Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

Topologia ad albero per PARMOD ASSIST

Candidato
Alessio Baldaccini

Relatore
prof. Marco Danelutto

Controrelatore
dott. Salvatore Ruggieri

Anno Accademico 2004-2005

Al mio amore Giulia

II

Ringraziamenti

Mi sembra giusto iniziare questi ringraziamenti citando la mia famiglia, che fin dall'inizio mi ha spronato ad intraprendere la strada universitaria; è grazie a loro se oggi ho raggiunto questo importante traguardo.

Voglio ringraziare i professori Marco Vanneschi, Domenico Laforenza e Laura Ricci che in questi anni mi hanno avvicinato sempre più al mondo delle architetture parallele e distribuite, verso il quale ho maturato un interesse crescente. Ma il ringraziamento più grande va senza dubbio al professor Marco Danelutto, per avermi dato l'opportunità di prendere parte ad un progetto di ricerca così importante, quale quello di ASSIST, e per il costante sostegno ricevuto nell'affrontare i problemi incontrati.

Un ringraziamento particolare va a Giulia, per l'aiuto indispensabile che mi ha dato in ogni momento e per aver sempre creduto in me e nelle mie capacità. Grazie di essermi stata così vicina e avermi sostenuto, soprattutto nei momenti più difficili.

Indice

CAPITOLO 1

Introduzione

ASSIST (*A Software development System based on Integrated Skeleton Technology*) [9] è un nuovo ambiente di programmazione orientato allo sviluppo di applicazioni parallele e distribuite ad alte prestazione. La sua realizzazione si basa sulla tecnologia della programmazione parallela strutturata e sugli skeletons [14, 15, 26]. Esempi di tale approccio sono `eSkel` [16] e `P3L` [19].

Uno dei principali obiettivi è quello di superare alcuni limiti posti dagli skeletons classici ed ottenere maggiore flessibilità e potenza espressiva, permettendo la definizione di nuove forme di parallelismo. Il nuovo costrutto di programmazione introdotto in ASSIST, il *modulo parallelo*, o *Parmod*, va incontro a tali esigenze. Rappresenta lo skeleton generico e permette di generalizzare ed estendere le caratteristiche di quelli classici, offrendo al programmatore la possibilità di definire forme di parallelismo specializzate, che possono essere trattate come skeletons di base nella composizione di programmi paralleli strutturati.

Le forme di parallelismo classiche, quali quelle data parallel o stream parallel, possono essere ridefinite tramite opportune specializzazioni del modulo parallelo.

Il Parmod costituisce una componente base nel modello di programmazione ASSIST ed ha un'interfaccia che gli permette l'interazione con gli altri moduli di una applicazione. I dati da elaborare vengono ricevuti dagli *streams di input* e i risultati sono trasmessi sugli *streams di output*. Per ogni stream di input può essere definita una *politica di distribuzione* che permette di trasmettere i dati ricevuti ai processori virtuali; mentre per ogni stream di output possiamo definire una *strategia di collezione* dei valori prodotti durante l'esecuzione.
La scelta del tipo di parallelismo da utilizzare per una determinata applicazione passa attraverso la definizione della *topologia*, che permette di identificare le entità computazionali (*processori virtuali*), che eseguono il calcolo. Ogni processore virtuale può interagire con gli altri tramite uno stato condiviso. Il tipo di topologia è legato allo *schema di naming* dei processori virtuali. Ad ognuno è associato un identificatore unico che può essere espresso parametricamente per definire varie forme di parallelismo. Tale schema è anche legato al modo con il quale lo stato viene distribuito, ma non influenza la struttura delle interazioni tra i processori virtuali.

Nella versione attuale di ASSIST possono essere utilizzate le seguenti topologie: *one*, *none* e *array*. Con la topologia *one* è possibile definire un Parmod con un solo processore virtuale, utile nei casi in cui si deve gestire il nondeterminismo dagli streams di input. La topologia *none* viene impiegata quando è necessario replicare unità di calcolo che eseguono una computazione puramente funzionale; in questo caso i processori virtuali sono completamente anonimi ed il loro naming non è significativo. La topologia *array* permette di definire un insieme di processori virtuali logicamente organizzati in base alla struttura dichiarata e di disporre di uno schema di naming da utilizzare per la definizione delle elaborazioni.

