniscono skeleton sia data parallel che stream parallel sono pochi.

Data un'applicazione da parallelizzare, il modello dei costi associato agli skeleton ci permette di dimensionare lo skeleton e scegliere, tra quelli applicabili al problema, quello che riesce ad avere le prestazioni migliori; ma rispetto agli overhead sopra citati il massimo che si può fare è trovare delle ottimizzazioni atte ad esempio a limitare il numero di copie e avere un supporto alle comunicazioni quanto più performante.

Un obiettivo molto importante della programmazione parallela tramite skeleton è fornire al programmatore un ambiente di programmazione che astragga quanto più possibile i dettagli relativi all'implementazione del parallelismo. E' importante fornire un ambiente che richieda all'utente di programmare il solo codice relativo al calcolo sequenziale nel modo più semplice e diretto possibile, consentendo il riuso di codice preesistente secondo una configurazione predeterminata data proprio dallo skeleton. Quest'obiettivo difficilmente è coniugato al raggiungimento delle prestazioni ideali, sebbene migliorino rispetto al caso sequenziale. Infatti si può dire che all'aumentare del livello di astrazione e di semplicità d'uso ci si allontana dalle prestazioni ideali, soprattutto in caso di applicazioni data parallel che operano su grandi quantità di dati e richiedono l'uso di strutture dati intermedie al fine del-

la distribuzione dei dati e del calcolo. Si può cercare di coniugare i due aspetti di astrazione e prestazioni limitando gli skeleton a strutture dati predeterminate e apportando ottimizzazioni basate su fusione di skeleton che evitino strutture dati intermedie - è il caso di skeTo - o introducendo skeleton specifici per un certo tipo di calcolo - è il caso della map-reduce di BlockLib. L'implementazione data flow - è il caso di Muskel - riesce a raggiungere prestazioni migliori (rispetto ad esempio a P^3L) riconducendo tutti gli skeleton ad una implementazione farm e su computazioni in cui c'è un forte riuso di dati (in skeleton data parallel) a causa della presenza delle cache locali sugli interpreti. Allo stesso modo si possono aumentare le prestazioni indirizzandosi ad architetture performanti quali i multiprocessori (come il CELL), in cui i costi di comunicazione non sono minimamente paragonabili a quelli che si hanno su cluster di PC, riuscendo ad avere elevate prestazioni data parallel anche su grana di calcolo medio-fine e basso grado di parallelismo. Anche in quest'ultimo caso per ottenere le prestazioni massime il livello del codice sequenziale che il programmatore deve scrivere si abbassa notevolmente, e tende a dover essere il quanto più dipendente all'architettura sottostante (è il caso di BlockLib su CELL). Un altro aspetto di valutazione di un sistema a skeleton riguarda le capacità espressive degli skeleton che offre, ovvero la molteplicità delle applicazioni che sono parallelizzabili attraverso quegli skeleton. Gli skeleton classici si adattano a molte applicazioni, ma chiaramente altre applicazioni hanno strutture diverse, sia regolari che non, che corrispondono a nuovi skeleton. Nel tentativo di generalizzare il concetto di skeleton - è il caso di Assist - al fine di poter esprimere strutture irregolari, si abbassa ancora una volta il livello del codice che deve essere scritto dal programmatore, rendendo di fatto quasi nullo il vantaggio di poter esprimere una certa applicazione come skeleton, avvicinandosi sempre di più al codice a livello dei processi del supporto che esegue effettivamente l'applicazione parallela. Inoltre a strutture irregolari è molto difficile associare un modello dei costi che ne descriva le prestazioni.

Nell'ottica di questa tesi un sistema a skeleton deve permettere di poter scrivere un programma parallelo scrivendo del codice il più simile possibile a quello che si scriverebbe per l'applicazione sequenziale funzionalmente equivalente. Di fondamentale importanza è la possibilità di riutilizzare il codice sequenziale preesistente

e migliorare allo stesso tempo le prestazioni rispetto al caso sequenziale, sebbene si resti lontani da quelle ideali. Il sistema deve nascondere al programmatore aspetti di basso livello come la specializzazione del codice rispetto ai processori reali su cui è eseguito, cosa che come abbiamo visto eSkel non fa. In questo senso il sistema oggetto della tesi è molto vicino all'approccio usato da MetaOCaml, in cui è il supporto a occuparsi della specializzazione del codice. La produttività del programmatore è essenziale: se per scrivere un'applicazione parallela tramite skeleton (riutilizzando codice esistente), al fine di raggiungere elevate prestazioni, l'utente deve scrivere codice lungo e complesso (come nel caso di eSkel, ASSIST, skeTo), può valere la pena scrivere direttamente il codice per programma parallelo senza avvalersi degli skeleton.

Capitolo 3

Gli skeleton in OcamlP3l

OcamlP3l [CLLP06] è una libreria di skeleton implementata in OCaml [Ler07], un linguaggio funzionale non puro. L'implementazione degli skeleton è basata su template ed è rivolta ad un'architettura cluster di PC.

Il linguaggio OCaml consente di avere una maggiore astrazione dovuta alla presenza di funzioni di ordine superiore, nel rispetto della definizione originaria in [Col91], ma anche la possibilità di utilizzare, nell'implementazione effettiva, i costrutti tipici dei linguaggi imperativi ed altre facilitazioni proprie del linguaggio, come la tipizzazione forte.

OcamlP3l consente la compilazione e la conseguente esecuzione di un programma parallelo in tre modalità diverse, corrispondenti a tre diverse semantiche:

- *semantica sequenziale*: l'esecuzione è sequenziale; permette un più facile debugging della logica del programma sequenziale;
- *semantica grafica*: mostra la computazione in termini di grafo i cui nodi sono i processori virtuali e gli archi i collegamenti tra di essi;
- *semantica parallela*: l'esecuzione è effettivamente parallela.

Nel seguito ci riferiremo alla semantica parallela.

Uno skeleton in *OcamlP3l* è ottenuto tramite una funzione; gli skeleton offerti dal sistema sono sia task parallel che data paralleli; li vedremo nei prossimi paragrafi facendo riferimento a [DDCL+07].

3.1 OcamlP3l: skeleton task parallel

3.1.1 Pipeline

In OcamlP3l lo skeleton pipeline definito nel paragrafo 1.2.1 è ottenuto tramite l'operatore infisso |||:

$$f_1 \; ||| \; f_2 \; ||| \; \cdots \; ||| \; f_n$$

dove f_i sono le funzioni calcolate dagli stadi della pipeline e possono a loro volta essere degli skeleton.

Per ottenere la pipeline dell'esempio del paragrafo 1.2.1 in OcamlP3l possiamo scrivere come in Code 1.

```
1 let f1 _ x = x +2;;
2 let f2 _ x = x * 6;;
3 let f3 _ x = x / 8;;
4 (* creazione della struttura del pipeline composta dai 3
    stadi *)
5 let mypipe = seq (f1) ||| seq (f2) ||| seq (f3);;
```

Code 1: Creazione di un pipeline di 3 stadi in OcamlP3l

Le prime 3 linee del codice Code 1 contengono la definizione delle tre funzioni che i tre stadi dovranno calcolare. Tutte queste funzioni hanno come primo parametro il tipo OCaml `unit` (simile al `void` in C), che, come vedremo dettagliatamente nella sezione 4.4, permette di per poter trattare l'inizializzazione di dati sia globale che locale.

La linea 5 contiene la creazione della struttura della pipeline: tramite l'operatore

`|||` si compongono due stadi consecutivi. La funzione `seq` , a sua volta uno skeleton, è un modo per ottenere un nodo di elaborazione a partire da una funzione. Anche quest'ultimo aspetto verrà ampiamente trattato in seguito.

3.1.2 Farm

In *OcamlP3l* lo skeleton farm definito nel paragrafo 1.2.2 è ottenuto tramite l'espressione

farm (f , n)

dove f è la funzione da calcolare, che può essere a sua volta uno skeleton e il parametro n indica il numero dei nodi worker.

Per ottenere lo skeleton farm per l'esempio del paragrafo 1.2.2 in *OcamlP3l* possiamo scrivere come in Code 2:

```
1 let f _ x = (x +2)*6/8;;
2 (* creazione della struttura del farm composto da 3
    worker *)
3 let myfarm = farm (seq (f),3);;
```

Code 2: Creazione di un farm di 3 worker in OcamlP3l

In linea 1 di Code 2 viene dichiarata la funzione `f` da calcolare e in linea 3 viene dichiarata la struttura del farm costituito da 3 worker che calcolano la funzione `f`.

3.2 OcamlP3l: skeleton data parallel

3.2.1 Mapvector

In *OcamlP3l* lo skeleton map definito nel paragrafo 1.3.1 è ottenuto con l'espressione

mapvector (f , n)

dove f è la funzione da calcolare per ogni elemento del vettore in ingresso, che può essere a sua volta uno skeleton, e n è il numero dei nodi worker.

Per ottenere lo skeleton map dell'esempio del paragrafo 1.3.1 in *OcamlP3l* possiamo scrivere come in Code 3.

```
1 let f _ x = (x +2)*6/8;;
2 (* creazione della struttura del map composto da 3 worker
    *)
3 let mymap = mapvector (seq (f),3);;
```

Code 3: Creazione di un map di 3 worker in OcamlP3l

Il codice Code 3 ha la stessa struttura del codice Code 2 , pertanto valgono le stesse considerazioni fatte nel paragrafo precedente.

L'implementazione effettiva in *OcamlP3l* 2.0 di questo skeleton è data in forma task parallel di farm, funzionalmente equivalente alla forma data parallel discussa in 1.3.2. L'emettitore decompone il vettore in ingresso in elementi che vanno a costituire lo stream di ingresso ai worker di un farm, che applicano `f` all'elemento ricevuto e inviano il risultato al collettore. Il collettore ha il ruolo di inserire nella giusta posizione del vettore di uscita il singolo risultato della computazione e, una volta completato tutto il vettore, di inviarlo sullo stream di uscita.

3.2.2 Reducevector

In *OcamlP3l* lo skeleton reduce definito nel paragrafo 1.3.2 è ottenuto tramite l'espressione

$$reducevector\ (f\ ,\ n\ )$$

dove f è la funzione binaria da applicare agli elementi del vettore di ingresso, che può essere a sua volta uno skeleton, e n è il numero dei nodi worker.

Per ottenere lo skeleton reduce dell'esempio del paragrafo 1.3.2 in *OcamlP3l* possiamo scrivere come in Code 4.

```
1 let f _ (x , y) = x + y;;
2 (* creazione della struttura del reduce composto da 4
     worker *)
3 let myreduce = reducevector (seq (f),4);;
```

Code 4: Creazione di un reduce di 4 worker in OcamlP3l

Il codice Code 4 restituisce una skeleton reduce che calcola la somma di tutti gli elementi di un vettore.

L'implementazione effettiva in *OcamlP3l* 2.0 di questo skeleton è data in forma task parallel di farm, funzionalmente equivalente alla forma data parallel discussa in 1.3.3. L'emettitore decompone il vettore di ingresso in coppie che vanno a costituire l'elemento dello stream di ingresso ai worker, che applicano la funzione f ai due elementi ricevuti e inviano i risultati parziali al collettore; il collettore provvede a reinviare ai worker coppie di risultati parziali finché non si determina la fine di questo ciclo, ovvero si percorrono tutti i livelli dell'albero logico discusso nel paragrafo 1.3.3 in Fig. 4.

3.3 OcamlP3l: skeleton di controllo

OcamlP3l fornisce alcuni skeleton di questo tipo (caratteristici del sistema), utili al coordinamento dell'esecuzione degli altri skeleton. Un esempio che abbiamo incontrato, senza approfondire, nei paragrafi precedenti è lo skeleton `seq`, che incapsula il codice sequenziale eseguito dai nodi di elaborazione degli worker degli skeleton paralleli. Altri esempi sono gli skeleton `parfun` e `pardo`.

A questa categoria appartiene anche lo skeleton generalmente denominato `loop`discusso in 1.4.1.

3.3.1 Loop

In *OcamlP3l* lo skeleton loop definito in 1.4.1 è ottenuto tramite l'espressione

loop (cond , f)

dove *cond* è la condizione di terminazione dell'iterazione e *f* rappresenta lo skeleton da iterare.

Supponiamo di voler rendere iterativo l'esempio in Code 1: per ottenere lo skeleton loop in *OcamlP3l* possiamo scrivere come in Code 5.

```
1 let f1 _ x = x +2;;
2 let f2 _ x = x * 6;;
3 let f3 _ x = x / 8;;
4 let mypipe = seq (f1) ||| seq (f2) ||| seq (f3);;
5 let mycond x = x > 400;; (* condizione di terminazione
     del loop *)
6 (* creazione dello skeleton loop *)
7 let myloop = loop ( mycond , mypipe);;
```

Code 5: Creazione di un loop di un pipeline di 3 stadi in OcamlP3l

Alla linea 5 di Code 5 viene dichiarata la funzione `mycond` che rappresenta la condizione di terminazione del loop. In linea 7 viene dichiarato il loop `myloop`: per ogni elemento dello stream il pipe itera il calcolo di *eq.1* finché il risultato non sarà maggiore di `400` e lo invierà sullo stream di uscita. Questo è un esempio di annidamento di skeleton.

3.3.2 Seq

Lo skeleton `seq`, visto negli esempi dei paragrafi 3.1.1 ,3.1.2 , 3.2.1 , 3.2.2 e 3.3.1 è ottenuto in *OcamlP3l* tramite l'espressione:

seq f

che trasforma una funzione sequenziale *f* in un nodo di elaborazione che applica *f* a tutti gli elementi dello stream di ingresso e invia il risultato sullo stream di uscita.

La funzione OCaml *f* deve avere questo tipo:

```
val f :  unit -> 'a -> 'b
```

e tramite l'applicazione della funzione `seq` otteniamo lo skeleton di base da passare come parametro agli altri skeleton. Con lo skeleton seq si rappresentano i nodi che computano la funzione effettiva, quindi i worker di farm, map e reduce e gli stadi del pipeline.

Lo skeleton `seq` rappresenta il "caso base" dell'annidamento di skeleton.

E' importante notare che in caso la funzione f necessiti di più di un parametro, come si vede in Code 4, questo deve essere passato come tupla (prodotto cartesiano) di valori . Infatti, in OCaml, una scrittura del tipo:

```
let f _ a b =  compute a b
```

ha tipo `unit -> 'a -> 'b -> 'c` che è un tipo diverso da quello richiesto `unit -> 'a -> 'b`.

L'uso delle tuple, che in OCaml sono un vero e proprio tipo base, permette uniformità di trattamento per tutte le funzioni a prescindere dal numero di parametri su cui operano.

3.3.3 Parfun

Lo skeleton parfun è ottenuto in *OcamlP3l* con l'espressione:

parfun (fun () → skeleton)

e restituisce, a partire da uno skeleton, eventualmente composto da più skeleton, una funzione di tipo:

```
'a stream -> 'b stream
```

che dà una rappresentazione dello skeleton come funzione che riceve uno stream in ingresso di tipo `'a` e restituisce uno stream di uscita di tipo `'b`.

`Parfun` ha lo scopo di far effettivamente instanziare la rete di processori virtuali che compongono lo skeleton solo una volta nell'entry point, `pardo` (paragrafo 3.3.4), dell'applicazione (e non in tutti i generici nodi di elaborazione), dove la funzione restituita da `parfun` verrà effettivamente applicata ad uno stream di ingresso.

Lo skeleton `parfun` viene usato in questo modo:

```
1 let parallel_calculation = parfun (fun () -> farm (seq(
    fun () a  -> a*a), 5)
```

Nell'esempio alla funzione `parfun` viene passato come argomento un farm di 5 worker che moltiplicano l'elemento dello stream per se stesso.

3.3.4 Pardo

Lo skeleton pardo è ottenuto in *OCamlP3l* tramite l'espressione :

pardo (fun () -> prog)

che definisce il programma principale (contenuto nell'espressione *prog*) dove gli skeleton *parfun*, precedentemente dichiarati, possono essere instanziati con gli stream di valori in ingresso e all'interno del quale non possono essere dichiarati nuovi *parfun.*

In Code 6 vediamo un tipico esempio di utilizzo di OcamlP3l: in linea 2 viene dichiarato il `parfun` che incapsula lo skeleton farm; in linea 3 viene chiamata la funzione `pardo` e nell'espressione passatale come parametro viene creato lo stream di interi (linea 4); in linea 5 si applica il risultato della funzione `parfun` allo stream di ingresso; in linea 6 si applica la funzione `print_res` (definita in linea 1) a tutti gli interi che compongono lo stream di uscita.

```
1 let print_res i = (print_int i; print_newline()) in
2 let parallel_calculation = parfun (fun () -> farm(seq(fun
      () a  -> a*a), 5) in
3 pardo(function ()->
4  let stream = P3lstream.of_list [2;1;1;3] in (* creazione
       di uno stream a partire dagli elementi di una lista
      *)
5  let r  = parallel_calculation stream in
6  P3lstream.iter print_res r;
7 );;
```

Code 6: creazione di un'applicazione parallela costituita da un farm di 5 worker che eleva al quadrato gli interi elementi dello stream e che stampa gli elementi dello stream di uscita

3.4 Annidamento di skeleton

Fin dall'introduzione abbiamo sottolineato l'importante caratteristica degli skeleton di poter essere annidati tra loro al fine di costituire un'applicazione parallela avente prestazioni migliori rispetto alla forma sequenziale e alla parallelizzazione tramite un unico skeleton.

Abbiamo visto, infatti, che tutti gli skeleton paralleli hanno come parametro un altro skeleton, che può essere lo skeleton seq o un altro skeleton parallelo. Ogni skeleton parallelo è dunque un albero di skeleton, la cui foglia è sempre uno skeleton seq.

E' possibile, ad esempio, che il generico worker di un farm necessiti di essere ulteriormente parallelizzato come una pipeline per ottenere prestazioni migliori. L'esempio della funzione F (*eq.1*), discussa nel paragrafo 1.2.1 e seguenti, si presta a chiarificare questo concetto.

Infatti il farm dell'esempio in Code 2 è costituito da worker che calcolano `f`; essendo questi worker, presi singolarmente dei moduli sequenziali e `f` una composizione di funzioni, possono essere parallelizzati con skeleton pipeline.

In *OcamlP3l* per ottenere l'applicazione parallela che utilizza un farm di pipeline possiamo scrivere come in Code 7.

```
1 let f1 _ x = x +2;;
2 let f2 _ x = x * 6;;
3 let f3 _ x = x / 8;;
4 (* creazione della struttura del pipeline composta dai 3
     stadi *)
5 let mypipe = seq (f1) ||| seq (f2) ||| seq (f3);;
6 (* creazione del farm con il pipeline annidato *)
```

```
7 let myfarm = farm (mypipe, 5);;
8 let parallel_calculation = parfun (fun () -> myfarm) in
9 pardo(function ()->
10  let stream = P3lstream.of_list [2;1;1;3] in (* creazione
       di uno stream a partire dagli elementi di una lista
     *)
11  let r  = parallel_calculation stream in
12  P3lstream.iter print_res r;
13 );;
```

Code 7: Creazione di un farm di 5 worker costituiti da un pipeline di 3 stadi in OcamlP3l

Alla linea 5 viene creato lo skeleton pipeline `mypipe`; alla linea 7 viene creato il farm `myfarm` che ha come parametro il pipeline precedentemente dichiarato; alla linea 8 allo skeleton `parfun` viene passato lo skeleton `myfarm` (farm di pipeline).

3.5 I colori

Le espressioni *OcamlP3l* che rappresentano gli skeleton paralleli che abbiamo visto nei paragrafi delle sezioni 2.1 e 2.2 hanno, oltre a quelli visti, dei parametri opzionali.

Ad esempio è possibile scrivere l'espressione che definisce uno skeleton farm composto da *n + 2* nodi di elaborazione:

$$farm\ \tilde{}col{:}c\ \tilde{}colv{:}[c_1....c_n]\ (seq\ f\ ,\ n)$$

dove la sintassi *OCaml*

$$\tilde{}lab{:}\ val$$

permette, tramite l'identificatore `lab` seguito da `:` , di dare un'etichetta "documentativa" ai parametri, e di indicare con il simbolo ˜ la loro opzionalità (i valori di default sono definiti nella dichiarazione della funzione).

Questi particolari parametri opzionali degli skeleton, *col* di tipo intero e *colv* di tipo lista di interi, sono detti *colori*.

I colori permettono di associare i nodi di elaborazione che compongono uno skeleton a un numero, eventualmente minore, di processori reali passati da linea di comando al momento del lancio del programma, potendone bilanciare esplicitamente il carico.

Il programma viene lanciato con un comando del tipo:

```
prog --p3lroot ip_1:port_1#col_1 ip_2:port_2#col_2 ...
   ip_i:port_i#col_i ...
```

dove `ip_i` e `port_i` (opzionale) sono l'indirizzo ip (o nome simbolico) e la porta di un processore reale, mentre `col_i` (opzionale) è il colore associato al processore reale. Nel caso in esame il farm è composto da emettitore e collettore con colore c e da n worker con colori $c_1 \ldots c_n$.

In generale m processori virtuali definiti dall'utente (uno per ogni processo effettivo) vengono associati con un semplice algoritmo, descritto in [DDCL+07], ai processori reali per i quali sia specificato da linea di comando un colore uguale o maggiore del proprio. Nel caso in cui i processori reali siano un numero minore di m su un processore reale vengono eseguiti più processi virtuali.

Capitolo 4

Ocamlp3l: implementazione

La semantica parallela di *OcamlP3l* è implementata in cinque file, o moduli Ocaml:

- `Parp3l`: contiene la definizione delle funzioni utente per ottenere gli skeleton (`|||`, `farm`, `mapvector`, `reducevector`, `loop`), le strutture dati per rappresentarli e le funzioni che le elaborano;
- `Nodecode`: contiene gli skeleton di controllo che configurano la rete dei processori reali che compongono lo skeleton parallelo (`parfun`, `pardo`);
- `Template`: contiene i template implementativi (funzioni) dei processori reali;
- `Server`: contiene il codice per la creazione dei processi che vanno in esecuzione sui processori reali;
- `Commlib`: contiene l'implementazione dei canali di comunicazione e delle primitive send e receive tramite socket unix, offrendone un'astrazione ad alto livello ai moduli `Nodecode` , `Template` e `Server`.

4.1 I p3ltree

Ogni skeleton accetta uno stream di elementi in input e restituisce uno stream di elementi in output.

Un esempio di utilizzo di skeleton è questo (esempio 1):

```
1 farm (seq (fun () a -> a*a), 2) ||| farm (seq (fun () a
    -> a/2), 3)
```

ovvero un pipeline di due stadi ognuno dei quali è costituito da un farm, il primo costituito da 2 worker che computano il quadrato dell'elemento dello stream in input, il secondo costituito da 3 worker che dividono per due l'elemento dello stream in input.

`seq` rappresenta la funzione da calcolare su ogni componente dello stream, o del singolo elemento dell'array componente dello stream in caso di data parallel, deve quindi essere sempre presente nell'espressione che rappresenta lo skeleton.

Da questo esempio è evidente come un'importante caratteristica degli skeleton sia la possibilità di essere annidati tra loro. Ogni skeleton (ad eccezione di `seq`), può contenerne un altro, e deve necessariamente contenerne almeno uno, e cioè la foglia `seq`. Gli skeleton hanno quindi una natura ricorsiva, e l'espressione che li codifica è pertanto rappresentabile da una struttura dati ad albero.

I tipi di queste funzioni sono cosi' dichiarati nel moduli `Parp3l`:

```
val seq : ?col:int -> (('a, 'b) io -> 'a -> 'b) ->('a, 'b
   ) p3ltree

val farm :
  ?col:int -> ?colv:int list -> ('a, 'b) p3ltree * int ->
      ('a, 'b) p3ltree

val mapvector :
  ?col:int -> ?colv:int list -> ('a, 'b) p3ltree * int ->
      ('a array, 'b array) p3ltree

val reducevector :
  ?col:int ->  ?colv:int list -> ('a * 'a, 'a) p3ltree *
    int -> ('a array, 'a) p3ltree
```

```
val ( ||| ) : ('a, 'b) p3ltree -> ('b, 'c) p3ltree -> ('a
    , 'c) p3ltree
```

dove:

```
type ('a, 'b) io = unit
```

si può vedere che ognuna di queste funzioni restituisce il tipo `p3ltree`, e che tutte, ad eccezione di `seq` (che è una foglia), prendono in ingresso, tra le altre cose, un altro `p3ltree`.

Naturalmente il tipo `p3ltree` è un tipo ricorsivo:

```
type ('a, 'b) p3ltree =
  | Farm of int * int list * (('a, 'b) p3ltree * int)
  | Pipe of ('a, 'b) p3ltree list
  | Mapvector of (int * int list * (('a, 'b) p3ltree *
    int))
  | Reduce of (int * int list * (('a, 'b) p3ltree * int))
  | Seq of (int * (('a, 'b) io -> 'a -> 'b))
```

parametrico nei tipi di ingresso (`'a`) e di uscita (`'b`) della funzione sequenziale da parallelizzare contenuta nella foglia `Seq`, che sono anche i tipi dello stream in ingresso e di uscita.

Vediamo a titolo d'esempio i codici delle funzioni `farm` e `seq` che restituiscono i `p3ltree` `Seq` e `Farm`.

```
1 let seq ?(col=0) (f: (('a,'b) io -> ('a ->'b))) = Seq (
    col, f);;
2
3 let farm ?(col=0) ?(colv=[])
4 ((tree : ('a , 'b, 'c, 'd, 'e, 'f) p3ltree), ncopy) =
    Farm (col, colv, (tree, ncopy));;
```

Le chiamate di funzione dell'esempio 1 restituiscono il `p3ltree` in Exp. 1.

```
- :  (int, int) Parp3l.p3ltree =


Pipe [Farm (0, [], (Seq (0, <fun>), 2)); Farm (0, [], (Seq (0, <fun>),
3))]
```
Exp. 1: Il valore restituito dall'applicazione dell'esempio 1

In Fig. 10 è mostrata una rappresentazione grafica di Exp. 1.

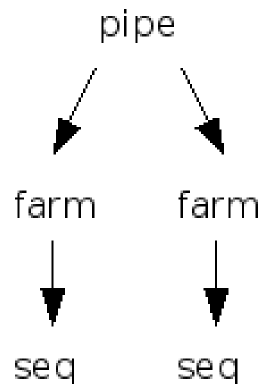

Fig. 10: L'espressione Exp. 1 è un albero : la pipe ha due figli farm, ogni figlio farm ha un figlio seq

Dal punto di vista dell'utilizzo, questi skeleton, vanno passati ad una funzione, *parfun*, la quale esprime la dipendenza tra tipi sopra detta, in questo modo

```
val parfun :  (unit -> ('a, 'b) Parp3l.p3ltree) -> 'a
   P3lstream.t ->'b P3lstream.t
```

come si vede, questa funzione prende in ingresso un `p3ltree`, e restituisce una funzione, che dato uno stream in ingresso, dello stesso tipo del primo parametro di `p3ltree`, restituisce uno stream di uscita, dello stesso tipo del secondo parametro di `p3ltree`.

Le relazioni di tipo create dalla dichiarazione della funzione `parfun` consentono di avere un controllo, a tempo di compilazione, di corrispondenza tra i tipi dello stream in ingresso e in uscita e la funzione sequenziale che calcola l'output a partire dall'input, e tutte le funzioni che sono poi applicate all'output.

Il risultato della funzione `parfun` va poi utilizzato nell'espressione passata in ingresso alla funzione `pardo`:

```
val pardo : (unit -> 'a) -> unit
```

al cui interno non è possibile "dichiarare" altri `parfun`, ma questi possono e devono solo essere applicati ad uno stream in ingresso.

Il pattern d'uso è il seguente:

```
1 let print_res i = (print_int i; print_newline()) in
2 let parallel_calculation = parfun (fun () -> farm (seq(
    fun () a  -> a*a), 5) in
3 pardo(function ()->
4 let stream = P3lstream.of_list [2;1;1;3] in
5  let r  = parallel_calculation stream in
6  P3lstream.iter print_res r;
7 );;
```

Code 8: Creazione di un'applicazione parallela costituita da un farm di 5 worker che eleva al quadrato gli interi elementi dello stream e che stampa gli elementi dello stream di uscita

4.2 Il modello d'esecuzione

L'esecuzione inizia con l'applicazione della funzione `pardo`.

Prima di chiarire esattamente come, per ora stabiliamo che ogni `parfun` corrisponde ad una rete di processori virtuali che rappresentano i nodi del grafo – skeleton, connessi tra loro da dei canali di comunicazione.

L'esempio 1, corrisponde a dieci nodi: il primo stadio del pipeline è composto da quattro nodi (due worker, un emettitore e un collettore); il secondo stadio del pipeline è composto da cinque nodi (tre worker, un emettitore e un collettore); a questi si aggiunge un nodo radice.

Il nodo radice, sempre presente, è responsabile della messa in opera degli altri nodi, che sono quelli che eseguono effettivamente la computazione parallela. Si occupa di inviare ad ogni nodo informazioni circa i suoi canali di input e output e circa il tipo di computazione che deve eseguire: ad esempio se è un emettitore, un collettore, o un worker. La radice crea poi un processo che invia al primo nodo della rete i dati dello stream di ingresso, e attende la ricezione dello stream dei risultati dall'ultimo nodo della rete.

Viceversa, i nodi non radice, attendono le informazioni di inizializzazione dalla radice ed in seguito eseguono il codice opportuno contenuto nel modulo `Template`.

Dal punto di vista operativo l'utente, una volta compilato il programma OcamlP3l tramite `ocamlp3lc` o `ocamlp3lopt` con opzione `-par`, su ogni nodo "esecutivo" dovrà mandare in esecuzione il programma, senza alcun parametro (obbligatorio). Mentre provvederà a mandare in esecuzione il programma sul nodo scelto come radice con l'opzione `--p3lroot` seguita dall'elenco dei nomi simbolici o indirizzi ip dei nodi su cui è stata fatta partire la computazione in precedenza. L'entry point del sistema (`pardo`) è contenuto nel modulo `Nodecode` nel quale è presente un codice del tipo mostrato in Code 9.

```
1 let procpool=ref [] in
2 let collect_processors proc = procpool:=proc::(!procpool)
    in
3 if !isroot then (* se presente opzione p3lroot isroot =
   true *)
4  (* codice eseguito solo nella root : costruzione della
    rete di processori virtuali, mapping tra processori
    virtuali e reali, invio informazioni di
    configurazione della rete ai processori reali, invio
    stream di input e ricezione stream di output *) ...
5 else node() (* codice eseguito sugli altri nodi *)
```

Code 9: Frammento di codice della funzione pardo

Il frammento di codice in Code 9 rende evidente la biforcazione che avviene a tempo d'esecuzione, a seconda della presenza o meno dell'opzione `p3lroot`, tra il codice eseguito nel nodo radice e quello eseguito sui nodi esecutori.

4.3 Dai p3ltree ai template

Ogni espressione (di tipo `unit -> p3ltree`) passata ad una `parfun` viene applicata e aggiunta, insieme al canale di input ed output, ad una lista di `parfun`; questa lista viene poi elaborata nel nodo root ai fini della costruzione di una struttura dati atta all'esecuzione dei passi di inizializzazione sopra citati.

Vediamo, per chiarezza, la dichiarazione del tipo `parfun` e della lista `parfuns`:

```
type parfun = Commlib.vp_chan option ref  *  Commlib.
   vp_chan option ref  * (unit -> (unit, unit) p3ltree)


val parfuns : parfun list ref
```

In particolare ogni `p3ltree`, viene trasformato in una `p3ltreeexp` :

```
type ('a, 'b) p3ltreeexp = Innode of ('a, 'b) innode
  | Leaf of ('a, 'b) node
and ('a, 'b) innode =
  | Farmexp of ('a, 'b) node * ('a, 'b) node * ('a, 'b)
     p3ltreeexp list * config option
  | Mapvectorexp of ('a, 'b) node * ('a, 'b) node * ('a,
     'b) p3ltreeexp list * config option
  | Reduceexp of ('a, 'b) node * ('a, 'b) node * ('a, 'b)
      p3ltreeexp list * config option
  | Pipexp of ('a, 'b) p3ltreeexp list * config option

and ('a, 'b) node = proctemplate * config option * ('a, '
   b) action option * color
```

Notiamo che anche una `p3ltreeexp` è una struttura ricorsiva, ma, a differenza del `p3ltree`, rende esplicita la topologia dello skeleton rappresentato dal `p3ltree` : stavolta una `p3ltreeexp` è un grafo, non un albero. L' `innode` delimita una macro area del grafo e rappresenta un certo skeleton, mentre il `node` è il singolo nodo del grafo che esegue una determinata funzionalità.

La rappresentazione dell'esempio 1 in termini di `p3ltreeexp` (prescindendo da alcuni dettagli) è data in Exp. 2.

```
- : (int, int) Parp3l.p3ltreeexp =
 Innode
 (Pipexp
 ([
 Innode
  (Farmexp ((Farmemit, None, None, 0), (Farmcoll 10, None
     , None, 0),
   [
   Leaf (Userfun, None, Some (Seqfun <fun>), 0);
   Leaf (Userfun, None, Some (Seqfun <fun>), 0);
   ],
   None));
 Innode
  (Farmexp ((Farmemit, None, None, 0), (Farmcoll 5, None,
     None, 0),
   [
   Leaf (Userfun, None, Some (Seqfun <fun>), 0);
   Leaf (Userfun, None, Some (Seqfun <fun>), 0);
   Leaf (Userfun, None, Some (Seqfun <fun>), 0);
   ],
   None))
 ], None))
```

Exp. 2: La p3ltreeexp corrispondente all'Exp. 1

Graficamente notiamo ancora meglio la differenza tra un `p3ltree` (Fig. 10) e una `p3ltreeexp` (la figura Fig. 11 è stata ottenuta usando lo strumento GrafP3l).

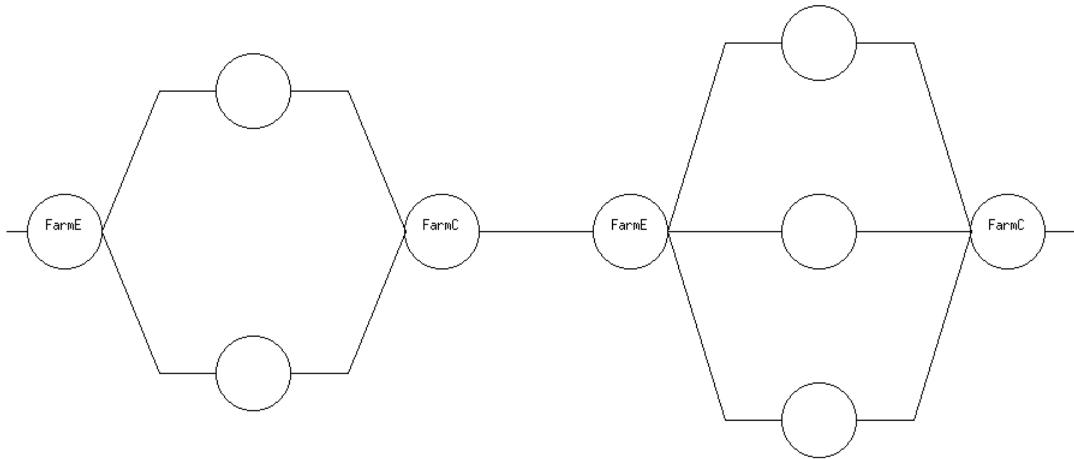

Fig. 11: Grafo dei processori virtuali che rappresenta la Exp. 2, ottenuto tramite GrafP3l

I nodi del grafo rappresentano i processori virtuali e gli archi i canali che li connettono; la radice (non visibile in figura) è collegata al primo nodo FarmE in uscita e al secondo nodo FarmC in ingresso.

Torniamo al tipo `node` e vediamo che tra i suoi componenti vi è il tipo `proctemplate`. Questo tipo specializza ciascun nodo con la computazione che deve effettivamente eseguire, cioè se si tratta di un nodo emettitore, collettore o worker (esecutore della funzione sequenziale) dei diversi tipi di skeleton. Il `proctemplate`, come accennato, è uno dei dati che vengono inviati dalla radice ai nodi della rete durante la fase di inizializzazione. Assieme al `proctemplate` il `node` memorizza le altre informazioni che la root deve inviare ai processori virtuali.

Riportiamo alcuni `proctemplate` che incontreremo riguardanti l'esempio 1:

```
type proctemplate =
  | Userfun
  | DemandUserfun
  | Farmemit
```

```
| IntDemandFarmemit
| ExtDemandFarmemit
| Farmcoll of int
```

Per ogni `p3ltree` della lista la radice applica la funzione `p3ldo` mostrata in Code 10.

```
val p3ldo :  cin -> cout -> ('a, 'b) p3ltree -> nproc *
   ('a, 'b) p3ltreeexp list * nproc
```

```
1 let p3ldo =
2  fun cin cout f ->
3   let bound =
4    demandadjust
5    (bindchan [cin] [cout] (nodeconf (leafalloc (expand 0
        f)))) in
6   let fstnode, lastnode = getchans bound in
7   (fstnode, (leafs_of bound), lastnode);;
```

Code 10: La funzione p3ldo

(dove i tipi `cin` e `cout` sono alias per interi, rappresentano rispettivamente i canali / processori di ingresso e di uscita).

La prima ed unica trasformazione da `p3ltree` a `p3ltreeexp` è effettuata dalla funzione:

```
val expand : color -> ('a, 'b) p3ltree -> ('a, 'b)
   p3ltreeexp
```

di cui riportiamo in Code 11 un frammento esplicativo per il caso di `farm` .

```
1 let rec expand col =
2  let color = get_color col in
3  function
4   | Seq (c, f) -> Leaf (Userfun, None, Some (Seqfun f),
      color c)
```

```
5      .
6      .
7      | Farm (c, cv, (t, n)) ->
8       let tmpcol = color c in
9       let tmpcolist = nvlist n tmpcol cv in
10      Innode
11      (Farmexp
12      ((Farmemit, None, None, tmpcol),
13      (Farmcoll n, None, None, tmpcol),
14      List.map (fun x -> expand x t) tmpcolist ,
15    None))
```

Code 11: La funzione expand

Dal codice in Code 11 cui si può notare che la funzione è applicata ricorsivamente ai nodi dell'albero `p3ltree` . L'espansione avviene creando, nel caso di un nodo `Farm`, un `innode Farmexp` che contiene il nodo emettitore (`proctemplate Farmemit`), il nodo collettore (`proctemplate Farmcoll`) e tanti nodi quanti sono i worker (tale numero è definito dal parametro n). La funzione `expand` è applicata ricorsivamente a tanti elementi quanti sono i worker tramite la funzione OCaml `List.map`, ed il risultato è una lista di `p3ltreeexp` che viene memorizzata nell'`innode`. Ogni worker, chiaramente, può essere a sua volta un `innode` (la proprietà dell'annidamento dei `p3ltree` è "sfruttata" e mantenuta nella creazione del grafo). La ricorsione termina sulle foglie, cioè sui `p3ltree Seq`, quando viene creata una `Leaf` che ha come `proctemplate Userfun`.

La Exp. 2 rappresenta esattamente il risultato dell'applicazione di `expand` ad Exp. 1.

Dopo l'applicazione della funzione `expand` alcuni componenti degli `innode` e dei `node` non sono ancora inizializzati (`None`), per questo tale risultato passa attraverso una serie di funzioni che apportano successivi raffinamenti alla `p3ltreeexp`.

A questo proposito riportiamo la dichiarazione dei tipi utilizzati in `node` e `innode` che non abbiamo ancora citato, a cui ci riferiremo nel corso della spiegazione:

```
type nproc = int
type config =
  | Seqconf of nproc * cin list option * cout list option
  | Inconf of cin option * cout option
```

contiene informazioni di configurazione di `node` e `innode`, precisamente:

- `Seqconf` è usato nei `node` e rappresenta l'identificatore del processore virtuale, la lista dei canali di ingresso e la lista dei canali di uscita;
- `Inconf` è usato negli `innode` e rappresenta il canale di ingresso e il canale di uscita dell'`innode`: in generale tali canali sono quelli dei processori virtuali che eseguono funzionalità di servizio (emettitore e collettore)

```
type ('a, 'b) action =
  | Seqfun of (('a, 'b) io -> ('a -> 'b))
  | OutChanSel of (Commlib.vp_chan list -> Commlib.
     vp_chan)
  | InChanSel of (Commlib.vp_chan list -> 'a * Commlib.
     vp_chan)
```

determina una funzionalità svolta dai `node` :

- `Seqfun` è usato nei nodi worker e contiene la funzione sequenziale che deve effettivamente essere calcolata
- `OutChanSel` e `InChanSel` sono usati nei nodi di servizio e determinano rispettivamente le strategie di scelta dei canali di output e input su cui inviare o ricevere i messaggi

Subito dopo `expand` viene applicata la funzione `leafalloc` :

```
val leafalloc : ('a, 'b) p3ltreeexp -> ('a, 'b)
   p3ltreeexp
```

di cui vediamo la struttura in Code 12.

```
1 let rec leafalloc = function
2  | Leaf n -> Leaf (allocnode n)
3    .
4    .
5  | Innode (Farmexp(n1, n2, tl, c)) ->
6     Innode (Farmexp (allocnode n1, allocnode n2, List.map
         leafalloc tl, c))
```

Code 12: La funzione leafalloc

La funzione `allocnode` :

```
val allocnode : 'a * 'b * 'c * 'd -> 'a * config option *
    'c * 'd
```

è così definita:

```
1 let allocnode (templ, _, a, b) = (templ, genconfig (), a,
    b)
```

mentre `genconfig` :

```
val genconfig : unit -> config option
```

è mostrata in Code 13.

```
1 let genconfig =
2  let count = ref 0 in
3  (fun () ->
4   let c = Some (Seqconf (!count, None, None)) in
5   count := !count + 1;
6   c);;
```

Code 13: La funzione genconfig

A questo punto ad ogni processore virtuale è assegnato un identificatore univoco, ovvero viene inizializzata una parte della componente `config` dei `node`. Nel caso del nostro esempio otteniamo la `p3ltreeexp` in Exp. 3.

```
- : (int, int) Parp3l.p3ltreeexp =
Innode
 (Pipexp
  ([Innode
   (Farmexp ((Farmemit, Some (Seqconf (12, None, None)),
      None, 0),
   (Farmcoll 2, Some (Seqconf (11, None, None)), None, 0)
      ,
   [Leaf
   (Userfun, Some (Seqconf (9, None, None)), Some (Seqfun
      <fun>), 0);
    Leaf
    (Userfun, Some (Seqconf (10, None, None)), Some (
      Seqfun <fun>), 0)],
   None));
   Innode
   (Farmexp ((Farmemit, Some (Seqconf (17, None, None)),
      None, 0),
   (Farmcoll 3, Some (Seqconf (16, None, None)), None, 0)
      ,
   [Leaf
   (Userfun, Some (Seqconf (13, None, None)), Some (
      Seqfun <fun>), 0);
    Leaf
    (Userfun, Some (Seqconf (14, None, None)), Some (
      Seqfun <fun>), 0);
    Leaf
    (Userfun, Some (Seqconf (15, None, None)), Some (
```

```
      Seqfun <fun>), 0)],
   None))],
  None))
```

Exp. 3: Risultato dell'applicazione della funzione leafalloc all'Exp. 2

Continuando nella catena delle applicazioni incontriamo `nodeconf` :

```
val nodeconf : ('a, 'b) p3ltreeexp -> ('a, 'b) p3ltreeexp
```

che vediamo in Code 14.

```
1 let rec nodeconf = function
2   | Farmexp (n1, n2, tl, c)) ->
3     let cin, cout = pe_of_node n1, pe_of_node n2 in
4      Innode
5      (Farmexp (n1, n2, List.map nodeconf tl,
6      Some (Inconf (Some cin, Some cout))))
7     .
8     .
9   | x -> x
```

Code 14: La funzione nodeconf

La funzione `nodeconf` restituisce una `p3ltreeexp` in cui viene inizializzata completamente la componente `config` degli `innode`, la vediamo in Exp. 4.

```
- : (int, int) Parp3l.p3ltreeexp =
Innode
(Pipexp
 ([Innode
   (Farmexp ((Farmemit, Some (Seqconf (3, None, None)),
      None, 0),
   (Farmcoll 2, Some (Seqconf (2, None, None)), None, 0),
   [Leaf
```

```
     (Userfun, Some (Seqconf (0, None, None)), Some (
        Seqfun <fun>), 0);
     Leaf
     (Userfun, Some (Seqconf (1, None, None)), Some (
        Seqfun <fun>), 0)],
     Some (Inconf (Some 3, Some 2))));
    Innode
   (Farmexp ((Farmemit, Some (Seqconf (8, None, None)),
     None, 0),
   (Farmcoll 3, Some (Seqconf (7, None, None)), None, 0),
   [Leaf
    (Userfun, Some (Seqconf (4, None, None)), Some (
       Seqfun <fun>), 0);
     Leaf
    (Userfun, Some (Seqconf (5, None, None)), Some (
       Seqfun <fun>), 0);
     Leaf
    (Userfun, Some (Seqconf (6, None, None)), Some (
       Seqfun <fun>), 0)],
    Some (Inconf (Some 8, Some 7))))],
    Some (Inconf (Some 3, Some 7))))
```

Exp. 4: Risultato dell'applicazione della funzione nodeconf all'Exp. 3

Il `config` dell'`innode` (`Inconf`) indica i canali/processori di input e di output allo skeleton, in questo caso rispettivamente il canale/processore emettitore e il canale/processore collettore.