Il programmatore ha una vista ad alto livello del Parmod; tramite il *linguaggio di coordinamento* definisce la struttura logica dei processori virtuali e la gestione dei dati di input e di output. Il framework fornisce il supporto necessario per l'allocazione dei processori virtuali sui processori fisici e lo scambio dei dati. Esso è del tutto trasparente al programmatore.

Esaminiamo il frammento di codice dell'esempio 1 e analizziamo nel dettaglio il ruolo della topologia.

Esempio 1. Definizione di un semplice Parmod con topologia array.

```
1  parmod esempio1(input_stream long elementi[6]) {
2    topology array[i:6] vps;
3    attribute long stato[6] scatter stato[*j] onto vps[j];
4
5    input_section {
6      guardia: on, , elementi  {
7        distribution elementi[*h] scatter to stato[j];
8      }
9    }
10
11   virtual_processors {
12     elab elaborazione(in guardia) {
13       VP i = 0..2 { elabora1(in stato[i] out stato[i]); }
14       VP i = 3..5 { elabora2(in stato[i - 1] out stato[i]); }
15     }
16   }
17 }
```

La riga 1 definisce l'interfaccia del modulo parallelo; questo riceve da un unico stream di input un vettore di 6 elementi, i quali vengono successivamente distribuiti tra i processori virtuali in base alla strategia selezionata nella sezione di input (righe 5-8).
La riga 2 contiene la definizione della topologia; nel nostro caso il Parmod è costituito da un insieme di sei processori virtuali logicamente strutturati come un vettore (figura 1.1). Ogni processore virtuale viene identificano utilizzando il valore dell'indice di tale vettore (la variabile *i*).

Il programmatore utilizza tale rappresentazione per definire lo stato distribuito, tramite una regola di mapping (riga 3), e per selezionare i processori virtuali ai quali assegnare le computazioni. Il tipo di parallelismo utilizzato e la modalità di cooperazione vengono definiti in maniera indipendente dalla struttura dei processori virtuali. Nell'esempio la computazione avviene secondo uno schema data parallel, in cui ogni processore virtuale esegue una funzione su un certo sottoinsieme di dati. Vengono definiti due dis-

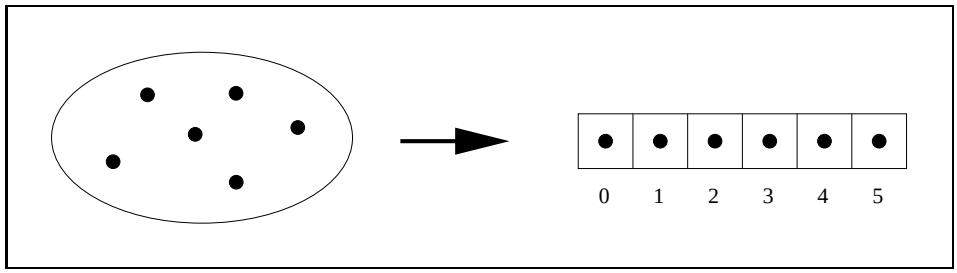


Figura 1.1: Topologia array: i processori virtuali vengono rappresentati come un vettore di 6 elementi.

tinti sottoinsiemi di calcolo, ognuno dei quali utilizza un diverso *pattern di comunicazione*: nel primo caso (riga 13) viene utilizzato il valore associato al processore virtuale selezionato, mentre nel secondo (riga 15) si utilizza il dato di un "vicino".
In sostanza, la topologia viene impiegata per definire la struttura concettuale delle entità computazionali, ovvero uno schema logico ad alto livello legato a come queste sono organizzate, che il programmatore utilizza per definire il comportamento del Parmod.

L'importanza di questo nuovo costrutto ha reso interessante lo studio di funzionalità e caratteristiche che potessero estenderne la generalità. In particolare, l'obiettivo della tesi è stato quello di integrare al suo interno il supporto per computazioni logicamente strutturate ad albero ed introdurre un nuovo tipo di topologia da affiancare a quelle già esistenti.
In letteratura abbiamo molti esempi di algoritmi e paradigmi che sfruttano o si basano su strutture ad albero; tra questi ricordiamo:

- *Algoritmi di ricerca e selezione*: sfruttano le caratteristiche degli alberi binari di ricerca e degli RB-alberi per effettuare operazioni di ricerca e selezione su un insieme di dati.