Gli ultimi componenti `config` dei node sono inizializzati dalla funzione `bindchan`, che sostanzialmente esplicita per ogni nodo quali sono i suoi canali/processori di input e di output: stabilisce gli archi di ingresso e di uscita di ciascun nodo del grafo.

La funzione bindchan:

```
val bindchan :  cin list -> cout list -> ('a, 'b)
   p3ltreeexp -> ('a, 'b) p3ltreeexp
```

è mostrata in Code 15.

```
1 let rec bindchan cinl coutl = function
2  | Leaf n -> Leaf (chconnect cinl coutl n)
3   .
4   .
5  | Innode (Farmexp (n1, n2, tl, (Some (Inconf (Some fin,
      Some fout)) as c))) ->
6     let winl, woutl = List.split (List.map getchans tl) in
7      Innode(Farmexp (chconnect cinl winl n1, (* connette l
         'emettitore ai worker in uscita e allo skeleton
         padre in ingresso *)
8      chconnect woutl coutl n2,(* connette il collettore ai
          worker in ingresso e allo skeleton padre in
         uscita *)
9      List.map (bindchan [fin] [fout]) tl, c)) (* connette
         ogni worker all'emettitore in ingresso e al
         collettore in uscita *)
```

Code 15: La funzione bindchan

Riportiamo per chiarezza porzioni del codice delle funzioni chconnect (Code 16), pe_of_node (Code 17) e getchans (Code 18).

```
val chconnect : cin list ->  cout list -> 'a * config
   option * 'b * 'c -> 'a * config option * 'b * 'c
```

```
1 let chconnect cinl coutl = function
2  | templ, Some (Seqconf (nproc, _, _)), a, b ->
```

```
3     (templ, Some (Seqconf (nproc, Some cinl, Some coutl)),
        a, b)
4   | _ -> failwith "Parp3l: unknown structure in chconnect"
      ;;
```

Code 16: La funzione chconnect

```
val pe_of_node : 'a * config option * 'b * 'c -> nproc
```

```
1 let pe_of_node = function
2   | _, Some (Seqconf (nproc, _, _)), _, _ -> nproc
3   | _ -> failwith "Parp3l: unknown structure in pe_of_node
      ";;
```

Code 17: La funzione pe_ of_ node

```
val getchans : ('a, 'b) p3ltreeexp -> nproc * nproc
```

```
1 let rec getchans = function
2   | Leaf n ->
3     let nproc= pe_of_node n in (nproc, nproc)
4     .
5     .
6   | Innode (Farmexp (n1, n2, _, _)) ->
7     (pe_of_node n1, pe_of_node n2)
8   | _ -> failwith "Parp3l: unknown structure in getchans"
      ;;
```

Code 18: La funzione getchans

Il nostro esempio diventa quindi la `p3ltreeexp` mostrata in Exp. 5.

```
- : (int, int) Parp3l.p3ltreeexp =
Innode
(Pipexp
  ([Innode
  (Farmexp((Farmemit, Some (Seqconf (3, Some [-1], Some
     [0; 1])), None, 0),
  (Farmcoll 2, Some (Seqconf (2, Some [0; 1], Some [8])),
     None, 0),
  [Leaf
   (Userfun, Some (Seqconf (0, Some [3], Some [2])),
   Some (Seqfun <fun>), 0);
   Leaf
   (Userfun, Some (Seqconf (1, Some [3], Some [2])),
   Some (Seqfun <fun>), 0)],
   Some (Inconf (Some 3, Some 2))));
  Innode
  (Farmexp ((Farmemit, Some (Seqconf (8, Some [2], Some
     [4; 5; 6])), None, 0),
  (Farmcoll 3, Some (Seqconf (7, Some [4; 5; 6], Some
     [-1])), None, 0),
  [Leaf
   (Userfun, Some (Seqconf (4, Some [8], Some [7])),
   Some (Seqfun <fun>), 0);
   Leaf
   (Userfun, Some (Seqconf (5, Some [8], Some [7])),
   Some (Seqfun <fun>), 0);
   Leaf
   (Userfun, Some (Seqconf (6, Some [8], Some [7])),
   Some (Seqfun <fun>), 0)],
```

```
  Some (Inconf (Some 8, Some 7))))],
 Some (Inconf (Some 3, Some 7))))
```

Exp. 5: Risultato dell'applicazione della funzione bindchan all'Exp. 4

La funzione `demandadjust` :

```
val demandadjust : ('a, 'b) p3ltreeexp -> ('a, 'b)
   p3ltreeexp
```

differisce da quelle fin qui presentate in quanto agisce sulle componenti `proctemplate` dei `node`: nel caso in cui si abbia a che fare con un farm (non interno ad un altro farm) l'emettitore e i nodi connessi dovranno implementare la strategia di assegnamento/richiesta dei task on demand (bilanciamento del carico). Si devono quindi modificare i `proctemplate` già impostati durante la `expand`. Come già detto, i `proctemplate` determinano il codice effettivo che sarà eseguito su ciascun nodo esecutivo. In Exp. 6 è mostrato il risultato della funzione `demandadjust` relativo al nostro esempio.

```
- : (int, int) Parp3l.p3ltreeexp =
Innode
 (Pipexp
  ([Innode
  (Farmexp
  ((ExtDemandFarmemit, Some (Seqconf (3, Some [-1], Some
     [0; 1])),
   None, 0),
  (Farmcoll 2, Some (Seqconf (2, Some [0; 1], Some [8])),
     None, 0),
   [Leaf
   (DemandUserfun, Some (Seqconf (0, Some [3], Some [2]))
     ,
   Some (Seqfun <fun>), 0);
    Leaf
```

```
(DemandUserfun, Some (Seqconf (1, Some [3], Some [2]))
    ,
  Some (Seqfun <fun>), 0)],
  Some (Inconf (Some 3, Some 2))));
 Innode
(Farmexp
((ExtDemandFarmemit, Some (Seqconf (8, Some [2], Some
   [4; 5; 6])),
None, 0),
(Farmcoll 3, Some (Seqconf (7, Some [4; 5; 6], Some
   [-1])), None, 0),
 [Leaf
 (DemandUserfun, Some (Seqconf (4, Some [8], Some [7]))
    ,
 Some (Seqfun <fun>), 0);
  Leaf
 (DemandUserfun, Some (Seqconf (5, Some [8], Some [7]))
    ,
  Some (Seqfun <fun>), 0);
  Leaf
 (DemandUserfun, Some (Seqconf (6, Some [8], Some [7]))
    ,
  Some (Seqfun <fun>), 0)],
 Some (Inconf (Some 8, Some 7))))],
Some (Inconf (Some 3, Some 7))))
```

Exp. 6: Risultato dell'applicazione della funzione demandadjust all'Exp. 5

Notiamo che, nel nostro caso, al posto dei `proctemplate Farmemit` e `Userfun` sono stati inseriti i `proctemplate ExtDemandFarmemit` e `DemandUserfun` che appunto determinano l'implementazione da parte dei `node` emettitore e worker della strategia

di distribuzione dei task on demand.

La `p3ltreeexp` non è però ancora completa: rimangono da inizializzare le componenti `action` dei `node`. Quest'ultimo passaggio è svolto dalla funzione `leafs_of`:

```
val leafs_of : ('a, 'b) p3ltreeexp -> ('a, 'b) p3ltreeexp
    list
```

che è mostrata in Code 19.

```
1 let rec leafs_of = function
2  | Leaf n -> [Leaf n]
3    .
4    .
5  | Innode (Farmexp (n1, n2, tl, _)) ->
6     Leaf (confemit n1) :: Leaf (confcollect n2) :: List.
         flatten  (List.map leafs_of tl)
7  | _ -> failwith "Parp3l: unknown structure in leafs_of"
     ;;
```

Code 19: La funzione leafs_of

Le funzioni `confemit` e `confcollect` sono riportate in Code 20 e Code 21.

```
val confemit : 'a * 'b * 'c * 'd -> 'a * 'b * ('e, 'f)
   action option * 'd
```

```
1 let confemit (pt, conf, _, col) =
2  (pt, conf, Some (OutChanSel (rr ())), col);;
```

Code 20: La funzione confemit

```
val confcollect : 'a * 'b * 'c * 'd -> 'a * 'b * ('e, 'f)
   action option * 'd
```

```
1 let confcollect (pt, conf, _, col) =
2   (pt, conf, Some (InChanSel Commlib.receive_any), col);;
```

Code 21: La funzione confcollect

Semplicemente, nel caso del farm, l'emettitore (non on demand) sceglie i worker a cui inviare i task secondo una strategia round robin (implementata dalla funzione `rr()`), mentre il collettore sceglie i canali da cui ricevere in modo non deterministico (funzione `receive_any` del modulo `Commlib`).

Finalmente il nostro esempio diventa la `p3ltreeexp` in Exp. 7.

```
- : (int, int) Parp3l.p3ltreeexp list =
[Leaf
 (ExtDemandFarmemit, Some (Seqconf (3, Some [-1], Some
    [0; 1])),
 Some (OutChanSel <fun>), 0);
 Leaf
 (Farmcoll 2, Some (Seqconf (2, Some [0; 1], Some [8])),
 Some (InChanSel <fun>), 0);
 Leaf
 (DemandUserfun, Some (Seqconf (0, Some [3], Some [2])),
 Some (Seqfun <fun>), 0);
 Leaf
  (DemandUserfun, Some (Seqconf (1, Some [3], Some [2])),
   Some (Seqfun <fun>), 0);
 Leaf
  (ExtDemandFarmemit, Some (Seqconf (8, Some [2], Some
     [4; 5; 6])),
   Some (OutChanSel <fun>), 0);
 Leaf
  (Farmcoll 3, Some (Seqconf (7, Some [4; 5; 6], Some
     [-1])),
   Some (InChanSel <fun>), 0);
 Leaf
```

```
  (DemandUserfun, Some (Seqconf (4, Some [8], Some [7])),
   Some (Seqfun <fun>), 0);
 Leaf
  (DemandUserfun, Some (Seqconf (5, Some [8], Some [7])),
   Some (Seqfun <fun>), 0);
 Leaf
  (DemandUserfun, Some (Seqconf (6, Some [8], Some [7])),
   Some (Seqfun <fun>), 0)]
```

Exp. 7: Risultato dell'applicazione della funzione leafs_of all'Exp. 6

Adesso la radice ha costruito la rappresentazione completa di tutti i processori virtuali delle reti `parfun` dichiarate dall'utente, e può mapparli sui processori reali passati come parametro dell'opzione ***p3lroot*** (eventualmente usando il parametro `color` ai fini di bilanciamento del carico). Fatto ciò può iniziare a inviare ad ogni processore reale le informazioni relative ai processori virtuali che sono stati mappati su di esso.

A questo punto possiamo esplicitare, in Code 22 , un frammento di codice della funzione `node` che abbiamo visto richiamare in Code 9, che è quello che viene eseguito su ogni processore reale.

```
val node: unit -> unit
```

```
1 let node () =
2   establish_smart_server nodeconfiguration;;
```

Code 22: La funzione node

la funzione `establish_smart_server` semplicemente permette ad ogni processore reale di creare un processo/thread per ogni processore virtuale mappato su di esso, mentre `nodeconfiguration` è più interessante, la vediamo in Code 23.

```
val nodeconfiguration : Commlib.vp_chan -> unit
```

```
1  let nodeconfiguration ch =
2   .
3   .
4    let (n,inl,outl,templ,f) = receive ch in
5      <configurazione canali di input e output tramite
         scambio di altri messaggi con la root>
6    .
7    .
8   match (templ,f) with
9     (Userfun,Seqfun f) ->
10   seqtempl f (setup_receive_chan myvpdata) (List.hd coutl
        )
11  | (DemandUserfun,Seqfun f) ->
12      demandseqtempl f (setup_receive_chan myvpdata) (List
          .hd coutl)
13
14  | (Farmemit,OutChanSel f)  ->
15      farmetempl f (setup_receive_chan myvpdata) coutl
16
17  | (ExtDemandFarmemit,OutChanSel f) ->
18     extdemandfarmetempl f (setup_receive_chan myvpdata)
         coutl
19   .
20   .
21   .
22  | _ -> failwith "Nodecode: illegal template or template/
       function combination found in nodeconfiguration"
```

Code 23: La funzione nodeconfiguration

Ecco che ancora abbiamo una differenziazione del codice eseguito a seconda del nodo

su cui si esegue; la prima l'abbiamo vista tra nodo radice e nodo non radice. Adesso ogni nodo riceve dalla radice le informazioni che lo caratterizzano (quelle contenute nel tipo `node`): il suo id, il `proctemplate` , la lista dei canali di input, la lista dei canali di output, e la sua `action`. Su ciascun nodo, dopo una fase di configurazione dei canali, a seconda del `proctemplate` (e della `action`) che riceve, viene eseguita una funzione diversa, quella che caratterizza il nodo, e la computazione parallela ha finalmente inizio.

4.4 I template

Le funzioni eseguite dai diversi nodi sono contenute nel modulo `Template`; per illustrare questo modulo prendiamo sempre ad esempio il caso del farm, con emettitore round robin. La scrittura da parte dell'utente di un programma parallelo come quello in Code 8 (supponendo che non sia eseguito il passo di `demandadjust`) porta alla creazione di sette processori virtuali: uno esegue il template corrispondente a `Farmemit`, cinque eseguono il template corrispondente a `Userfun`, l'altro il template corrispondente a `Farmcoll`. I nodi, nell'ambito dell'esecuzione della computazione parallela cooperano a scambio di messaggi, rappresentati dal tipo:

```
type ('a) packet =
  | UserPacket of 'a * sequencenum * tag list
  | EndStream
  | EndOneStream
```

`UserPacket` rappresenta il messaggio che contiene i dati su cui operare, gli altri due sono messaggi utilizzati per implementare il protocollo di terminazione.

```
type sequencenum = int
```

è il numero di sequenza che identifica ogni elemento dello stream di ingresso a ciascuna rete *parfun*

```
type tag = Seqtag | Farmtag | ...
```

informazioni accessorie da inviare assieme al dato principale, utili, tra le altre cose, al controllo di correttezza dell'esecuzione.

La radice inizia inviando lo stream dei dati sotto forma di `UserPacket`, determinando i `sequencenum`, uno diverso per ogni elemento dello stream, e con `tag Seqtag`. Una volta esaurito lo stream verso una rete `parfun`, invia a questa il messaggio `EndOneStream`. Esaurite le computazioni invia a tutte le reti `parfun` il messaggio `EndStream`, che induce la terminazione di tutti i processi coinvolti.

Come si vede in Code 23, i template coinvolti nel caso del Code 8, nell'ordine, sono:

```
val farmetempl : (Commlib.vp_chan list -> Commlib.vp_chan
   ) -> Commlib.vp_chan -> Commlib.vp_chan list -> unit
```

prende in ingresso la funzione di scelta del canale di output, il canale di input, in questo caso corrispondente direttamente al canale di uscita della radice e la lista dei canali di uscita verso i worker (corrispondenti ai canali di input dei `seqtempl`).

```
val seqtempl :  (unit -> 'a -> 'b) -> Commlib.vp_chan ->
   Commlib.vp_chan -> unit
```

prende in ingresso la funzione dal calcolare, il canale di input e il canale di output, che nel caso in esame corrispondono rispettivamente al canale di output dell'emettitore e al canale di input del collettore.

```
val farmctempl : 'a -> Commlib.vp_chan list -> Commlib.
   vp_chan -> unit
```

prende in ingresso la lista dei canali di input dai worker (quelli in uscita dai `seqtempl`) e il canale di output, in questo caso quello in ingresso alla radice.

Vediamo la struttura di questi template di esempio (analoga ad altri template dello stesso tipo) rispettivamente in Code 24 , Code 25 e Code 26.

```
1 let farmetempl f ic ocl =
2  let notfinished = ref true in
3  while !notfinished do
4   let theoc = f ocl in
```

```
5    match receive ic with
6     | UserPacket(p, seqn, tl) ->
7       cl_send theoc (UserPacket (p, seqn, Farmtag :: tl));
8     | EndStream  ->
9       List.iter (fun x -> cl_send x EndStream) ocl;
10      List.iter vp_close ocl; vp_close ic;
11      notfinished := false
12    | EndOneStream  ->
13      List.iter (fun x -> cl_send x EndOneStream) ocl;
14  done;;
```

Code 24: Il template che implementa l'emettitore del farm

Il template `farmetempl`(Code 24) riceve ciclicamente un messaggio e lo inoltra ogni volta al worker successivo determinato dalla funzione `f`, aggiungendo il tag `Farmtag` in testa alla `tag list` dell'`Userpacket`.

```
1  let seqtempl f ic oc =
2   let f = instanciate_io f ic oc in
3   let notfinished = ref true in
4   while !notfinished do
5    match receive ic with
6     | UserPacket(p, seqn, tl) ->
7       cl_send oc (UserPacket (f p, seqn, tl));
8     | EndStream  ->
9       cl_send oc EndStream;
10      vp_close oc; vp_close ic;
11      notfinished := false
12    | EndOneStream  ->
13      cl_send oc EndOneStream;
14  done;;
```

Code 25: Il template che implementa il generico worker

Il template `seqtempl` (Code 25) riceve ciclicamente il messaggio dal canale di input, vi applica la funzione e lo invia sul canale di uscita (collettore).

```
1  let farmctempl f icl oc =
2   let notfinished = ref true in
3   while !notfinished do
4    let inpkt, ic = receive_any !okicl in
5    match inpkt with
6    | UserPacket (p, seqn, Farmtag :: r) ->
7      <codice per ordinamento dei pacchetti in uscita>
8      cl_send oc (UserPacket(p, seqn, r));
9    | EndStream  ->
10     vp_close ic;
11     <codice per la memorizzazione del numero di EndStream
          ricevuti>
12     if <ricevuto EndStream da tutti i worker> then begin
13      cl_send oc EndStream;
14      vp_close oc;
15      notfinished := false
16     end
17   | EndOneStream  ->
18     <codice per la memorizzazione del numero di
         EndOneStream ricevuti>
19     if <ricevuto EndOneStream da tutti i worker> then
         cl_send oc EndOneStream;
20   | _ ->
21     printf "INTERNAL ERROR: farmctempl got a non-farm
         packet!!!";
22     print_newline ();
23  done;;
```

Code 26: Il template che implementa il collettore del farm

Il template `farmctempl` (Code 26) riceve ciclicamente i messaggi dai `seqtempl` e li invia sul canale di output privi del `tag` di testa `Farmtag` ed eventualmente tenendo conto del `sequencenum`.

Da queste ultime tre porzioni di codice notiamo il funzionamento del protocollo di terminazione. Ogni template cicla su una variabile booleana inizialmente `true`. Quando l'emettitore riceve un messaggio `EndStream` lo inoltra su tutti i canali in uscita. Ogni worker che riceve tale messaggio lo inoltra sul canale di uscita (collettore). Il collettore quando riceve un `EndStream` da un canale di ingresso memorizza l'avvenuta ricezione e una volta ricevuto questo messaggio da tutti i canali in ingresso lo inoltra sul canale di uscita. Tutti i template, dopo aver ricevuto e inoltrato il messaggio, assegnano il valore `false` alla variabile `notfinished` in guardia al ciclo `while`, permettendo la loro terminazione normale. Lo stesso principio di inoltro si applica per il messaggio `EndOfOneStream` (però questo messaggio non provoca la terminazione del ciclo). La radice si aspetta di ricevere (in questo caso dal collettore) degli `UserPacket`, poi degli `EndOfOneStream`, e solo dopo degli `EndStream`. Solo la ricezione dei messaggi in quest'ordine provoca la terminazione normale anche della radice, viceversa termina con un fallimento.

Il codice della radice relativo alla sua terminazione è contenuto nel modulo `Nodecode`, nelle funzioni `parfun` e `pardo`.

Tutti i template del modulo `Template` hanno un funzionamento analogo con le varianti riguardanti il funzionamento on demand e il funzionamento data parallel. In caso di funzionamento on demand la variante di `seqtempl` inizia ogni iterazione del ciclo principale inviando un messaggio di disponibilità sul canale di comunicazione col processo a monte (la variante on demand dell'emettitore per esempio). Questo, per simmetria, inizia ogni iterazione ricevendo non deterministicamente il messaggio di disponibilità da un canale, tra quelli dei processi a valle, sul quale invia il task da eseguire (`Userpacket`). L'emettitore su domanda deve inviare i messaggi per la terminazione a tutti i worker solo dopo che questi gli hanno inviato il messaggio di

disponibilità.

4.5 Inserimento di nuovi skeleton

Da quanto visto nelle sezioni precedenti risulta implicita la procedura che il programmatore (della libreria) deve seguire in caso di inserimento di un nuovo skeleton. I passi da seguire sono nell'ordine:

- modificare il modulo `Parp3l`
  - aggiungere al tipo `p3ltree` il nuovo costruttore che rappresenta il nuovo skeleton
  - definire la funzione utente che restituisce il nuovo `p3ltree`;
  - aggiungere al tipo `proctemplate` i costruttori che rappresentano i template che devono essere eseguiti dai nodi che compongono il nuovo skeleton;
  - eventualmente definire una nuova `action` da associare ai `node` della nuova `p3ltreeexp`;
  - aggiungere al tipo `innode` (contenuto nella `p3ltreeexp`) il costruttore che rappresenta il grafo del nuovo skeleton;
  - aggiungere il match case relativo al nuovo `p3ltree` nella funzione `expand` in modo da trasformare il nuovo `p3ltree` nella nuova `p3ltreeexp`;
  - per ognuna delle funzioni della catena di chiamate contenute in `p3ldo` successive a `expand` aggiungere un match case relativo alla nuova `p3ltreeexp` che apporti le opportune modifiche alla `p3ltreeexp`;
- modificare il modulo `Template`
  - definire le funzioni che rappresentano i nuovi template introdotti in `proctemplate`
  - eventualmente aggiungere nuovi costruttori al tipo `tag`;

– modificare il modulo `Nodecode`

  – aggiungere il match case relativo alla nuova coppia (`proctemplate`, `action`) nella funzione `nodeconfiguration` che richiami la funzione definita al punto precedente.

Nel caso in cui si voglia semplicemente modificare l'implementazione di uno skeleton già esistente (senza modificarne la topologia) il programmatore deve solo agire sulla funzione che rappresenta il template dello skeleton.

Se invece si vuole modificare anche la topologia o altri aspetti di uno skeleton esistente si deve agire sui match case delle funzioni che manipolano la `p3ltreeexp` che rappresenta lo skeleton per gli aspetti che interessano (es. `bindchan` per la topologia).

4.6 Il ruolo delle lambda-astrazioni

Fin qui abbiamo incontrato alcune funzioni che prendono in ingresso un'altra funzione. In alcuni di questi casi la funzione in ingresso ha come primo argomento il tipo *unit* (ovvero l'applicazione senza parametri).

I primi due casi sono:

```
val parfun :
 (unit -> ('a, 'b) Parp3l.p3ltree) -> 'a P3lstream.t -> '
    b P3lstream.t


val pardo : (unit -> 'a) -> unit
```

Ricordiamo che:

- la funzione `parfun` prende in ingresso la funzione in ingresso la funzione che, applicata, restituisce il p3ltree che rappresenta lo skeleton

- la funzione `pardo` prende in ingresso la funzione che, applicata, restituisce l'espressione che rappresenta la computazione parallela. In particolare, come si vede in Code 8, contiene il l'applicazione dello stream di ingresso alle espressioni `parfun` precedentemente create tramite la funzione `parfun`.

L'ulteriore astrazione rispetto all'argomento `unit` fa sì che:

- nel caso di `parfun`, il valore che contiene l'albero della computazione, che poi viene usato per la costruzione della `p3ltreeexp` necessaria ad inizializzare la rete dei processori virtuali, possa essere ottenuto, tramite l'applicazione della funzione, solo nel nodo radice. - nel caso di `pardo`, il codice vero e proprio che dà inizio all'esecuzione del programma parallelo possa essere eseguito soltanto nel nodo root. Solo nel nodo root viene applicata la funzione in ingresso al `pardo` e si ottiene come risultato l'espressione `'a`, e in ultima analisi l'esecuzione del codice principale del programma parallelo.

L'altro caso è:

```
val seq : ?col:int -> (('a, 'b) io -> 'a -> 'b) -> ('a, '
   b) p3ltree
```

dove il tipo `('a, 'b) io` è un alias per il tipo `unit`. Come si vede sia in esempio 1 che in Code 8 la funzione sequenziale passata a `seq` non è una semplice funzione `'a -> 'b` , ma una funzione `unit -> 'a -> 'b`.

Come spiegato in [CLLP06] l'ulteriore argomento `unit` in questo caso permette di raggiungere un importante risultato e cioè la differenziazione tra un'inizializzazione globale e un'inizializzazione locale dei dati usati dalla funzione sequenziale.

```
1 let f =
2  let a = initialize() in
3   fun () -> fun x -> calc a;;
4  ...
5  let sequential = seq f
```

Code 27: Worker con dati inizializzati globalmente e replicati

Infatti l'applicazione del primo argomento unit avviene solo nei nodi worker (come naturalmente anche l'applicazione agli altri argomenti della funzione vera e propria da calcolare), quindi una scrittura del tipo in Code 27 provoca l'inizializzazione del dato `a` una volta per tutte e la sua replicazione in tutti i nodi worker (quando il nodo worker applica l'argomento `unit` alla funzione `f` il dato a è gia stato inizializzato, prima dell'invio della chiusura al nodo).

Viceversa una scrittura del tipo in Code 28 provoca l'inizializzazione del dato `a` localmente ad ogni nodo worker (il dato `a` viene inizializzato dopo l'applicazione della funzione `f` all'argomento `unit` da parte del worker).

```
1  let f =
2   fun () ->
3     let a = initialize() in fun x -> calc a;;
4  ...
5  let sequential = seq f
```

Code 28: Worker con dati inizializzati localmente

Capitolo 5

Modifica degli skeleton data parallel esistenti

Nella sezione 4.5 abbiamo brevemente discusso le modalità da seguire per inserire o modificare skeleton esistenti. In questo capitolo vedremo in particolare cosa è stato fatto per modificare in senso data parallel l'implementazione degli skeleton `mapvector` (paragrafo 3.2.1) e `reducevector` (paragrafo 3.2.2), precedentemente implementati in modo stream parallel. Per lo skeleton `mapvector` è bastato modificare il template implementativo dell'emettitore e del collettore introdurre un nuovo `proctemplate` per il worker; mentre per lo skeleton `reducevector`, modificato nella topologia, è stato necessario agire anche sulla funzione `bindchan`.

Nelle prossime due sezioni vedremo il dettaglio delle modifiche apportate nei due skeleton.

5 .1 Mapvector

Come abbiamo visto nel paragrafo 3.2.1 nella versione precedente dello skeleton `mapvector` la sua implementazione effettiva era task parallel. In particolare l'emettitore, ricevuto l'array dello stream di ingresso, generava a partire da questo uno stream costituito dagli elementi dell'array e li inviava per l'elaborazione ai worker,

implementati con lo stesso template di `farm` e `pipeline` (`seqtempl`). Il collettore si occupava di creare l'array di uscita con gli elementi, ricevuti dai worker, nella giusta posizione.

In Code 29 vediamo un frammento di codice del vecchio emettitore; alle linee 10-16 vediamo la generazione del sotto stream di elementi dell'array ricevuto.

```
1  let mapvectoretempl f ic ocl =
2   let notfinished = ref true in
3   while !notfinished do
4   try
5   (* ricezione dell'array in ingresso *)
6     match receive ic with
7     | UserPacket(p, seqn, tl) ->
8      (* generazione del sottostream e distribuzione ai
         worker*)
9       let l = Array.length p in
10      for i = 0 to l - 1 do
11      (* prossimo worker f e' la funzione rr() che
          implementa la strategia round robin *)
12      let theoc = f ocl in
13       (* invio task *)
14      cl_send theoc
15      (UserPacket (p.(i), seqn, Mapvectortag (i, l) :: tl)
          )
16      done;
17   | EndStream  ->
18       ...
19   | EndOneStream  ->
20       ...
21  with
22  | End_of_file ->
```

```
23          ...
24   done;;
```

Code 29: Il template che implementava l'emettitore del mapvector

Al fine di dare un'implementazione data parallel allo skeleton sono stati semplicemente modificati i template dell'emettitore, del worker e del collettore. In particolare l'emettitore a partire dall'array ricevuto dallo stream di ingresso genera delle partizioni di tale array e le invia ai worker.

In Code 30 vediamo un frammento dell'implementazione del nuovo emettitore `mapvector`.

```
1  let mapvectoretempl f ic ocl =
2   let notfinished = ref true in
3   let nw = List.length ocl in
4   while !notfinished do
5   try
6   match receive ic with
7   | UserPacket(p, seqn, tl) ->
8    let datalen = Array.length p in
9    (* la funzione scatter restituisce un iteratore di
        partizioni di p *)
10   let pack = scatter p nw in
11   let tag = ref 0 in
12   for i=0 to (nw - 1) do
13   begin
14    let theoc = f ocl in
15    let partition = pack () in (* la prossima partizione*)
16    let dim = Array.length partition in
17    cl_send theoc
18    (UserPacket (partition, seqn, Mapvectortag(!tag ,
         datalen) :: tl));
```

```
19      tag:= !tag + dim; (* tag per la gestione dell'array di
            output nel collettore*)
20     end;
21     done
22   | EndStream  ->
23        ...
24   | EndOneStream  ->
25        ...
26 with
27   | End_of_file ->
28        ...
29 done
30 ;;
```

Code 30: Il nuovo template per l'emettitore del mapvector

Alle linee 12-20 di Code 30 vediamo che l'emettitore invia una partizione dell'array `p` restituita dall'iteratore `pack` (linea 15) ottenuto dalla funzione `scatter` (linea 10); invia nel `Mapvectortag` la lunghezza dell'array globale (`datalen`) e l'indice di inizio della partizione (`tag`) al fine di permettere al collettore una corretta ricomposizione dell'array di uscita.

Il worker data parallel è una variante del `seqtempl`,`mapvetorseqtempl`, che riceve dall'emettitore una porzione `p` dell'array elemento dello stream inviatagli dall'emettitore e invia nell'`Userpacket` al collettore l' array dei risultati calcolato tramite la funzione OCaml `Array.map f p` (anziché ricevere il singolo elemento `a` dell'array e calcolare `f a` come accadeva in `seqtempl`).

Il nuovo collettore provvede a ricostruire l'array di output a partire dalle partizioni ricevute dai worker mediante informazioni contenute nell'apposito `tag Maptag.`

Il tutto funziona modificando la funzione `mapvector` da quella originaria che vediamo in Code 31

```
1 let mapvector
```

```
2   ?(col=0) ?(colv=[]) ((tree : ('a, 'b) p3ltree), ncopy)
      =
3   (Obj.magic (Mapvector (col, colv, (tree, ncopy))) : ('a
       array, 'b array) p3ltree);;
```

Code 31: La vecchia funzione mapvector

a quella che vediamo in Code 32.

```
1 let mapvector
2   ?(col=0) ?(colv=[]) ((tree : ('a , 'b, 'c, 'd, 'e, 'f)
      p3ltree), ncopy) =
3   match tree with Seq(f) -> (Obj.magic (Mapvector (col,
      colv, (Mapvectorseq (f), ncopy))): ('a array, 'b
      array) p3ltree)
4   | _-> (Obj.magic (Mapvector (col, colv, (Streamervector
      (tree),ncopy))): ('a array, 'b array) p3ltree)
5 ;;
```

Code 32: La nuova funzione mapvector

Vediamo che in questa nuova implementazione, dal momento che l'utente usa comunque la funzione `seq` per inserire il codice sequenziale, è necessario introdurre un nuovo `p3ltree` associato dalla funzione `expand` ad un nuovo `proctemplate` (`Mapvectorworker`). Se `mapvector` contiene il solo `p3ltree Seq` questo viene sostituito col nuovo `p3ltree Mapvectorseq`; se invece siamo di fronte ad un annidamento più complesso (ovvero ogni worker è costituito da uno skeleton task parallel) è necessario introdurre un altro `p3ltree` (`Streamervector`).

`Streamervector` è associato ad un template che, similmente al vecchio emettitore, genera uno stream di elementi dell'array di ingresso e lo inoltra al primo nodo dello skeleton task parallel annidato e che poi ricompone i risultati ottenuti in un array da inviare al collettore.

Essendo stati introdotti dei nuovi `proctemplate` è necessario aggiungere alla funzione `nodeconfiguration` i match case ad essi relativi; è necessario introdurre i nuovi match case nelle altre funzioni della catena di chiamate in `p3ldo` con codice analogo a quello degli altri match case.

5.2 Reducevector

Come abbiamo visto nel paragrafo 3.2.2 lo skeleton `reducevector` implementa l'albero logico che vediamo in Fig. 4 a pagina 8 in modo task parallel. Volendo implementare lo skeleton `reducevector` in modo data parallel si potrebbe pensare ad un template isomorfo all'albero di Fig. 4 ; utilizzando questo template i processori virtuali sono $2n-1$: le n foglie più gli $n-1$ nodi intermedi. Per risparmiare processori virtuali è possibile implementare l'albero logico con un cubo di $lg_2\ n$ dimensioni utilizzando solo le n foglie corrispondenti ai worker, come descritto in [Van09]. La computazione a cubo consta di $log_2\ n$ iterazioni in cui le funzioni dei nodi intermedi dell'albero binario sono svolte ad ogni iterazione ad un sottoinsieme dei nodi worker. In Fig. 12 vediamo come mappare un albero binario con 4 foglie (a destra) su un cubo di 2 dimensioni (a sinistra): nell'esempio le iterazioni sono due, alla prima iterazione i worker che detengono x_1 e x_3 inviano il loro contenuto rispettivamente ai nodi che detengono x_2 e x_4 , che calcolano i risultati parziali calcolati nell'albero dai nodi intermedi di primo livello; alla seconda iterazione il nodo che detiene x_2 invia il risultato parziale calcolato all'iterazione precedente al nodo che detiene x_4 , che calcola il risultato finale usando il valore appena ricevuto e quello calcolato da esso stesso all'iterazione precedente.

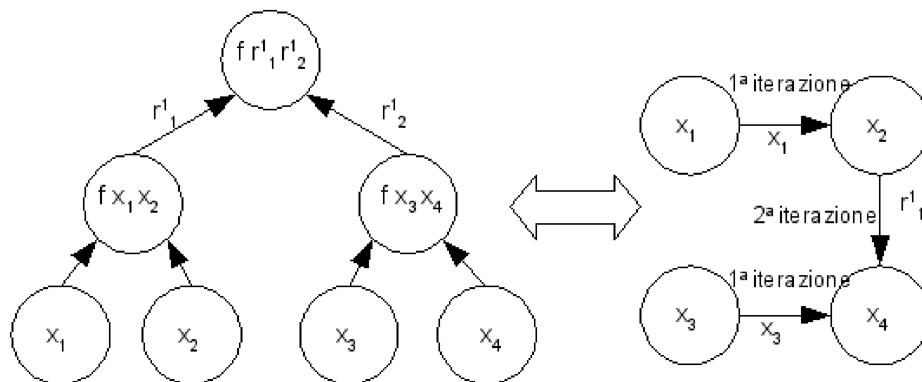

Fig. 12: Un albero binario con 4 foglie e 7 nodi totali mappato su in cubo di 2 dimensioni e 4 nodi totali

Utilizzando il formalismo LC già introdotto diamo lo pseudocodice del generico processore virtuale worker.

```
1  int A[M]; channel [N][N] worker; channel out out_chan;
2  worker [j] :: {
3  channel out worker[j][*]; channel in worker[*][j]
4  int partial_res := A[j];
5  int temp;
6  for (i = 0; i< log2M; i++) {
7   temp := partial_res;
8   if (j%2^i=i) then send(worker[j][j+2^(i-1)] ,
       partial_res);
9   if (j%2^i = 0) then {
10   receive (worker[j -2^(i-1)][j], partial_res);
11   partial_res:= f (partial_res , temp);
12   }
13  }
14 if ( j = M) then send (out_chan, partial_res); /*l'ultimo
      worker detiene il risultato finale alla fine delle
     iterazioni */
```

```
15 }
```

Ad ognuna delle $log_2\, n$ iterazioni il worker deve:

- decidere se è uno dei worker che devono inviare il proprio risultato parziale e inviarlo al giusto worker (linea 8);
- decidere se è uno dei worker che devono ricevere un risultato parziale e riceverlo dal giusto worker (linee 9-11)
- in caso di ricezione calcolare il nuovo risultato parziale applicando la funzione al nuovo valore ricevuto e a quello precedentemente calcolato (linea 11).

Alla fine delle iterazioni l'ultimo worker detiene il risultato finale della reduce e può inviarlo sul canale di uscita (linea 14).

Nel caso realistico in cui i worker detengano una partizione dell'array di ingresso anziché un singolo elemento ogni worker procede ad una prima fase di calcolo della *fold* sulla partizione locale e in seguito procede alla seconda fase iterativa sopra descritta.

Vediamo che questa nuova implementazione, rispetto a quella task parallel precedentemente implementata dal sistema prevede anche degli archi tra i nodi worker, ovvero dei canali di comunicazione tra di essi.

Ovviamente è necessaria una modifica delle funzioni che implementano emettitore, collettore e soprattutto worker, secondo le stesse modalità seguite per lo skeleton `mapvector`, e che quindi non mostreremo. E' necessario inoltre introdurre una modifica alla funzione `bindchan` (Code 15 a pag 77) rispetto al comportamento standard per gli altri skeleton, al fine di collegare i worker tra di loro. In Code 33 vediamo tale modifica.

```
1 let rec bindchan cinl coutl = function
2   ....
3   | Innode (Reduceexp (n1, n2, tl, (Some (Inconf (Some fin
      , Some fout)) as c))) ->
```

```
4    let winl, woutl = List.split (List.map getchans tl) in
5     Innode (Reduceexp
6     (chconnect cinl winl n1, chconnect woutl coutl n2,
7     (bindworkers tl fin fout), c))
```

Code 33: Modifica della funzione bindchan

La funzione `bindworkers` (linea 7) collega opportunamente i worker tra di loro usando a sua volta la funzione `bindchan`.

Le altre funzioni della catena di chiamate in `p3ldo` vanno modificate solo inserendo un match case che si comporta in modo analogo agli altri.

Capitolo 6

Map: un nuovo skeleton data parallel

Oggetto principale della tesi è stato l'inserimento di un nuovo skeleton che permettesse map e stencil fissi anche iterativi con condizione di terminazione globale.

Si è introdotto un costrutto simile alla map P^{3}L visto in 2.3.1 che la estende permettendo di esprimere anche le condizioni di iterazione. Nel far questo si sono seguite le linee guida contenute in [Li03], il cui fondamento teorico, consistente in un modello di calcolo per la distribuzione matrici dense a processori di elaborazione, si trova in [DCP03] (esteso successivamente a matrici multidimensionali e provvisto di un calcolo per la ricostruzione delle matrici in [DCLP07]).

In particolare in [Li03] viene illustrato un algoritmo per la distribuzione dei dati e vengono definiti alcuni tipi per le strutture dati che traducono in *OcamlP3l* la sintassi di P^{3}L; nel seguito vedremo i dettagli. Il lavoro di tesi è stato volto all'implementazione effettiva di quanto delineato in [Li03] integrando il nuovo skeleton nel sistema esistente.

In questo capitolo vedremo inizialmente come il nuovo skeleton è presentato all'utente, introducendo la sintassi, saranno illustrati in seguito l'implementazione e alcuni esempi di applicazioni note.

6.1 Il punto di vista dell'utente

Abbiamo visto nel capitolo 4, senza farvi esplicito riferimento, che in una libreria funzionale, quale è *OcamlP3l*, l'illusione di avere a che fare con la sintassi di un linguaggio, piuttosto che con delle chiamate di funzioni, è data implicitamente dalla sintassi delle funzioni di ordine superiore che non richiedono virgole o parentesi tra i parametri (se non facciamo uso di tuple).

Le parole chiave di P^3L sono sostituite da costruttori di tipo; il controllo di correttezza sintattica che in P^3L è svolto dal front-end del compilatore è svolto in *OCamlP3l* parte tramite type checking (a tempo di compilazione), parte tramite controlli a tempo di esecuzione.

6.1.1 Distribuzione dei dati in P^3L

Prima di procedere nella trattazione è bene dare una definizione informale alle diverse strategie di distribuzione dei dati sui processori virtuali a cui ci riferiremo in seguito. Le strategie principali sono tre:

- *scatter*: ogni elemento dell'array di ingresso è distribuito al corrispondente processore virtuale;
- *multicast*: ogni elemento (o insieme di elementi) dell'array di ingresso è distribuito ad un sottoinsieme dei processori virtuali;
- *broadcast*: ogni elemento (o insieme di elementi) dell'array di ingresso è distribuito a tutti i processori virtuali.

Vediamo adesso come le tre strategie sono esprimibili in P^3L; riprendiamo dal paragrafo 2.3.1 la sintassi P^3L relativa allo skeleton map.

```
1 map name in (input-array-decl) out (output-array-decl)
2 nworker x,y,...
3  inner-skel
4 end map
```

Riprendiamo anche la sintassi da usare nei parametri di ingresso a inner-skel per indicare la distribuzione dei dati ai processori virtuali.

Ad esempio abbiamo visto che

in `(a[*i][], b[][*j])` **out** `(c[*i][*j]) (distr.1)`

indica che per ogni elemento `(i,j)` dell'array di uscita `c`, il corrispondente processore virtuale `(i,j)`, riceve in ingresso l'i-esima riga di `a` e la j-esima riga di `b`; ciò significa che la i-esima riga è ricevuta in ingresso da `j` processori virtuali, e che la j-esima colonna è ricevuta in ingresso da `i` processori virtuali, ovvero ci troviamo di fronte a due esempi di multicast.

Per esprimere una broadcast possiamo scrivere

in `(a[])` **out** `(c[*i][*j]) (distr.2)`

con cui si indica che per ogni elemento `(i,j)` dell'array di uscita `c`, il corrispondente processore virtuale `(i,j)` riceve in ingresso l'intero array `a`; ovvero `a` è inviato in ingresso a tutti i processori virtuali.

Per esprimere una scatter possiamo scrivere

in `(a[*i][*j])` **out** `(c[*i][*j]) (distr.3)`

con cui si indica che per ogni elemento `(i,j)` dell'array di uscita `c`, il corrispondente processore virtuale `(i,j)` riceve in ingresso il corrispondente elemento `(i,j)` di `a`.

P^3L offre anche la possibilità di esprimere stencil fissi sempre tramite la sintassi degli indici, ad esempio con

in `(a[*i-a...*i+b][*j])` **out** `(c[*i][*j]) (distr.4)`

si indica che per ogni elemento `(i,j)` dell'array di uscita `c` il corrispondente processore virtuale `(i,j)` riceve in ingresso $(a+b+1)$ elementi della j-esima colonna, dall'elemento `i-a` all'elemento `i+b` inclusi.

L'array di output deve avere lo stesso numero di dimensioni espresso nella clausola `nworker` e tutti gli indici usati per gli array di input devono comparire in ognuno degli array di output.