- *Algoritmi divide et impera*: per risolvere un determinato problema, l'algoritmo lo scompone in un insieme di sottoproblemi più piccoli, ed una volta risolti questi, i risultati ottenuti vengono ricombinati per ottenere la soluzione iniziale. L'esecuzione avviene in maniera ricorsiva sul sottoproblema da risolvere e si può utilizzare una rappresen-

tazione ad albero per l'intero processo. Esempi di questo paradigma sono gli algoritmi mergesort e quicksort.

- *Algoritmi branch & bound*: questo approccio viene impiegato per risolvere problemi di ottimizzazione. Si crea un albero delle scelte utilizzando: (a) una regola di branch che determina la scelta successiva; (b) una funzione di bound che fornisce un limite alla funzione costo per le scelte generate; (c) un metodo di visita dell'albero.

L'utilizzo pratico che viene fatto di tali algoritmi rende significativo lo studio di nuovi metodi per parallelizzarne ed ottimizzarne l'esecuzione. Allo stadio attuale dello sviluppo, ASSIST non fornisce un supporto nativo per computazioni strutturate ad albero; queste devono essere emulate tramite la topologia array.
Quindi, introdurre un nuova topologia per la gestione di tali applicazioni permette di aumentare la potenza espressiva del linguaggio di coordinamento e di estendere il supporto di ASSIST ad applicazioni multidisciplinari.

Lo sviluppo delle nuove funzionalità è stato suddiviso in due fasi. La prima si è concentrata sullo studio del modello di programmazione di ASSIST e del modulo parallelo. Sono state analizzate le sezioni che ne costituiscono la struttura e il modo in cui specializzarle per ottenere il tipo di parallelismo desiderato. Inoltre, è stato preso in esame un Parmod con topologia array, la più simile per caratteristiche e funzionamento a quella ad albero, per valutare nel dettaglio il tipo di stato che può essere associato ai processori virtuali, la distribuzione dei dati di input e la collezione dei risultati prodotti. L'analisi di questi fattori ha permesso di definire i requisiti del progetto.
Nella seconda fase è stata implementata la topologia *tree*. Il lavoro si è concentrato completamente sul compilatore; per ogni fase è stato studiato il codice sorgente, modificandolo ed adattandolo in base alle esigenze. Le parti che hanno richiesto maggior impegno sono state quella relativa ai controlli semantici e quella relativa alla generazione del codice C++ per gli eseguibili.

La valutazione della prestazioni del Parmod modificato è stata effettuata su diversi aspetti:

- *Controllo del parallelismo*. Questo tipo di test è stato eseguito per ogni programma utilizzato durante le prove. Ha permesso di valutare il corretto funzionamento del Parmod e della gestione dei processori virtuali.

- *Tempo di completamento, scalabilità, efficienza*. Con questi tests sono state valutate le performance reali della nuova topologia. Abbiamo utilizzato una semplice applicazione di riferimento, confrontando il tempo di completamento ottenuto rispetto a quello ideale per una serie di esecuzioni al variare della grana computazionale. Questo è servito per valutare la scalabilità e l'efficienza dell'implementazione.

- *Parmod array contro Parmod albero*. Sono state confrontate le prestazioni, in termini di tempo di completamento, per la stessa applicazione nel caso di topologia array e ad albero. I risultati hanno permesso di valutare la potenza espressiva della nuova versione del Parmod nei confronti di quella originale e la differenza di performance che si ottiene.

- *Parmod modificato contro Parmod originale*. Questa serie di prove ha permesso di stimare l'overhead introdotto dall'implementazione all'interno del Parmod. È stato valutato il tempo di completamento per la stessa applicazione, realizzata utilizzando la topologia array, compilando i sorgenti con la versione originale del compilatore e con quella modificata.

I risultati ottenuti hanno mostrato un comportamento molto buono, dal punto di vista della scalabilità e dell'efficienza, al variare della grana computazionale dell'applicazione da parte della nuova topologia. Nel complesso, a parità di potenza espressiva, le prestazioni sono state paragonabili a quelle offerte dalla topologia array. Nel momento in cui si sfrutta una maggiore espressività, la topologia ad albero risulta migliore di quella array.