6.1.2 Distribuzione dei dati in OcamlP3l

La sintassi degli indici vista nel paragrafo precedente è rappresentata in *OcamlP3l* dal tipo

```
type indexpattern =
  | All
  | Ranf of int * (int * int)
  | One of int
```

usato dai tipi

```
type body_in_list = BodyIn of indexpattern list list

type  body_out_list = BodyOut of  indexpattern list list
```

In questo caso le parole chiave `in` e `out` sono sostituite dai costruttori di tipo `BodyIn` e `BoyOut`, la sintassi degli indici per ogni parametro (array) è sostituita da una lista di `indexpattern`, e la lista di tutti i parametri è sostituita da una lista di liste di `indexpattern`.

Vediamo come usare questi tipi per riscrivere le espressioni di distribuzione dei dati del paragrafo 6.1.1:

```
(distr.  1) BodyIn [[One(1);All], [All;One(2)]] BodyOut[[One(1);One(2)]]

(distr.  2) BodyIn [[All]] BodyOut [[One(1);One(2)]]

(distr.  3) BodyIn [[One(1);One(2)]] BodyOut [[One(1);One(2)]]

(distr.  4) BodyIn [[Ranf(1,(-a,b));One(2)]] BodyOut [[One(1);One(2)]]
```

Il costruttore `One(i)` sostituisce il pattern P^3L `[*i]`, il costruttore `All` sostituisce il pattern P^3L `[]`, il costruttore `Ranf(i,-a,b)` sostituisce il pattern P^3L `[*i-a...*i+b]`; la corrispondenza tra gli indici è espressa da identificatori interi corrispondenti.

Quindi una scatter è espressa da una lista di `indexpattern` del solo tipo `One(i)`; una multicast da una lista di `indexpattern` che ne contiene almeno uno del tipo `All`; una broadcast da una lista di `indexpattern` del solo tipo `All`.

6.1.3 Bigarray

Il nostro obiettivo è trattare matrici multidimensionali; il sistema dei tipi OCaml ci limita nel nostro obiettivo in quanto per ogni dimensione di matrice esiste un tipo diverso, ovvero un `'a array` è un tipo diverso da un `'a array array`, e così via. Una qualsiasi soluzione generica al trattamento di matrici multidimensionali non compilerebbe.

Per questo motivo si è scelto di utilizzare come strutture dati su cui operare gli array definiti nella libreria Ocaml *Bigarray*[Ler07], che rappresenta array numerici, sia reali che complessi, da 1 a 16 dimensioni come un unico tipo chiamato *Genarray*.

I Bigarray possono avere il layout di memoria di array C o Fortran, per questo possono essere usati anche in codice C o Fortran tramite l'interfaccia offerta da OCaml per questi linguaggi, favorendo così il riuso (con parziali modifiche) di codice preesistente.

I due layout di memoria differiscono per

- la memorizzazione degli elementi della matrice: per righe nel layout C e per colonne nel layout Fortran;
- la numerazione degli indici: il primo elemento dell'array ha indice 0 nel layout C mentre ha indice 1 nel layout Fortran .

Vediamo brevemente come utilizzare i Bigarray e successivamente il ruolo che hanno in *OCamlP3l*.

Il tipo Genarray è così definito

```
type ('a, 'b, 'c) Genarray.t
```

dove `'a` è il tipo OCaml dell'array, `'b` il tipo effettivo e `'c` il layout di memoria (`c_layout` o `fortran_layout`).

Per creare un Genarray si usa la funzione

```
val create :  ('a,'b) Bigarray.kind ->

       'c Bigarray.layout-> int array -> ('a,'b, 'c) Genarray.t
```

dove il primo parametro rappresenta il tipo degli elementi numerici contenuti nel `Genarray`, il secondo rappresenta il layout di memoria (`c_layout layout` o `fortran_layout layout`), il terzo definisce le dimensioni dell'array. Ad esempio, per creare una matrice `3x5` di `float` a 32 bit con layout C scriviamo:

```
let genarray = Genarray.create float32 c_layout [|3;5|];;
```

Per accedere in lettura e in scrittura ad un elemento del Genarray si usano le funzioni

```
val get :  ('a, 'b, 'c) Genarray.t-> int array -> 'a

val set :  ('a, 'b, 'c) Genarray.t-> int array -> 'a -> unit
```

dove il secondo parametro è un array che contiene gli indici dell'elemento cui si vuole accedere. Ad esempio se vogliamo sostituire all'elemento `(0,2)` di `genarray` il suo quadrato possiamo scrivere il codice seguente:

```
1.let old = Genarray.get genarray [|0;2|] in

2.Genarray.set genarray [|0;2|] (old*old);;
```

Il nuovo skeleton riceve uno stream di ingresso formato da liste di `Genarray` di elementi di un certo `kind` e restituisce uno stream formato da liste di `Genarray` di elementi di un altro `kind`.

Per dichiarare la dimensione dei `Genarray` in ingresso e uscita - al fine di poterne controllare la corrispondenza a tempo d'esecuzione - e il tipo dei `Genarray` di uscita - al fine di poterli creare effettivamente - sono stati introdotti i tipi

```
type array_decl = int list
```

```
type in_decl = In of array_decl list

type ('a, 'b) out_decl = Out of ('a, 'b) Bigarray.kind *
   array_decl list
```

La dichiarazione nell'esempio del prodotto di matrici P^3L mostrato in 2.3.1

`in (int a[10][10], int b[10][10]) out (int c[10][10])`

con questi nuovi tipi diventa

`In([[10;10];[10;10]]) Out(int32, [[10;10]])`

e indica che lo stream di ingresso è formato da elementi di tipo lista di (due) `Genarray` bidimensionali 10x10 e lo stream di uscita è formato da elementi di tipo lista di (un) `Genarray` bidimensionale 10x10.

Come in P^3L la funzione sequenziale che deve fornire l'utente deve essere scritta in termini del processore virtuale; ma nel nostro nuovo skeleton i dati in ingresso al processore virtuale sono sempre rappresentati da `Genarray`. In particolare riguardo alle distribuzioni viste in 6.1.2. abbiamo che il generico processore virtuale `(i,j)` nei casi:

`(distr. 1) BodyIn [[One(1);All], [All;One(2)]] BodyOut[[One(1);One(2)]]`

riceve due `Genarray` monodimensionali corrispondenti alla riga `i` del primo `Genarray` e la colonna `j` del secondo e restituisce un elemento del `Genarray` di uscita

`(distr. 2) BodyIn [[All]] BodyOut [[One(1);One(2)]]`

riceve una lista composta dall'intero `Genarray` monodimensionale elemento dello stream di ingresso

`(distr. 3) BodyIn [[One(1);One(2)]] BodyOut [[One(1);One(2)]]`

riceve un `Genarray` monodimensionale composto da un unico elemento corrispondente all'elemento `(i,j)` del `Genarray` elemento dello stream di ingresso e restituisce un elemento del `Genarray` di uscita

`(distr. 4) BodyIn [[Ranf(1,(-a,b));One(2)]] BodyOut [[One(1);One(2)]]`

riceve un `Genarray` monodimensionale composto da *(a+b+1)* elementi corrispondenti agli elementi `(i-a,j),(i-a+1,j),...(i,j),(i+1,j),...(i+b,j)` del `Genarray` elemento dello stream di ingresso e restituisce un elemento del `Genarray` di uscita. Come vedremo in 6.2.2 in questo caso esistono degli elementi di bordo che eccedono la matrice di input; per questo è stato introdotto il parametro di tipo

```
type bounded = Bounded | Unbounded
```

per indicare se tali elementi eccedenti debbano contenere valori fasulli (`Unbounded`) oppure i valori consecutivi sul lato opposto della stessa dimensione (`Bounded`).

La funzione sequenziale ha in ingresso una lista di `Genarray` e restituisce una lista di elementi del tipo del `kind` in `Out` che vanno a formare nell'ordine i `Genarray` dello stream di uscita; a lista contenuta nel costruttore `Out` deve avere la stessa lunghezza della lista restituita dalla funzione sequenziale, se le lunghezze delle due liste sono diverse si avrà un errore a tempo di esecuzione.

6.1.4 Processori virtuali/reali

La forma dei processori virtuali dichiarata in P^3L dalla clausola `nworker x,y,..` è rappresentata con il tipo

```
type nworker = Worker of int list;;
```

ad esempio `nworker 2,4` diventa `Worker [2;4]`.

Anche in *OcamlP3l* gli interi nella lista che segue `Worker` determinano il numero di processori reali; il numero di dimensioni della matrice dei processori virtuali è lo stesso della matrice dei processori reali ed è determinato dal numero di questi interi, ovvero la lunghezza della lista del costruttore `Worker`.

Gli array di output hanno la stessa forma e dimensioni dei processori virtuali, e ogni processore virtuale `(i,j)` restituisce l'elemento `(i,j)` degli array di output. La lista di `indexpattern` che rappresenta ogni array di output ha la stessa lunghezza della lista contenuta in `Worker` e tutti gli elementi devono essere di tipo `One(i)` con `i` tutti diversi tra loro.

Ogni processore reale restituisce la partizione corrispondente di elementi dell'array di output, ad esempio nel caso

`Worker ([2;2])  BodyOutList([[One(1);One(2)]]) Out ([[40;20]])` ogni processore reale detiene una partizione dell'array di output di dimensioni `20x10`, più precisamente abbiamo la seguente distribuzione di dati ai processori reali:

processore virtuale	One(1)	One(2)
(0,0)	[0...19]	[0...9]
(0,1)	[0...19]	[10...19]
(1,0)	[20...39]	[0...9]
(1,1)	[20...39]	[10...19]

dove con la notazione *[a...b]* si intende l'intervallo di indici da a a b.

6.1.5 Iterazione

La map *OcamlP3l* aggiunge alla map di P^{3}L la possibilità di esprimere, oltre alla clausola `endmap` che chiude lo skeleton, una condizione di iterazione. La condizione può essere semplice o dipendere globalmente dai valori dell'array calcolato ad ogni iterazione. Questa possibilità è espressa dai tipi

```
type ('a, 'b) endmap =
  | Times of int
  | Funcs of ('a->'b) * (('b -> 'b -> 'b) * 'b) * ('b ->
     bool)
type ('a, 'b) endmapwith =  EndMap | EndMapWith of ('a, '
   b) endmap
```

Dunque il nostro nuovo skeleton ha una struttura di questo tipo

`map` *`InArrayDecl BodyInList OutArrayDecl BodyOutList`* `(`

`inner-skel`

)

endmapwith

Con il costruttore `EndMap` si rappresenta la terminazione della map senza iterazioni; con il costruttore `EndMapWith` si introduce una condizione di iterazione.

La condizione di terminazione è espressa dal tipo `endmap`:

- il costruttore `Times(n)` indica n iterazioni senza condizione sull'output;

- il costruttore `Funcs (f, (g , init), cond)` indica che la condizione di terminazione è calcolata sull'array Y calcolato ad ogni iterazione secondo questo algoritmo:

```
1  foreach y in Y do
2    init:= g (f (y) , init)
3  done;
4  if cond(init) then end-map
```

  Vediamo che a ogni elemento `y` viene applicata la funzione `f`; la funzione `g` viene applicata similmente ad una reduce (ma a partire da un elemento iniziale `init`) all'array modificato da `f` e calcola un valore al quale viene infine applicata la funzione `cond`. Se la funzione `cond` restituisce `true` l'iterazione della map termina.

  Tralasciando `init` possiamo vedere la nostra nuova map con condizione di terminazione come una composizione di due map e una reduce in loop, più precisamente, indicando con `s` la funzione sequenziale calcolata dal processore virtuale, possiamo scrivere che la semantica informale del nuovo skeleton in termini degli skeleton visti nel capitolo 1 è

  loop (map s) (cond (reduce g (map f)))

  Trattandosi di una reduce la funzione `g` in `Funcs` deve godere della proprietà associativa e commutativa.

  Vediamo alcuni esempi di utilizzo:

`Funcs ((fun a ->a*a>100),((fun a b -> a && b), true),(fun a -> a))`

indica che la map itera finché i quadrati di tutti gli elementi dell'array calcolato non sono maggiori di 100.

`Funcs ((fun a ->a*a>100),((fun a b -> a || b), true),(fun a -> a))`

indica che la map itera finché almeno un quadrato di un elemento dell'array calcolato non è maggiore di 100.

`Funcs ((fun a ->if a>100 then 1 else 0),((fun a b -> a + b), true),(fun a -> a>10))`

indica che la map itera finché non ci sono almeno 10 elementi dell'array calcolato maggiori di 100.

6.1.6 Un esempio completo

Dopo quanto detto nei paragrafi precedenti possiamo finalmente vedere la dichiarazione della funzione introdotta per restituire il nuovo skeleton.

```
val map:
  ?col:int -> ?colv:int list -> in_decl -> ('k,'l)
     out_decl -> nworker -> body_in_list -> body_out_list
      -> (('a,'b,'c) Genarray.t list ,  'd list, 'e, 'f,
     'k, 'l) p3ltree->  ('e, 'f) endmapwith  -> (('a,'b,'
     c) Genarray.t list, ('d, 'l, 'c) Genarray.t list, 'e
     , 'f, 'k, 'l) p3ltree
```

Questa dichiarazione esprime le dipendenze tra i tipi introdotti finora: in particolare lo skeleton annidato è di tipo:

`(('a,'b,'c) Genarray.t list , 'd list, 'e, 'f, 'k, 'l) p3ltree`

che esprime con i primi due parametri i tipi di ingresso e di uscita della funzione sequenziale; mentre lo skeleton restituito è di tipo

`(('a,'b,'c) Genarray.t list, ('d, 'l, 'c) Genarray.t list, 'e, 'f, 'k, 'l) p3ltree`

che esprime con i primi due parametri i tipi dello stream di ingresso e di uscita dello skeleton; vediamo che il tipo `'d` restituito dalla funzione sequenziale si ripete nel tipo del `Genarray` nella lista che costituisce l'elemento dello stream di uscita.

Notiamo che il tipo `p3ltree` ha dei parametri ulteriori rispetto a quelli visti nel capitolo 4, questo a causa dell'introduzione dei `Bigarray` e del loro `kind` e della presenza del tipo `endmap`.

I vincoli di tipo espressi dalla dichiarazione della funzione `map` sono stati introdotti al fine di poter avere un controllo, a tempo di compilazione, della correttezza dei tipi della funzione sequenziale definita dall'utente.

Vediamo un semplice esempio di utilizzo del nuovo skeleton. In Code 34 è mostrato il codice relativo ad un'applicazione che raddoppia gli elementi di un array bidimensionale 40x20 finché tutti gli elementi non sono maggiori di una certa soglia (nell'esempio 700).

```
1  let mapskel = map  Unbounded (In ([[40;20]])) (Out(int32,
       [[40;20]]))
2   (Worker([10;10])) (* matrice dei worker a cui
       distribuire gli elementi della matrice in ingresso *)
3   (BodyIn ([[One(1);One(2)]])) (BodyOut([[One(1);One(2)]])
       )
4   (seq(
5     fun _ x->
6      let a  = (List.nth x 0) in (* la funzione riceve in
          ingresso una lista di lunghezza 1 *)
7      let el = Genarray.get a [|0;0|] in (* trattandosi di
          scatter il processore virtuale riceve un Genarray
          formato da un unico elemento*)
8      [el*2] (* restituisce una lista di 1 intero *)
9    ))
10  (EndMapWith (Funcs ((fun a -> a > 700),((fun a b -> a &&
        b ), true), fun b -> b))))
```

```
11        ;;
```

Code 34: Applicazione map che raddoppia gli elementi di un Genarray finchè non sono tutti maggiori di 700.

L'array di output all'iterazione i diventa l'array di input all'iterazione $i+1$, in caso di m array di input e di n array di output. Se $n<m$ gli n array di input all'iterazione i diventano i primi n array di input all'iterazione $i+1$, gli ultimi $m - n - 1$ restano invariati; se $n>m$ i primi m array di output all'iterazione i diventano gli m array di input all'iterazione $i+1$.

Rispetto a P^{3}L ciò che deve scrivere l'utente è più complesso; questo è dovuto al fatto che P^{3}L è un linguaggio e *OcamlP3l* una libreria, e l'utente deve fare esplicito uso di strutture dati anziché di parole chiave del linguaggio. Nel capitolo 8 daremo un'indicazione su come superare questa limitazione.

6.2 Implementazione

Abbiamo visto in 4.5 quali sono i passi per inserire un nuovo skeleton nella nostra libreria, li vedremo ora applicati all'inserimento del nuovo skeleton `map`. Prima di vedere i nuovi template è il caso di spiegare brevemente gli algoritmi utilizzati per la distribuzione dei dati, e per il calcolo delle condizioni di terminazione descritti in [Li03].

6.2.1 Algoritmo di distribuzione

In 6.1.4 abbiamo mostrato un esempio di associazione degli indici di una matrice di output alla matrice dei processori reali, vediamo ora come calcolare quest'associazione nel caso più generale degli array di input.

Siano i parametri `Worker (matrix)` `In([`$\texttt{dims}_1$`;`$\texttt{dims}_2$`;...;`$\texttt{dims}_n$`])`, per ogni array di input in posizione `j` scorriamo la lista di `indexpattern` corrispondente in `BodyIn` e per ogni suo elemento `p` in posizione `q`:

– se `p` è di tipo `One(d)` la dimensione della partizione di ogni worker è data dal rapporto (`dim`) tra la dimensione in posizione `q` di $\texttt{dims}_\texttt{j}$ e la dimensione in posizione `d` in `matrix` ; in particolare il worker `i` sulla dimensione `d` detiene l'intervallo di indici della dimensione `q` `[i*dim...(i+1)*dim]`.

In Fig. 13 vediamo con diversi tratteggi l'assegnazione delle partizioni ai diversi worker nel caso di

```
In([[6;6]]) Worker([2;2]) e BodyIn([[One(1);One(2)]]);
```

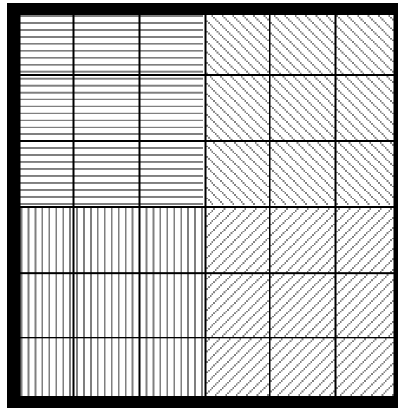

Fig. 13: Distribuzione di una matrice 6x6 ad una matrice di worker 2x2 in caso di pattern [One(1);One(2)]

– se `p` è di tipo `Ranf(d, (-x,y))` la dimensione della partizione di ogni worker è data dal rapporto (`dim`) tra la dimensione in posizione `q` di $\texttt{dims}_\texttt{j}$ e la dimensione in posizione `d` in `matrix` più `y+x`; in particolare il worker `i` sulla dimensione `d` detiene l'intervallo di indici della dimensione `q` `[i*dim-x...(i+1)*dim+y]`.

In Fig. 14 vediamo con diversi tratteggi l'assegnazione delle partizioni ai diversi worker nel caso di

```
In([[6;6]]) Worker([2;2]) e BodyIn([Ranf(1,(-1,1));One(2)]]).
```

Vediamo che esistono delle porzioni ai bordi della dimensione `Ranf` che eccedono la matrice di input, e sono occupati da valori fasulli oppure dai valori consecutivi all'estremo opposto a seconda del parametro `bounded`; le intersezioni tra i tratteggi indicano gli elementi condivisi tra i diversi worker.

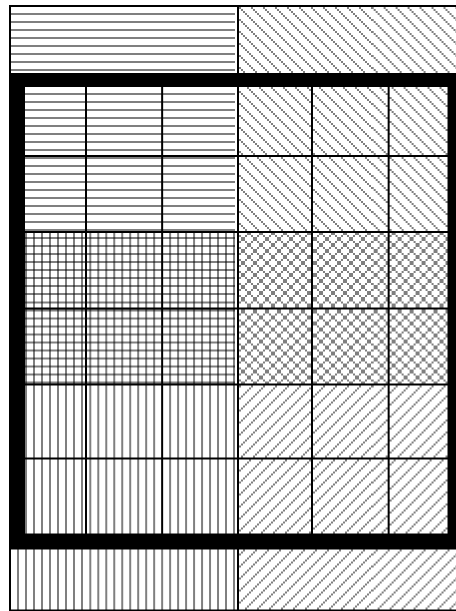

Fig. 14: Distribuzione di una matrice 6x6 ad una matrice di worker 2x2 in caso di pattern [Ranf(1,(-1,1));One(2)]

– se `p` è di tipo `All` la dimensione della partizione di ogni worker è data dalla dimensione N in posizione `q` di $\texttt{dims}_{\texttt{j}}$; in particolare ogni worker detiene l'intervallo di indici della dimensione `q` `[0...N]`.

In Fig. 15 vediamo con diversi tratteggi l'assegnazione delle partizioni ai diversi worker nel caso di

```
In([[6;6]]) Worker([2;2]) e BodyIn([[All;One(2)]]);
```

anche in questo caso le intersezioni tra i tratteggi indicano gli elementi condivisi tra i diversi worker.

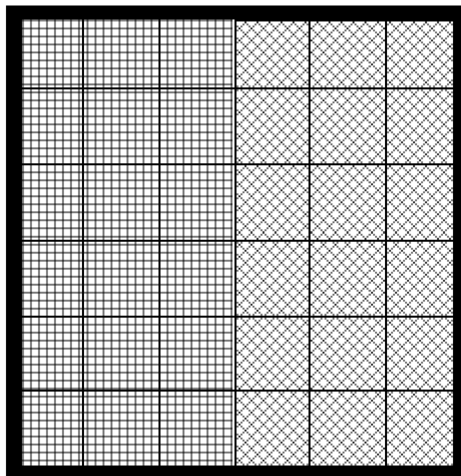

Fig. 15: Distribuzione di una matrice 6x6 ad una matrice di worker 2x2 in caso di pattern [All;One(2)]

Per ogni array di input viene calcolata una lista di associazioni tra indici di worker e porzioni dell'array; se è presente un `indexpattern` di tipo `All` in posizione `j` ci troviamo di fronte a una multicast e l'associazione è tra una partizione dell'array e un insieme di worker. Per indicare la differenza tra un insieme di worker ed un worker unico si utilizza un indice di worker con il valore speciale `-1` nelle posizioni occupate dagli `All`. Ad esempio, nel caso `j = 0` , la coppia `([0;0], [(0,5);(0,5)])` indica l'associazione tra il worker in posizione `(0,0)` e la porzione `[(0,5) (0,5)]` di un array bidimensionale, mentre la coppia `([-1;0], [(0,10);(0,5)])` indica l'associazione tra tutti i worker lungo la colonna `0` e la porzione `[(0,10) (0,5)]` di un array bidimensionale. La multicast è eseguita secondo un algoritmo distribuito sui worker, meno costoso rispetto ad un algoritmo centralizzato in cui l'emettitore invia i dati a tutti i worker. L'emettitore invia la partizione interessata, insieme all'indice di worker associato, al primo worker della dimensione `j` della matrice dei worker; quest'ultimo provvede ad inviare la partizione ricevuta agli altri worker lungo la stessa dimensione. Se gli `indexpattern` di tipo `All` sono più di uno, ogni worker che ha ricevuto la partizione (dal primo worker) provvede ad inoltrarla agli altri worker lungo la dimensione su cui è presente l'`indexpattern All` successivo. Ogni

worker decide se inoltrare o meno la partizione ricevuta e su quale dimensione confrontando il proprio indice con quello ricevuto assieme alla partizione: se questi sono uguali il worker è il destinatario ultimo della partizione e non la inoltra, altrimenti sostituisce il primo `-1` presente nell'indice di worker ricevuto insieme alla partizione con l'indice, nella stessa posizione, del worker a cui inoltra sia la partizione che il nuovo indice di worker. La distribuzione termina quando l'indice di worker inviato insieme alla partizione non contiene più alcun `-1`. Questo procedimento può essere applicato in quanto è dimostrato in [DCLP07] che un'operazione di multicast (o broadcast) equivale ad una scatter (o broadcast) più un insieme di broadcast parallele; in Fig. 16 vediamo un'esemplificazione grafica del procedimento applicato ad una matrice di worker di due dimensioni.

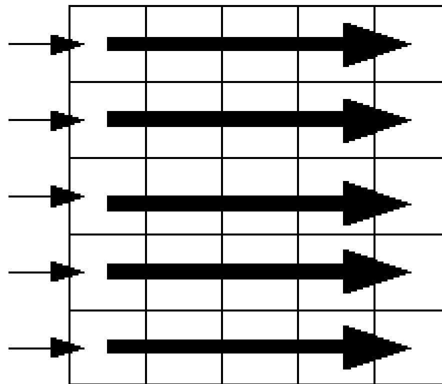

Fig. 16: La multicast equivale a scatter + broadcast

6.2.2 Modello di calcolo

Vediamo più precisamente come avviene il calcolo della matrice di output in funzione della matrice di input e il rapporto tra processori virtuali e reali. Abbiamo più volte ricordato che la matrice dei processori virtuali è isomorfa alla matrice di output e che ogni processore virtuale è responsabile della modifica dell'elemento della matrice di output a lui corrispondente. Inoltre, come visto in 6.1.3, la funzione sequenziale data dall'utente rappresenta il calcolo svolto dal processore virtuale e la dimensione dei parametri di ingresso alla funzione dipende dall'`indexpattern` associato al bigarray di input. Abbiamo anche detto più volte che su processore reale è allocata

una sotto matrice di processori virtuali e una partizione della matrice di input e output composta dagli elementi associati a tali processori virtuali. Rappresentiamo graficamente queste relazioni: come nel paragrafo 6.2.1 indichiamo con tratteggi diversi gli elementi di input distribuiti ai diversi processori reali; con il bordo nero indichiamo l'elemento di output del processore virtuale e con colore grigio gli elementi di input necessari al calcolo dell'output evidenziato. Utilizziamo ancora il caso `Worker ([2;2]) In([6;6]) Out([6;6])` in cui ogni processore reale contiene `3x3` processori virtuali e altrettanti elementi della matrice di output e deve quindi applicare `3x3` volte la funzione sequenziale a diversi parametri di input per ottenere i `3x3` elementi di output. In Fig. 17 vediamo il caso `BodyIn([[One(1);One(2)]])`: ogni processore reale per calcolare ogni elemento della sotto matrice di output a lui associata deve estrarre l'elemento corrispondente della matrice di input e applicarvi la funzione sequenziale. In figura sono mostrati l'input e l'output necessari al processore reale `(0,0)` per il calcolo dell'elemento `(1,1)` della sotto matrice a esso associata.

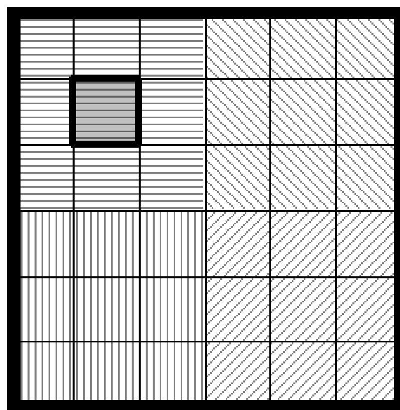

Fig. 17: Calcolo di una matrice 6x6 su una matrice di worker 2x2 in caso di pattern [One(1);One(2)]

In Fig. 18 vediamo il caso `BodyIn([[Ranf(1,(-1,1));One(2)]])`: ogni processore reale per calcolare ogni elemento della sotto matrice di output a lui associata deve estrarre un'array contenente l'elemento corrispondente della matrice di input e i due elementi precedente e successivo lungo la seconda dimensione e applicarvi la funzione

sequenziale. In figura sono mostrati l'input e l'output necessari al processore reale (0,0) per il calcolo dell'elemento (1,1) della sotto matrice a esso associata.

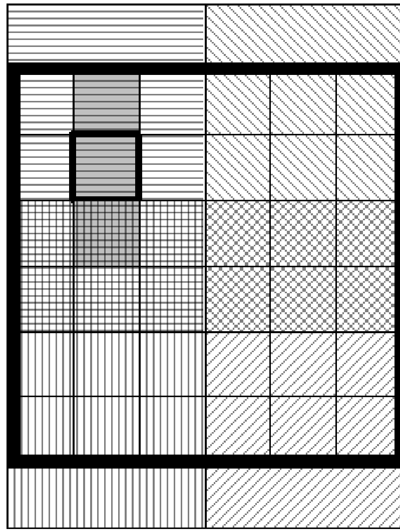

Fig. 18: Calcolo di una matrice 6x6 su una matrice di worker 2x2 in caso di pattern [Ranf(1,(-1,1));One(2)]

In Fig. 19 vediamo il caso `BodyIn([[All; One(2)]])`: ogni processore reale per calcolare ogni elemento della sotto matrice di output a lui associata deve estrarre un'array contenente l'intera colonna cui appartiene della matrice di input e applicarvi la funzione sequenziale. In figura sono mostrati l'input e l'output necessari al processore reale (0,0) per il calcolo dell'elemento (1,1) della sotto matrice a esso associata.

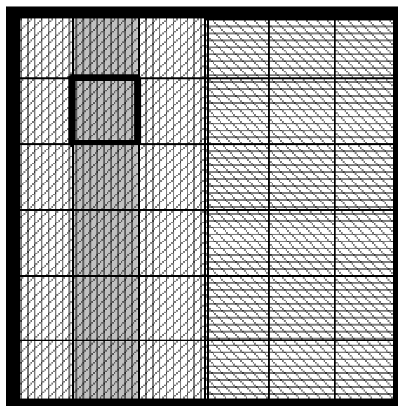

Fig. 19: Calcolo di una matrice 6x6 su una matrice di worker 2x2 in caso di pattern [All;One(2)]

6.2.3 Scambio dati e calcolo della condizione di terminazione

In caso di iterazione e in presenza di `indexpattern` di tipo `Ranf` o `All` si rende necessario uno scambio dati tra processori virtuali ad ogni iterazione ai fini del calcolo all'iterazione successiva. Infatti, come abbiamo già detto, ad ogni iterazione il processore virtuale modifica l'elemento corrispondente dell'array di input, elemento che nel caso in esame è utilizzato in lettura da altri processori virtuali nel corso del calcolo dell'iterazione successiva; ogni processore virtuale invia il proprio elemento modificato agli altri processori virtuali e riceve l'elemento modificato dagli altri. I processori reali modificano partizioni di elementi e di conseguenza si scambiano sottoinsiemi di tali partizioni. I sottoinsiemi delle partizioni di dati da scambiare sono calcolati semplicemente trovando le intersezioni tra le partizioni di output e le partizioni di input di array corrispondenti (calcolate come in 6.2.1). In Fig. 20 vediamo una rappresentazione grafica del caso di array `In([6x6]) Worker([2;2])` e `[Ranf(1,(-1,1));One(2)]`.

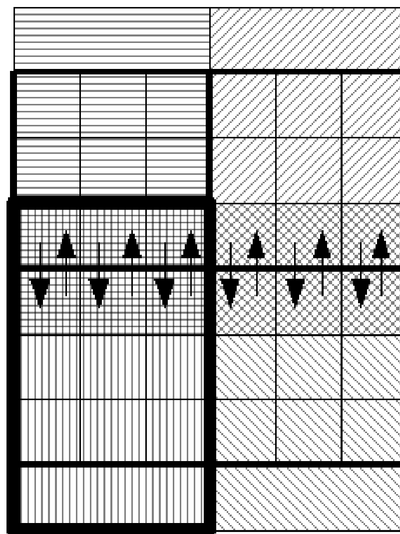

Fig. 20: Rappresentazione della distribuzione di una matrice 6x6 su una matrice di worker 2x2 con [Ranf(1,(-1,1)); One(2)]; le frecce rappresentano gli scambi di elementi tra i worker

In caso di `Funcs (f, (g , init), cond)`, ad ogni iterazione ciascun worker applica all'array di output Y la prima parte dell'algoritmo visto in 6.1.5 , ovvero

```
foreach y in Y do init:= g (f (y) , init)
```

ed invia il valore finale di `init` al collettore, che applica la funzione `g` a tutti gli elementi ricevuti,vi applica `cond` ed invia il risultato ai worker che continuano o terminano il ciclo a seconda dell'esito ricevuto.

6.2.4 Dai p3ltree ai template

Come nel caso dello skeleton `mapvector` in 5.1 sono stati introdotti due nuovi `p3ltree`

```
type ('a, 'b, 'c, 'd, 'e, 'f) p3ltree = ....
  | Map of (int*int list*(('a, 'b, 'c, 'd, 'e, 'f)
     p3ltree * int)* ('c,'d, 'e, 'f)mapPar)
  | Mapseq of (int *  (('a, 'b) io -> ('a -> 'b)) * int *
      ('c,'d, 'e, 'f)mapPar)
```

dove il tipo `mapPar` è così definito:

```
type ('a,'b, 'c, 'd) mapPar =  int array * (int array
   list* indexpattern array list * ('c,'d) kind * int
   array list * indexpattern array list * ('a, 'b)
   endmapwith);;
```

e rappresenta i parametri (visti nella sezione precedente) passati alla map dall'utente e che devono essere propagati fino ai template finali che implementano il nuovo skeleton.

Infatti vediamo che `mapPar` è contenuto anche nei proctemplate introdotti per l'emettitore, il worker e il collettore del nuovo skeleton:

```
type ('a, 'b, 'c, 'd) proctemplate = ...
  | Mapemit of int * ('a,'b, 'c, 'd)mapPar
  | Mapworker of int * ('a, 'b, 'c, 'd)mapPar
  | Mapcollector of int * ('a, 'b, 'c, 'd)mapPar
```

In Code 35 vediamo un frammento della funzione `map`.

```
1 let map ?(col=0) ?(colv=[]) (inl : in_decl) (outl: ('j, '
    k) out_decl) (nw : nworker) (body_in : body_in_list) (
    body_out: body_out_list) (tree: (('c,'d,'e) Genarray.t
     list , 'f list, 'a , 'b, 'j, 'k) p3ltree) (endmap)
    =
2  let nwa = match mw with Nworker (w) -> w in
3  let par = ... in (* creazione struttura mapPar *)
4  let n = (List.fold_right ( * ) nwa  1) in
5  let worker = match tree with Seq(c, f) -> Mapseq(c , f ,
      n , par)
6  | _  -> ... in
7  let skel = Map (col, colv, (worker, n ), par) in
8  (Obj.magic (skel) : (('c,'d,'e) Genarray.t list, ('f, 'k
     , 'e) Genarray.t list, 'a, 'b, 'j, 'k) p3ltree) ;;
```

Code 35: Funzione map che restituisce il nuovo skeleton.

Per uniformare il trattamento del nuovo `p3ltree` e della nuova `p3ltreeexp` a quello degli altri skeleton, la matrice dei worker definita nel parametro `nw` viene trattata come una lista (monodimensionale) e il numero dei worker totali è dato dalla moltiplicazione di tutti gli elementi della dimensione della matrice (linea 4).

Il nuovo `innode` che rappresenta il grafo del nuovo skeleton è:

```
type ('a, 'b, 'c, 'd, 'e, 'f) innode = ...
  | Mapexp of ('a, 'b, 'c, 'd, 'e, 'f) node * ('a, 'b, 'c
     , 'd, 'e, 'f) node *  (('a, 'b, 'c, 'd, 'e, 'f)
     p3ltreeexp list) * config option
```

In Code 36 vediamo come è stata modificata la funzione `expand`, aggiungendo i match case relativi ai nuovi `p3ltree`; attraverso la funzione `expand` il `mapPar` viene propagato ai `proctemplate` `Mapworker` (linea 3), `Mapemit` (linea 8) e `Mapcollector` (linea 9).

```
1 let rec expand (col:color) =
2  ...
3  | Mapseq (c, f, n, par) -> Leaf (Mapworker (n, par),
     None, Some (Seqfun f), color c) (* creazione della
     foglia *)
4  | Map (c, cv, (t, n), mappar) ->
5   let tmpcol = color c in
6   let tmpcolist = nvlist n tmpcol cv in
7   Innode (Mapexp (*creazione dell'innode *)
8   ((Mapemit (n , mappar), None, None, tmpcol),
9   (Mapcollector (n, mappar), None, None, tmpcol),
10    List.map (fun x -> expand x t) tmpcolist,
11  None))
```

Code 36: La funzione expand modificata con i due nuovi match case relativi ai nuovi p3ltree.

Un'altra modifica si introduce nella funzione `leafs_of` (Code 37), in quanto l'emettitore del nuovo skeleton ha bisogno di una funzione di selezione dell'indice del worker a cui inviare i dati diversa da `rr()`, a causa della linearizzazione della matrice dei worker effettuata in `map`. La nuova funzione, `normalizeindex`, traduce gli indici all'interno della matrice multidimensionale dei worker nell'indice che tale worker ha nella lista monodimensionale che li contiene ed è passata all'emettitore tramite la nuova action `MapOutChanSel`.

```
1 let confmapemit (pt, conf, _, col) =
2  (pt, conf, Some (MapOutChanSel (normalizeindex)), col);;
3 let rec leafs_of = function
4   ...
5  | Innode (Mapexp (n1, n2, tl, _))  ->
6   Leaf (confmapemit n1) :: Leaf (confcollect n2) ::
7   List.flatten (List.map leafs_of tl)
```

Code 37: La funzione confmapemit e la funzione leafs_of modificata con il nuovo match case relativi al nuovo innode.

Come abbiamo visto ai fini dell'implementazione dello stencil fisso iterativo i worker devono scambiarsi dati calcolati ad ogni iterazione e quindi essere connessi tra di loro; è necessario anche un canale di comunicazione dal collettore ai worker ai fini del calcolo della condizione di terminazione tramite `Funcs`. Di conseguenza, come avvenuto per lo skeleton `reducevector` visto in 5.2, si rende necessaria una modifica in `bindchan`, la vediamo in Code 38, dove alle linee 9 e 10 sono inserite le nuove connessioni tra i `node`.

```
1  let rec bindchan cinl coutl = function
2   ...
3   | Innode
4    (Mapexp (n1, n2, tl, (Some (Inconf (Some fin, Some fout
       )) as c))) ->
5    let winl, woutl = List.split (List.map getchans tl) in
6    let coll = pe_of_node n2  in
7     Innode
8      (Mapexp (chconnect cinl winl n1,
9      chconnect woutl (coutl@winl) n2, (* connessione in
         uscita tra collettore e worker*)
10     List.map (bindchan (fin::coll::woutl) (fout::winl))
         tl, c))
```

Code 38: La funzione bindchan modificata per connettere i worker tra di loro e al collettore.

Le altre funzioni della catena di chiamate in `p3ldo` vanno modificate semplicemente inserendo il match case relativo a `Mapexp` con codice analogo agli altri match case.

6.2.5 I template

Vediamo i nuovi template corrispondenti ai `proctemplate` introdotti in 6.2.3; in `nodeconfiguration` abbiamo aggiunto tre nuovi match case

```
1 let nodeconfiguration ch =
2  ...
3 match (templ,f) with
4  ...
5  | ((Mapemit (_, pars)) , MapOutChanSel f) -> mapetempl f
      pars (setup_receive_chan myvpdata) coutl
6  | (Mapworker(_,pars)  , Seqfun f) -> mapseqtempl (Obj.
     magic f : (unit -> ('a, 'b, 'c) Bigarray.Genarray.t
     list -> 'd list)) pars
7  | (Mapcollector (n, pars) ,InChanSel f) -> mapctempl
     pars (setup_receive_chans inl myvpdata " Mapcoll ") (
     coutl)
```

Code 39: La funzione nodeconfiguration con i nuovi match case relativi ai nuovi proctemplate.

da cui notiamo che le nuove funzioni in `Template` hanno tra i parametri `mapPar`.

Prima di vedere l'implementazione dei nuovi template presentiamo due tipi di dato che questi usano; sono contenuti in un nuovo modulo `P3lmap`, che contiene anche altre funzioni di utilità nell'implementazione della map:

```
type workerindex = int array
```

rappresenta l'indice del worker all'interno della matrice di worker `Worker`

```
1 type range = (int * int) array
```

rappresenta una partizione di un bigarray; per ogni dimensione due interi rappresentano l'estremo inferiore e l'estremo superiore degli indici del bigarray contenuti in quella dimensione.

In Code 40 vediamo la funzione che implementa l'emettitore:

```
val mapetempl :
(Commlib.vp_chan list ->(int array * int array)-> Commlib.vp_chan)->
('a, 'b, 'c, 'd)Parp3l.mapPar->Commlib.vp_chan ->Commlib.vp_chan list ->
unit
```

```
1  let mapetempl f (pars: ('a, 'b, 'c, 'd)mapPar) ic ochl =
2   let nw, bound, (adeclL, indexL, _ ,_,_,_ )  = pars in (*
       parametri *)
3   (* creazione della lista di associazione tra worker e
      partizioni *)
4   let conf: (workerindex * range) array list = List.map2 (
      config nw) adeclL indexL in
5   let f = f ochl in (* inizializzazione della funzione di
      traduzione da indice della matrice a indice nella
      lista dei worker *)
6   let ocl = ref [] in (* in caso di multicast l'emettitore
       non comunica con tutti i worker; ai fini della
      terminazione corretta si mantengono informazioni sui
      worker a cui ha inviato dei dati *)
7   let sendindex i=cl_send (f(i, nw)) (UserPacket(i,(-1),[
      Workerindextag]))
8   in
9   apply_all (getInitRange nw) sendindex; (* invio a tutti
      i worker l'indice *)
10  let notfinished = ref true in
11  while !notfinished do
12  try
13   match receive ic with
14    | UserPacket(p, seqn, tl) ->
15    (* pos indica la posizione dell'array all'interno della
         lista dei parametri della funzione sequenziale *)
```

```
16        let pos = ref 0 in
17        List.iter2 (fun aconf b ->
18        (Array.iter (fun (wi, r) ->
19        (* si ottiene l'indice del worker nella lista ochl e
             si aggiunge a ocl *)
20        let ch = f (wi, nw) in
21        ocl:= !ocl @ [ch];
22        (* si invia la partizione assegnata a ciascun worker
             *)
23        cl_send (ch) (UserPacket(((extract_packet b r)),seqn,
             Maptag (!pos,wi) :: tl))
24    )   aconf);
25        pos:= !pos + 1;
26    ) conf p;
27     | EndStream  ->
28        List.iter (fun x -> cl_send x EndStream) !ocl;
29        List.iter vp_close ochl; vp_close ic;
30        notfinished := false
31     | EndOneStream  ->
32        List.iter (fun x -> cl_send x EndOneStream) !ocl;
33   with ...
34  done ;;
```

Code 40: La funzione mapetempl

L'emettitore inizia la computazione inviando a tutti i worker l'array che rappresenta il loro indice in termini della matrice dei worker (linee 7-9); tale indice è utile ai worker ai fini dell'esecuzione della multicast distribuita e tale scopo è stato introdotto il nuovo `tag Workerindex`. In seguito l'emettitore inizia a ricevere lo stream di liste di bigarray in ingresso e a partizionare, applicando la funzione `extract_packet`, e inviare al corrispettivo worker ogni bigarray singolarmente (li-

nee 16-26). Si segue l'algoritmo in 6.2.1, implementato dalla funzione `config` , il cui risultato è assegnato all'identificatore `conf` (linea 4) . Ogni bigarray della lista in ingresso è inviato singolarmente in quanto ha un `indexpattern` differente dagli altri e quindi può seguire un percorso di distribuzione tra worker diverso in caso di multicast.

L'implementazione dell'algoritmo distribuito per la multicast rende necessaria l'introduzione del nuovo `tag Maptag of int * P3lmap.workerindex`.
Il primo parametro intero in `Maptag` indica la posizione del bigaray inviato all'interno della lista dei parametri in ingresso, il secondo parametro indica l'indice del worker destinatario. Il worker controlla tale indice per decidere se inoltrare o meno il bigarray ricevuto in caso di multicast.

Inoltre, dal momento che l'algoritmo distribuito per la multicast prevede che l'emettitore non comunichi direttamente con tutti i worker, ai fini di una corretta terminazione teniamo traccia dei canali effettivamente utilizzati nella variabile `ocl` (linea 21) e la utilizziamo per l'inoltro dei messaggi `EndStream` (linea 28) e `EndOneStream` (linea 32)

Vediamo ora il la funzione che implementa il worker:

```
val mapseqtempl :

  (unit -> ('a, 'b, 'c) Bigarray.Genarray.t list -> 'd list) -> ( 'd, 'f,
'd, 'h) Parp3l.mapPar ->

  Commlib.vp_chan list -> Commlib.vp_chan list  -> unit
```

Notiamo subito che i frammenti di codice in Code 41 delineano un codice molto più complesso e lungo di quello dei worker degli altri skeleton sia a causa dell'algoritmo distribuito per l'esecuzione della multicast, sia a causa della gestione dello stencil e della condizione di iterazione.

```
1 let mapseqtempl f ( pars : ('c, 'b, 'c, 'd) mapPar ) icl
     ocl =
2  let nw, b, (adeclL , indexL, outkind, odeclL, oindexL ,
      endMap) = pars in
```

```
3   let ic, icoll, winl = .... (* canali di input :
       emettitore, collettore, worker*)
4   let oc, woutl = .... (* canali di output : collettore,
       worker *)
5   let oclend = ref [oc] in (* lista dei canali con cui
       avviene effetiva comunicazione per trattare la
       terminzione *)
6   let n_pars = List.length adeclL in
7   (* struttura per gestire la ricezione separata di tutti
       i parametri *)
8    ....
9   let f = instanciate_io f ic oc in (* funzione
       sequenziale da calcolare *)
10  let myindex = match receive ic with (* ricezione proprio
        indice di matrice dall'emettitore *)
11  | UserPacket (ind, (-1), Workerindextag::r ) ->ind
12  | _ -> failwith "wrong type of message"  in
13  let notfinished = ref true in
14  while !notfinished do
15  try
16  (* si riceve dall'emettitore o dagli altri worker in
       caso di multicast *)
17   let (pkt, ich) = receive_any (ic::winl) in
18   (match pkt with
19    | UserPacket(big, seqn,  (Maptag(i, config_index) as m
         )::tl) ->
20     (* algoritmo di distribuzione multicast *)
21     distribute oclend myindex normalize nw tl
          config_index big i seqn ;
22     (* ricezione parametri *)
23     ....
```

```
24      let nc = ... (* numero parametri ricevuti finora *)
25      (* tutti i parametri sono stati ricevuti *)
26      if (nc = n_pars) then
27      begin
28       let bigL : (('x, 'y, 'l) Genarray.t) list = ... in
            (* lista dei parametri su cui eseguire il calcolo
             *)
29       (* calcolo della prima o unica iterazione *)
30       let outL = ref (genApply nw adeclL indexL  outkind
            odeclL  oindexL bigL f) in
31       (match endMap with
32       | EndMap -> (* caso map semplice, abbiamo gia' fatto
            *) ()
33       | EndMapWith e ->(* caso iterativo*)
34         (* 1 *)(* calcolo delle eventuali intersezioni per
              lo scambio dati *)
35         ....
36         (* 2 *)(* otteniamo cond funzione per il calcolo
             della condizione di terminazione, contiene
             anche le eventuali comunicazioni col collettore
              *)
37         ....
38         let localrange = ... (* range della partizione
             dell'array di input modificato dal calcolo *)
39         (* Obj.magic e' un cast, per il type system il
             bigarray di input e il bigarray di output non
             hanno lo stesso tipo, ma devono averlo
             effettivamente *)
40         let inL = (Obj.magic bigL : ('c,'d,'l) Genarray.t
             list) in
41         while cond !outL do
```

```
42          (* modifica array di input con risultati ottenuti
              all'iterazione precedente *)
43           for i = 0 to (List.length !outL - 1) do
44            inject_packet (List.nth localrange i) (List.nth
                inL i) (List.nth !outL i)
45           done;
46           (* 3 *) (* scambio dati *)
47           ....
48           (* calcolo iterazioni successive alla prima *)
49           outL :=  (genApply nw adeclL indexL  outkind
               odeclL  oindexL (Obj.magic inL: ('x, 'y, 'l)
               Genarray.t list) f);
50          done;
51          (* loop finito o mai iniziato, si invia risultato
              al collettore *)
52          cl_send oc (UserPacket (!outL, seqn, Maptag(-1,
              myindex)::tl));
53      end
54    | EndStream  ->
55        (* 4 *) (* trattamento terminazione stream*)
56    | EndOneStream  ->
57        (* 5 *) (* trattamento terminazione one stream *)
58    with  ...
59  done
60  ;;
```

Code 41: Frammenti della funzione mapseqtempl

Prima di iniziare la computazione vera e propria è necessario ricevere il proprio indice, `myindex`, dall'emettitore (linea 10-12) e in seguito attendere di ricevere tutte le partizioni dei parametri (linee 22-26), che come già detto seguono ognuno un per-

corso diverso a seconda del proprio `indexpattern`. Le partizioni possono arrivare ad ogni worker sia dall'emettitore sia dagli altri worker, per questo vediamo che si usa la funzione per la ricezione non deterministica `receive_any (ic::winl)` (linea 17); ricevuta ogni partizione si provvede ad inoltrarla (se necessario) agli altri worker aplicando la funzione `distribute` (linea 21). Ricevuti tutti i parametri, nella lista `bigL`, (linea 27) si calcola il risultato della prima o unica iterazione, ovvero la lista dei bigarray che rappresentano le partizioni di output del worker (`outL`), attraverso la funzione `genApply` (linea 30) che implementa il modello di calcolo visto in 6.2.2 .

Se il parametro `endmap` è `EndMap` si passa ad inviare il risultato al collettore (linea 52).

Se il parametro `endmap` è di tipo `Times o Funcs` si provvede a calcolare una volta per tutte `(* 1 *)` le eventuali intersezioni tra i `range` di output del worker con i `range` di input degli altri worker (a cui inviare le parti modificate) e tra i `range` di input del worker e i `range` di output degli altri worker (da cui ricevere le parti modificate). Di seguito `(* 2 *)` si ottiene la funzione `cond` che applicata ad `outL` determina la continuazione o meno del ciclo (linea 41). Prima di passare allo scambio dati con gli altri worker `(* 3 *)` e al calcolo dell'iterazione successiva (linea 49) è necessario modificare il bigarray di input con i risultati precedentemente ottenuti (linee 43-45); questo passo può essere ottimizzato nel caso di `indexpattern` di output corrispondente a quello di input semplicemente facendo di `bigL` un riferimento e assegnando `outL` a `bigL` . Al termine del ciclo, come nel caso `EndMap`, si inviano i risultati finali al collettore.

Per una corretta terminazione dell'applicazione è necessario che i messaggi `EndStream (* 4 *)` e `EndOneStream (* 5 *)` seguano gli stessi percorsi indipendenti seguiti dai diversi parametri di input per la distribuzione dall'emettitore ai worker e che quindi ogni worker riceva tanti di questi messaggi quanti sono i parametri nella lista di ingresso. Solo allora ogni worker inoltra il messaggio al collettore per la terminazione anche di quest'ultimo.

In Code 42 vediamo come è ottenuta la funzione `cond`.

```
1 let cond =
```

```
2  match e  with
3   | Times n ->
4     let it = ref (n -1) in
5     fun outL -> it:= !it - 1;
6     !it >= 0
7   | Funcs (f1, (f2, initV), f3) ->
8     fun outL->
9     (* fmap calcola il risultato locale dell'applicazione
         di f2 ad ogni bigarray di output *)
10     let fmap b =
11     let init = ref initV in
12     let fold i = init:= f2 (f1 (Genarray.get b i)) !init
          in
13     (* applica fold a tutti gli elementi del bigarray b
          *)
14     let () = apply_all (getInitRange (Genarray.dims b))
          fold in
15     !init
16   in
17     let local = List.map (fmap) outL in (* applica fmap a
           tutti i parametri di output*)
18     (* invio risultato local al collettore *)
19     let () = cl_send oc (UserPacket (local, seqn, Looptag
          ::tl)) in
20     match receive oc with  (* ricezione valore finale dal
           collettore *)
21     | UserPacket (global, _ ,_ ) -> (not global)
22     | _ -> failwith "not expected type of message"
23   )
```

Code 42: Funzione cond per il calcolo della condizione di terminazione

In caso di `Times` la funzione decrementa un contatore e restituisce vero finché il contatore è maggiore di zero (linee 3-6). In caso di `Funcs` si segue l'algoritmo in 6.2.2 applicando ad ogni bigarray della lista di output la funzione `fmap` (definita alle linee 10-15); la lista dei risultati locali ottenuta viene inviata al collettore (linee 17-19) e dal collettore si attende il risultato globale (linea 20) restituendo la sua negazione (linea 21). Notiamo l'introduzione del nuovo `tag Looptag` che contraddistingue i messaggi scambiati tra i worker e il collettore in questa fase.

Infine vediamo la funzione che implementa il collettore:

```
val mapctempl :
   ('d, 'e, 'f, 'd) Parp3l.mapPar ->
  Commlib.vp_chan list -> Commlib.vp_chan list -> unit
```

```
1  let mapctempl (pars: ('a,'b,'c,'d)mapPar) icl ocl=
2   let oc, woutl = match ocl with root::wl -> root, wl
3   | _ -> failwith "Mapctempl: not enough channel output"
       in
4   ....
5   let notfinished = ref true in
6   (* per gestione calcolo della condizione di terminazione
        *)
7   let loopcount, acc =  ref 0, ref [] in
8   let n = List.length icl in
9   ... (* struttura gestione partizioni *)
10  let nw, _, (_, _ , (outkind: ('c, 'd) kind) , (odeclL:
       int array list), (oindexL : indexpattern array list),
        (endMap : ('a, 'b) endmapwith) ) = pars in
11  ....
12  while !notfinished do
13   try
```

```
14  let inpck, ic = receive_any icl in
15  (
16  match inpck with
17  | UserPacket (bigL, seqn, Maptag(-1, index) :: tl) ->
18    let rangeL = (List.map2 (range_of_worker index nw )
        oindexL odeclL ) in
19    let count, outBigL = ... (* numero partizioni
        ricevute, lista di bigarray di output da riempire
        *) in
20    (* inserimento partizioni ricevute nei bigarray di
        output *)
21    for i = 0 to ((List.length rangeL) - 1) do
22     inject_packet (List.nth rangeL i) (List.nth outBigL
         i) (List.nth bigL i)
23    done;
24    ...
25    (* tutte le partizioni sono state ricevute, outBigL e
        ' completa *)
26    if count = n then cl_send oc (UserPacket(outBigL,
        seqn, tl));
27  | UserPacket (localL, seqn, Looptag:: tl) ->
28  (* ricezione valori locali in caso di Funcs *)
29    let f2, f3 = match endMap with EndMapWith (Funcs (_,(
        f2, _), f3)) ->
30    f2 , f3
31    | _ -> failwith "not expected endmap type" in
32    (* cast necessario per aggirare l'inferenza dei tipi
        *)
33    let redlocal = (Obj.magic localL : 'b list) in
34    loopcount := !loopcount + 1;
35    acc :=  if !loopcount = 1 then redlocal
```

```
36      else List.map2 f2 !acc redlocal;
37      if !loopcount = n then begin
38       List.iter (fun ic -> (* si calcola l'and degli esiti
             ottenuti in caso di piu' array di output*)
39       cl_send ic (UserPacket ((List.fold_right (&&) (List.
           map (f3) !acc) true), seqn,Looptag::tl)))
40       icl;
41       loopcount:= 0;
42      end
43    | UserPacket (_ , _, tl) -> failwith "Internal Error"
44    | EndStream  ->
45      .... (* gestione terminazione *)
46    | EndOneStream  ->
47      ....)
48 with
49 | End_of_file ->
50      .... done ;;
```

Code 43: La funzione mapctempl

In Code 43 vediamo come il collettore in questo nuovo skeleton abbia un doppio compito: oltre a quello di ricomporre i bigarray di uscita (linee 17-26), ha anche quello di calcolare l'esito globale della condizione di terminazione (linee 27-42). Vediamo che in caso di n bigarray di output `f2` è applicata a agli n valori locali ricevuti da ogni worker (linee 35-36) e la funzione `f3` è poi applicata agli n risultati finali ottenendo n esiti booleani dalla cui congiunzione deriva l'esito finale inviato a tutti i worker (linea 39).

Per garantire la possibilità di annidare nel nuovo skeleton un altro skeleton, di tipo task parallel, che rappresenta una parallelizzazione del worker `map`, come nel caso di `mapvector`, è necessario introdurre un nodo che genera lo stream dei dati di input al singolo processore virtuale e li invia allo skeleton annidato. Lo skeleton annidato

esegue la funzione sequenziale sulla partizione di dati ricevuta e reinvia il risultato al generatore di stream per la ricomposizione della partizione iniziale e il successivo invio al collettore `map`.

6.3 Esempi

In questa sezione vediamo alcuni applicazioni note parallelizzate attraverso il nuovo skeleton map in *OcamlP3l*.

Useremo spesso la funzione `printBigaxb` per visualizzare ogni bigarray di output ottenuto

```
1 let printBigaxb a b big =
2 for i = 0 to (a-1) do
3  for j = 0 to (b-1) do
4   Printf.printf "%d "(Genarray.get big [|i;j|]);
5  done;
6 print_newline();
7 done;;
```

Code 44: La funzione di utilità printBigaxb

6.3.1 Mandelbrot

A partire dall'insieme dei punti del piano complesso è possibile determinare l'appartenenza o meno di tali punti all'insieme di Mandelbrot[Man79]. Un punto c del piano complesso appartiene all'insieme di Mandelbrot se la successione $z_n = z_{n-1} + c$ diverge; la condizione di divergenza della successione è $/z_n/ > 2$. L'insieme di Mandelbrot è un'insieme frattale.

Quest'applicazione è parallelizzabile tramite una map semplice a partire da un bigarray bidimensionale di numeri complessi (rappresentati dal tipo Ocaml `Complex.t`), che rappresenta il piano complesso, distribuiti ad una matrice bidimensionale di processori virtuali. Ogni processore virtuale esegue il calcolo della successione fino ad un

determinato valore di n e restituisce 0 se la successione diverge, 1 non diverge. Questa computazione è caratterizzata da un'elevata varianza di calcolo, di conseguenza il carico dei processori virtuali risulta sbilanciato; questo sbilanciamento è ridimensionato dal fatto che su ogni processore reale è allocato un sottoinsieme di processori virtuali. Un'implementazione task parallel sotto forma di farm on demand (come quella esistente per mapvector in precedenza) ovviamente non soffrirebbe di questo sbilanciamento.

In Code 45 vediamo come scrivere il codice di quest'applicazione in *OcamlP3l*

```
1  let n = 100;; (* numero di iterazioni *)
2  let a, b = 128,128;; (* dimensione del piano complesso *)
3  (* creazione del bigarray che rappresenta il piano
     complesso *)
4  let complex_plane a b =
5  let srcA = Genarray.create complex64 c_layout [|a;b|] in
6  for x = 0 to a do
7   for y = 0 to b do
8    Genarray.set srcA [|x;y|]
9    ({Complex.re = (float_of_int x); Complex.im = (
       float_of_int y)});
10  done;
11 done;
12 srcA;;
13
14 let mandel =
15  parfun(function () ->
16   map  Bounded (In ([[a;b]])) (Out(int, [[a;b]]))
17   (Worker([2;2]))
18   (BodyIn ([[One(1);One(2)]])) (BodyOut([[One(1);One(2)
       ]]))
19   (seq(
```

```
20      fun _ x->
21       let a  = (List.nth x 0) in (* il bigarray di input *)
22       let c = Genarray.get a [|0|] in
23       let zi = ref Complex.zero in
24       for i=0 to n do
25       (* calcolo di zi^2+c*)
26        zi:= Complex.add (Complex.mul !zi !zi) c;
27       done;
28      (* condizione di convergenza |zi| < 2.0 *)
29       if (Complex.norm !zi) < 2.0 then [1] else [0]
30       ) )
31    EndMap
32   ) ;;
33 pardo(function ()->
34    let plane = complex_plane a b in
35    let st = P3lstream.of_list [[plane]] in
36    let r  = mandel st in
37    P3lstream.iter (fun el -> List.iter (printBigaxb a b)
        el) r;
38 );;
```

Code 45: L'applicazione mandelbrot

Il codice eseguito dal generico processore virtuale è contenuto nello skeleton `seq` (linee 19-30): ogni processore virtuale riceve un bigarray contentente il punto `c` di tipo `Complex` la cui appartenenza all'insieme di Mandelbrot deve essere verificata. Una volta eseguite le `n` iterazioni (linee 24-27), che calcolano gli elementi `zi` della successione z_n,viene valutata la condizione di convergenza (linea 29) e restituito un `1` in caso di appartenenza, uno `0` altrimenti. Vengono utilizzate le funzioni `Complex.add`, `Complex.mul` e `Complex.norm` rispettivamente per calcolare la somma, il prodotto e il modulo dei numeri complessi. Nella funzione `pardo` vediamo la creazione del

piano complesso (linea 34) tramite la funzione `complex_plane`, la generazione uno stream contenente il bigarray appena creato (linea 35) e l'applicazione della parfun `mandel` a tale stream (linea 36). Il risultato è ottenuto alla linea 37 e stampato semplicemente. Lo stesso risultato potrebbe essere utilizzato da un'applicazione grafica che associa un pixel di colore nero agli elementi `1` e bianco agli elemento `0` contenuti nel bigarray di output in modo da visualizzare l'immagine del frattale ottenuto.

6.3.2 Prodotto di matrici

In Code 46 vediamo il codice OcamlP3l che rappresenta il prodotto di matrici riga per colonna visto in 2.3.1. Ogni processore virtuale opera su una riga e una colonna della matrice di input e restituisce un elemento della matrice di output.

```
1  let a, b = 120,120;; (* dimensione delle matrici *)
2  (* creazione di una matrice contenente valori casuali *)
3  let create a b =
4  let srcA = Genarray.create int c_layout [|a;b|] in
5  for i = 0 to (a - 1) do for j = 0 to (b - 1) do
6   Genarray.set srcA [|i;j|] (Random.int 5)
7  done; done;
8  srcA;;
9
10 let prodmatrix =
11  parfun(function () ->
12   map  Bounded (In ([[a;b]; [b;a]])) (Out(int, [[a;a]]))
13   (Worker([5;5]))
14   (BodyIn ([[One(1);All]; [All; One(2)]])) (BodyOut([[One
      (1);One(2)]]))
15   (seq(
16    fun _ x->
17     let a1 , a2 = (List.nth x 0) , (List.nth x 1) in (*
        le due matrici di input *)
```

```
18      let c = ref 0 in (* il risultato prodotto da ogni
          processore virtuale *)
19      for i = 0 to (b-1) do (* prodotto riga per colonna
          ricevute *)
20       c:= !c + ((Genarray.get a1 [|i|]) * (Genarray.get a2
            [|i|]));
21      done;
22      [!c]
23    )  )
24    EndMap
25   ) ;;
26 pardo(function ()->
27   let srcA , srcB = (create a b) ,  (create b a)
28   let st = P3lstream.of_list [[srcA; srcB]] in
29   let r  = prodmatrix st in
30   P3lstream.iter (fun l ->List.iter (printBigaxb a b) l)
       r
31 );;
```

Code 46: L'applicazione prodotto di matrici

Il codice eseguito dal generico processore virtuale è contenuto nello skeleton `seq` (linee 15-23): ogni processore virtuale riceve in input un bigaray contenente una riga della matrice di input e un bigarray contenente una colonna della matrice di input ed esegue il prodotto scalare tra questi due bigarray (linee 19-21), restituendo il risultato (linea 22). Nella funzione `pardo` vediamo la creazione delle due matrici da moltiplicare (linea 27) tramite la funzione `create`, la generazione uno stream contenente i due bigarray appena creati (linea 28) e l'applicazione della parfun `prodmatrix` a tale stream (linea 29). Il risultato è ottenuto alla linea 30 e stampato semplicemente.

6.3.3 Game of life

Nel gioco della vita abbiamo una matrice di cellule che possono essere vive o morte; nel corso del gioco una cellula rimane in vita solo se ha esattamente due tra gli otto vicini in vita. Le cellule in vita possono essere rappresentate da un `1`, quelle morte da uno `0`. In Code 47 vediamo il codice *OcamlP3l* che esegue 8 passi del gioco della vita: ogni processore virtuale opera su una matrice `3x3` che contiene la cellula associata al processore virtuale e i suoi vicini sulle due dimensioni e restituisce la cellula viva o morta a seconda del numero di vicini vivi. Ad ogni iterazione la matrice di cellule è quella risultante dall'iterazione precedente.

```
1  let a = 10;; (* dimensione della matrice di cellule *)
2  let steps = 8;; (* numero di passi del gioco da eseguire
     *)
3  let create a b = (* creazione di una matrice di cellule
     con la struttura
4      01010101
5      01010101
6      01010101
7     *)
8  let srcA = Genarray.create int c_layout [|a;a|] in
9  for i = 0 to (a - 1) do for j = 0 to (b - 1) do
10 Genarray.set srcA [|i;j|] (if (i mod 2 = 0) && (j mod 2 =
     0) then 1 else 11.0)
11 done; done;
12 srcA;;
13
14 let lifestep =
15  parfun(function () ->
16   map  Bounded (In ([[a;a]])) (Out(int , [[a;a]]))
17   (Worker([2;2]))
18   (BodyIn ([[Ranf(1,(-1,1));Ranf(2,(-1,1))]]))
```

```
19    (BodyOut([[One(1);One(2)]]))
20    (seq(
21     fun _ x->
22      let a  = (List.nth x 0)  in
23      let c = ref 0 in (* contatore delle celulle vicine in
          vita *)
24      for i = 0 to 2 do
25       for j = 0 to 2 do
26        if not (i=1 && j=1) then (* la cellula centrale non
            conta essendo la cellula controllata dal
           processore virtuale *)
27         c:= !c + if((Genarray.get a [|i;j|]) = 1) then 1
            else 0 ;
28       done;
29      done;
30      if (!c = 2) then [1] else [0]; (* se i vicini sono 2
         vive, altrimenti muore *)
31    )  )
32     (EndMapWith (Times steps)) (* ripetizione per steps
         volte *)
33  ) ;;
34
35 pardo(function ()->
36  let playground = create a a in
37  let st = P3lstream.of_list [[playground]] in
38  let r  = lifestep st in
39  P3lstream.iter (fun l ->List.iter (printBigaxb a a) l) r
40 );;
```

Code 47: L'applicazione game of life

Il codice eseguito dal generico processore virtuale è contenuto nello skeleton `seq` (linee 20-31): ogni processore virtuale riceve in input un bigaray contenente una sotto matrice `3x3` della matrice di input (contenente la cellula rappresentata dal processore virtuale e i suoi 8 vicini) e conta il numero di vicini vivi. (linee 24-29). Se i vicini vivi sono due (linea 30) restituisce `1` (cellula viva), altrimenti `0` (cellula morta). Da notare l'utilizzo del costruttore di tipo `Times` per indicare l'iterazione del gioco per `steps` volte (linea 32). Nella funzione `pardo` vediamo la creazione della matrice di cellule iniziale (linea 36) tramite la funzione `create`, la generazione uno stream contenente il bigarray appena creato (linea 37) e l'applicazione della parfun `lifestep` a tale stream (linea 38). Il risultato è ottenuto alla linea 39 e stampato semplicemente.

Capitolo 7

Risultati sperimentali

Come discusso in 2.4 i sistemi a skeleton sono valutabili da punti di vista diversi: da un lato il raggiungimento delle prestazioni massime, quanto più vicine all'ideale, dall'altro la semplicità d'uso, l'alto livello di astrazione e il riuso di codice preesistente. Il nostro sistema è valutabile dal secondo punto di vista, riuscendo comunque ad avere risultati migliori rispetto ad un codice sequenziale.

In questo capitolo vediamo i risultati sperimentali relativi alla misura di soli due parametri tra quelli visti in 1.5, ovvero il tempo di completamento e la scalabilità, di un'applicazione map in assenza di stream. Infatti siamo interessati a valutare dopo quanto tempo l'utente può ottenere i risultati finali di un'applicazione data parallel pura e di quanto l'esecuzione parallela migliora le prestazioni rispetto all'applicazione sequenziale corrispondente. Ci interessa soprattutto capire qual è la grana del calcolo oltre la quale si riescono a bilanciare i costi di gestione dello skeleton e a ottenere prestazioni che migliorano quelle del corrispettivo calcolo sequenziale. Per misurare la grana abbiamo utilizzato una funzione sequenziale `f`, misurato il suo tempo di completamento, e ad ogni test ripetuta un numero variabile `n` di volte su ogni input; il numero `n` di volte che questa funzione sintetica è ripetuta per noi rappresenta la grana di calcolo. In altri casi si può intendere per grana il rapporto tra il tempo di calcolo e la latenza di comunicazione del dato su cui si esegue il calcolo. In altre parole vogliamo trovare il tempo di completamento

della funzione sequenziale applicata dal processore virtuale oltre il quale riusciamo a trarre vantaggi dall'esecuzione parallela col nostro nuovo skeleton.

Per far questo abbiamo usato un'applicazione sequenziale sintetica che in calcola la funzione `f` (con tempo di completamento noto), applicandola `n` volte ad ogni elemento di una matrice di input `120x120`, variando per ogni test il valore di `n`. Abbiamo quindi confrontato questo tempo di completamento con il tempo di completamento della corrispondente parallelizzazione tramite lo skeleton `map`. Nella versione parallela il calcolo sequenziale eseguito da ciascun processore virtuale è rappresentato dal singolo ciclo di ripetizioni dell'applicazione della funzione `f` che nella versione sequenziale è eseguito per ogni elemento della matrice. Per ogni esecuzione parallela di ciascun test abbiamo modificato il valore di `n` e per ogni valore di `n` abbiamo modificato l'ampiezza della matrice dei worker (quindi il grado di parallelismo), utilizzando tanti nodi fisici quanti sono i worker (più due per emettitore e collettore). Inoltre per testare come variano le prestazioni in funzione delle diverse distribuzioni dei dati di ingresso, abbiamo utilizzato `indexpattern` diversi, corrispondenti a scatter, stencil e multicast. I test sono stati eseguiti sul cluster del Polo Fibonacci dell'Università di Pisa costituito da PC con processori AMD Athlon(tm) XP 2600+ e con sistema operativo Linux.

L'applicazione sequenziale da parallelizzare ha questa struttura

```
1 let srcA = Genarray.create float32 c_layout [|a;a|];;
2 for i = 0 to (a - 1) do (* riempimento del bigarray di
     input con numeri casuali *)
3  for j = 0 to (b - 1) do
4   Genarray.set srcA [|i;j|] (Random.float 6.28)
5  done;
6 done;;
7 let out = Genarray.create float32 c_layout [|a;a|];;
8 let count = ref 0.0 ;; (* il generico valore di ogni
     elemento del bigarray di output*)
9 let n = ...;; (* la grana del calcolo*)
```

```
10 for j= 0 to (a - 1) do
11  for k= 0 to (b -1) do
12   for i = 1 to n do (* iterazione di f per n volte *)
13   let res  = (f(Genarray.get srcA [|j;k|])) in
14 (* per evitare sgradite ottimizzazioni del compilatore
     usiamo il risultato e sommiamolo ad un numero casuale
     *)
15   count:= !count +. Random.float (float_of_int i) +. res;
16   done;
17  Genarray.set out [|j;k|] !count;
18  done;
19 done;;
```

Il corrispondente skeleton map ha la seguente struttura

```
1 let worker = Worker (...);; (* la matrice di worker *)
2 let n = ...;; (* la grana del calcolo *)
3 let pattern = ...;; (* la tipologia di distribuzione *)
4 let mapskel = map  Bounded (In ([[a;b]])) (Out(float32,
    [[a;b]]))
5  (worker)
6  (BodyIn ([pattern])) (BodyOut([[One(1);One(2)]]))
7  (seq(
8    fun _ x->
9     let a1 = (List.nth x 0) in (* il bigarray di input *)
10     let count = ref 0.0 in
11     for i = 0 to n do (* ripetizione di f per n volte per
          ottenere ogni elemento del bigarray di output *)
12      count:= !count +. Random.float (float_of_int i) +. (
          f(Genarray.get a1 [|0|]))
13     done;
14     [!count] (* il valore di output del processore
```

```
        virtuale *)
15   )  )
16   (EndMap)
```

Vediamo come variano i tempi di completamento in funzione della grana `n` del calcolo della nostra applicazione nei diversi casi di distribuzione dei dati, modificando il parametro `pattern`, al variare del grado di parallelismo, modificando il parametro `worker`.

In Fig. 21 vediamo il tempo di completamento in caso di scatter, ovvero `pattern = [One(1);One(2)]`; si nota come, a partire da grana di calcolo 4, all'aumentare della stessa e del grado di parallelismo il tempo di completamento diminuisca sempre di più rispetto al caso sequenziale.

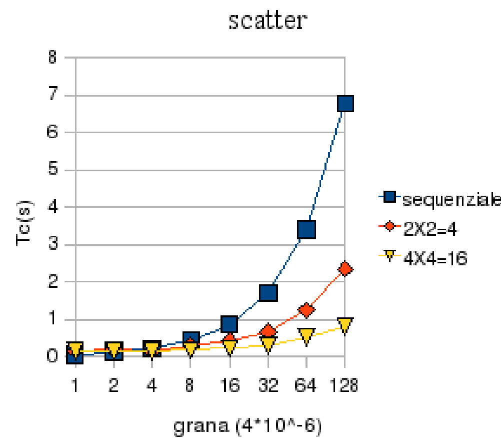

Fig. 21: Tempo di completamento in funzione della grana di calcolo al variare del grado di parallelismo in caso di scatter

L'andamento del tempo di completamento in caso di scatter è rispecchiato dall'andamento della scalabilità, in Fig. 22 vediamo l'andamento della scalabilità in caso di matrice di worker `4x4`: all'aumentare della grana la scalabilità aumenta. Questo significa che all'aumentare del tempo di calcolo del singolo processore virtuale i vantaggi della parallelizzazione rispetto al caso sequenziale aumentano, pur rimanendo lontani dalla scalabilità ideale (nel caso in esame 16).

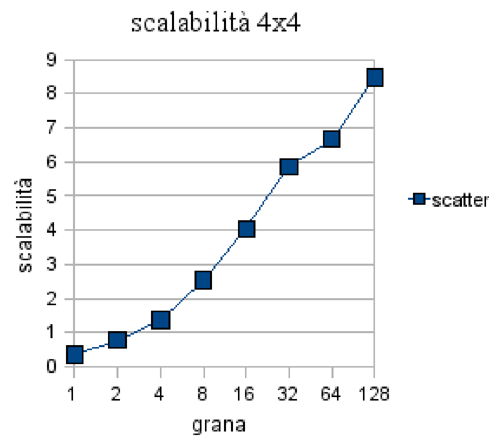

Fig. 22: Scalabilità in caso di scatter su una matrice di worker 4x4 in funzione della grana

In Fig. 23 vediamo il tempo di completamento in caso di stencil fisso (h = 2), ovvero `pattern = [Ranf(1,(-2,2));Ranf(2,(-2,2))]`; vediamo che l'andamento è simile al caso di scatter ma la grana a partire dalla quale si hanno miglioramenti rispetto al caso sequenziale è maggiore (16) ed in generale rispetto al caso scatter è maggiore il tempo di completamento.

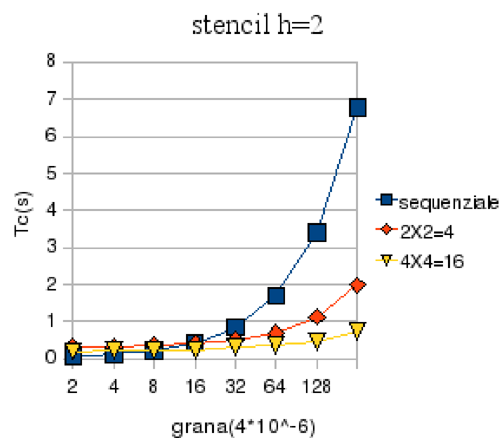

Fig. 23: Tempo di completamento in funzione della grana di calcolo al variare del grado di parallelismo in caso di stencil fisso con h=2

In Fig. 24 vediamo l'andamento della scalabilità in caso di stencil: anche in questo caso vediamo che all'aumentare della grana la scalabilità aumenta, raggiungendo valori minori rispetto al caso di scatter.

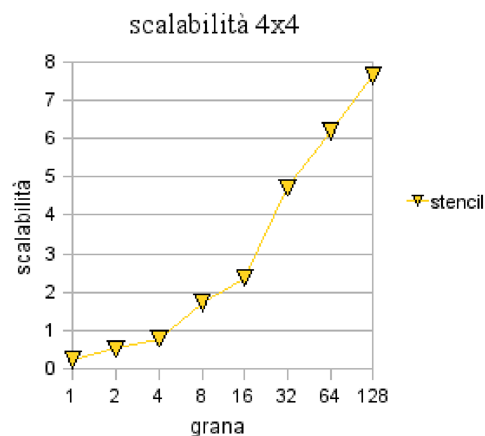

Fig. 24: Scalabilità dello stencil su una matrice di worker 4x4 in funzione della grana

In Fig. 25 vediamo il tempo di completamento in caso di multicast, ovvero `pattern = [One(1); All]`; anche qui si nota la stessa tendenza vista nei casi precedenti ed un ulteriore aumento della soglia della grana a partire dalla quale si hanno miglioramenti rispetto al caso sequenziale. Questi peggioramenti rispetto al caso scatter sono dovuti alle copie di dati per l'estrazione delle parti di dati su cui applicare la funzione sequenziale; tali overhead possono essere superati introducendo una cache delle partizioni di dati già estratte.

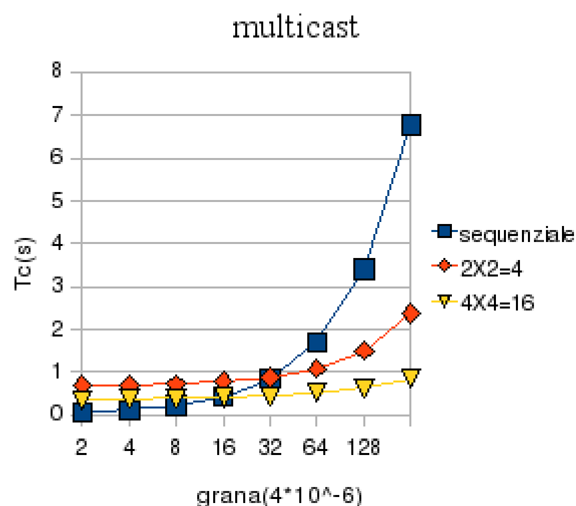

Fig. 25: Tempo di completamento in funzione della grana di calcolo al variare del grado di parallelismo in caso di multicast

In Fig. 26 vediamo l'andamento della scalabilità in caso di multicast: anche in questo caso vediamo che all'aumentare della grana la scalabilità aumenta, raggiungendo valori minori rispetto al caso di stencil.

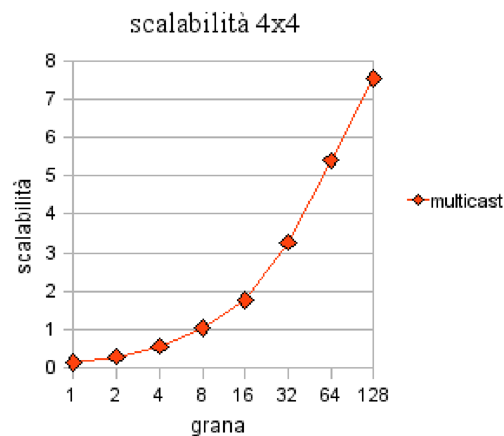

Fig. 26: Scalabilità in caso di multicast su una matrice di worker 4x4 in funzione della grana

In Fig. 27 vediamo una sintesi di quanto visto finora, diamo una visione di insieme delle curve che rappresentano le scalabilità riscontrate nei diversi casi di distribuzione: la scalabilità maggiore si ha in caso di scatter, mentre peggiora lievemente con lo stencil e diminuisce ulteriormente in caso di multicast. Questo chiaramente corrisponde alla degradazione del tempo di completamento dello stencil e della multicast rispetto alla scatter; in tutti i casi all'aumentare della grana si hanno comunque miglioramenti rispetto al caso sequenziale.

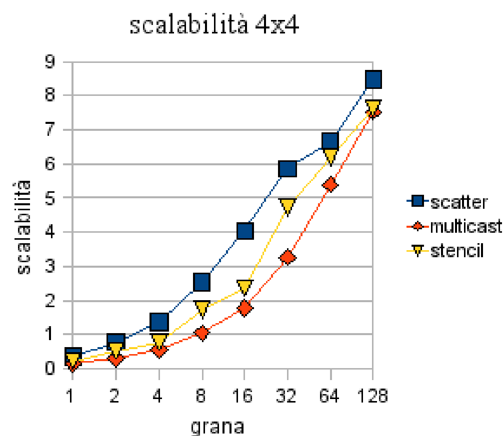

Fig. 27: Scalabilità in funzione della grana nei casi di scatter, stencil e multicast

Da quanto esposto finora risulta che il nuovo skeleton è adatto ai casi in cui la funzione sequenziale eseguita da ogni processore virtuale abbia una grana grossa, che sia in grado di bilanciare le latenze di comunicazione delle partizioni di dati e i costi per la creazione di strutture dati intermedie.

In generale, come abbiamo visto nel corso della tesi, tutti gli skeleton contenuti nella libreria *OcamlP3l* offrono all'utente la possibilità di esprimere ad alto livello un programma parallelo in modo molto semplice. Molto importate è la possibilità di riusare parti di codice sequenziale preesistente al fine di creare un'applicazione parallela costituita da queste parti di codice coordinate attraverso la struttura degli skeleton.

Dal punto di vista delle prestazioni, il comportamento del sistema rispecchia quanto visto in generale in 2.4. Gli skeleton task parallel offerti dal sistema danno buoni risultati in caso di grana di calcolo medio-fine, mentre quelli data parallel, come abbiamo appena visto, sono adatti a calcolo a grana grossa. Sicuramente, rispetto ad altri sistemi più orientati al raggiungimento di alte prestazioni, i valori dei parametri visti sopra, tempo di completamento e scalabilità, sono maggiori; d'altra parte questi sistemi abbassano notevolmente il grado di astrazione e la semplicità d'uso rispetto al nostro.

Capitolo 8

Conclusioni e sviluppi futuri

In questa tesi abbiamo riorganizzato e debuggato gli skeleton data parallel di *OcamlP3l*, traformando la loro implementazione task parallel in una implementazione effettivamente data parallel, ed abbiamo introdotto nel sistema un nuovo skeleton per applicazioni data parallel basate su stencil; con il nuovo skeleton è possibile eseguire map e stencil fissi anche iterativi su matrici numeriche da 1 a 16 dimensioni.

Il nuovo skeleton rappresenta il corrispettivo in *OcamlP3l* del costrutto map del linguaggio per la programmazione strutturata P^3L visto in 2.3.1. Rispetto a P^3L offre in più il trattamento di matrici multidimensionali e la possibilità di esprimere contestualmente allo skeleton la condizione di terminazione, dipendente dai risultati globali o meno, in modo da poter evitare ad ogni iterazione la redistribuzione dei dati sui worker.

Rispetto ai sistemi passati in rassegna nel capitolo 2 abbiamo raggiunto una notevole potenza espressiva coniugata ad un alto livello d'astrazione.

Dal punto di vista delle prestazioni abbiamo verificato che si ottengono buoni risultati per grana grossa di calcolo pur rimanendo lontani dalle prestazioni ideali.

Il sistema è limitato dall'utilizzo dei bigarray, che possono contenere solo elementi atomici numerici; è possibile trattare soltanto matrici numeriche e usare le sole distribuzioni viste in 6.1.2 senza poter organizzare i dati a priori in modo più consono ad alcune applicazioni. Inoltre un bigarray, per come è rappresentato in memoria ha

costi di accesso superiori rispetto a matrici rappresentate come array di array. Queste limitazioni possono essere superate solo riuscendo a sostituire l'implementazione dei bigarray con una più flessibile ed efficiente. Inoltre abbiamo visto che l'introduzione dei bigarray ha avuto impatto sui tipi `p3ltree` e `p3ltreeexp` a causa della presenza di nuovi parametri nei tipi `Genarray` ed `endmapwith`; ciò rende necessario l'uso di alcuni vincoli di tipo nel codice utente, rendendo di fatto il nuovo sistema non retrocompatibile.

Quest'ultimo problema e la verbosità del codice per esprimere lo skeleton map possono essere superati tramite l'uso di un preprocessore che restituisca il codice del tipo in Code 34 a seguito di un'analisi sintattica di un codice utente più semplice con sintassi simile a quella di P^3L. Il preprocessore CamlP4 potrebbe essere uno strumento adatto alla risoluzione del nostro problema, in quanto consente di estendere la sintassi OCaml attraverso interfacce predefinite.

Abbiamo visto nel capitolo 7 che in caso di multicast, essendo necessario copiare gli stessi dati più volte (nel worker), le prestazioni peggiorano; a tal fine è in corso di implementazione di una cache dei dati già copiati precedentemente al fine di evitare le copie multiple.

Un interessante sviluppo del progetto consiste nell'implementazione del modello dei costi sia per gli skeleton task parallel che che per quelli data parallel preesistenti sia per il nuovo skeleton map basato sui risultati teorici in [DCLP07], riguardanti i costi delle strategie di distribuzione. In tal modo è possibile, ad esempio in caso di `mapvector` o `reducevector`, scegliere automaticamente quali template implementativi usare tra quelli task parallel preesistenti e quelli data parallel visti nel capitolo 5 a seconda del costo che le due implementazioni hanno.

Bibliografia

[AEK08] Markus Alind, Mattias V. Eriksson, and Christoph W. Kessler. Blocklib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.

[AMP07] Danelutto M. Aldinucci M. and Dazzi P. Muskel: a skeleton library supporting skeleton set expandability. *Scalable Computing: Practice and Experience*, 8(4):325–341, 2007.

[BC05] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In *The International Conference on Computational Science (ICCS 2005) , Part II, LNCS 3515*, pages 764–771. Springer Verlag, 2005.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3l: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[BDPV99] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. Skie: a heterogeneous environment for hpc applications. *Parallel Comput.*, 25(13-14):1827–1852, 1999.

[CLLP06] Roberto Di Cosmo, Zheng Li, Xavier Leroy, and Susanna Pelagatti. Skeleta parallel programming with ocamlp3l 2.0. In *Parallel Processing Letters*, volume 1, pages 1–13, 2006.

[Col91] Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.

[Dan99] Marco Danelutto. Dynamic run time support for skeletons. Technical report, University of Pisa, 1999.

[DCLP07] Roberto Di Cosmo, Zheng Li, and Susanna Pelagatti. A calculus for parallel computations over multidimensional dense arrays. *Comput. Lang. Syst. Struct.*, 33(3-4):82–110, 2007.

[DCP03] R. Di Cosmo and S. Pelagatti. A Calculus for Dense Array Distributions. *Parallel Processing Letters*, 13(3):377–388, 2003.

[DDCL+07] M. Danelutto, R. Di Cosmo, X. Leroy, Z. Li, S. Pelagatti, and P. Weiss. *OcamlP3l 2.0: User Manual*, 2007.

[Kle76] L. Kleinrock. *Queueing Systems - Computer Applications*, volume 1. Wiley-Interscience, New York, New York, 1976.

[Ler07] Xavier Leroy. *The Objective Caml System release 3.11*, 2007.

[Li03] Zheng Li. Efficient implementation of map skeleton for the ocamlp3l system. Technical report, INRIA, 2003.

[LOmP05] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Pe na-marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, 2005.

[Man79] Benois B. Mandelbrot. Fractal aspects of the iteration of z $\rightarrow \lambda$ z (1-z) for complex λ and z. *Non-Linear Dynamics*, 1979.

[MCGH05] Anne Benoit Murray, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *In: 11th Intl Euro-Par: Parallel and Distributed Computing, vol. 3648 of LNCS, 761-770, Lisbona*, pages 761–770. Springer-Verlag, 2005.

[Met] MetaOcaml. http://www.metaocaml.com.

[MIEH06] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 13, New York, NY, USA, 2006. ACM.

[MKI+04] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A fusion-embedded skeleton library. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 644–653. Springer, 2004.

[paIR] The Cell project at IBM Research. http://www.research.ibm.com/cell.

[Pel93] Susanna Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Dipartimento di Informatica, 1993.

[SF08] Jocelyn Serot and Joel Falcou. Functional meta-programming for parallel skeletons. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 154–163, Berlin, Heidelberg, 2008. Springer-Verlag.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[Van02] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[Van09] Marco Vanneschi. *Architetture parallele e distribuite.* SEU, 2008-09.