La parte restante della tesi è stata organizzata come segue:

- **Capitolo 2**. Viene descritto l'ambiente di sviluppo di ASSIST, le sue caratteristiche principali e si analizzano in dettaglio le sezioni del modulo parallelo. Dopodiché l'attenzione si sposta sul compilatore; vengono descritte le varie fasi e come un programma sorgente viene tradotto nel codice C++ dell'applicazione.

- **Capitolo 3**. Viene fatta una presentazione ad alto livello del progetto logico della nuova topologia. Partendo dal linguaggio di coordinamento si descrivono i costrutti da introdurre e le principali scelte fatte.

- **Capitolo 4**. Questa parte è dedicata all'implementazione. È suddivisa in tre sezioni, nelle quali vengono prese in esame le fasi del compilatore e per ognuna di esse si descrivono in dettaglio le modifiche e le aggiunte apportate.

- **Capitolo 5**. Vengono descritte le prove effettuate. Per ogni programma utilizzato si analizza il suo funzionamento e si mostrano i risultati sperimentali ottenuti.

- **Capitolo 6**. Si fa un'analisi dell'intero lavoro svolto durante la tesi e si descrivono alcuni sviluppi futuri legati ad aspetti non trattati durante l'implementazione.

- **Appendici**. Contengono il codice delle principali classi realizzate durante l'implementazione ed i sorgenti dei programmi utilizzati per le prove.

CAPITOLO 2

Stato dell'arte

In questo capitolo viene presentato l'ambiente di sviluppo di ASSIST, le sue caratteristiche e gli strumenti messi a disposizione del programmatore per sviluppare applicazioni parallele strutturate.
Dopo una prima parte dedicata ad una panoramica generale di ASSIST e delle sue funzionalità, si passa ad analizzare il modulo parallelo, prestando particolare attenzione agli aspetti della sua struttura che possono essere modellati per specializzare l'esecuzione e definire nuove forme di parallelismo. Infine, presentiamo il compilatore; si descrivono le fasi che lo compongono e come queste intervengono nella creazione del codice eseguibile di una applicazione.

2.1 Il mondo di ASSIST

ASSIST [4, 28, 29] fornisce un ambiente di programmazione completo, espressivo e flessibile. Tramite il linguaggio di coordinamento [12, 13] (ASSIST-CL) è possibile definire i programmi paralleli strutturati; il compilatore (`astCC`) permette di compilare i sorgenti ed ottenere gli eseguibili

di una applicazione; i tools di supporto a run-time, (`assistrun` e `clam`) facilitano le operazioni di esecuzione.
Il framework è sviluppato a livelli ed offre un alto grado di performance e portabilità per diverse piattaforme e applicazioni [2, 6]. Ad esso è assegnato il compito di fornire il supporto necessario per la gestione delle applicazioni (comunicazioni, mapping dei processori virtuali sui processori reali, sincronizzazione).

Grazie alla possibilità di utilizzare diversi linguaggi ospiti (C, C++ e Fortran) per definire le computazioni, ASSIST favorisce il *riuso del codice*. Inoltre, è possibile esportare un programma ASSIST come oggetto CORBA [18, 24] o riferire oggetti CORBA all'interno del codice, fornendo il supporto per *l'interoperabilità* delle applicazioni [22].
Nella versione attuale è in corso di sviluppo il supporto per le griglie computazionali [7, 11], nell'ambito del progetto nazionale di ricerca *Grid.it* [21]. In particolare, è in atto uno studio relativo ai vari modelli a componenti [5, 8, 17], tra cui CCM [23] ed i Web Services [30], per valutare le possibilità di integrazione di tale approccio all'interno dell'ambiente di sviluppo di ASSIST.

2.1.1 Struttura di un programma

Una applicazione ASSIST viene definita come un grafo generico, in cui i nodi costituiscono le componenti e gli archi gli streams, cioè i canali di comunicazione. Una componente può essere un modulo sequenziale o parallelo. I moduli interagiscono scambiandosi informazioni tramite gli streams. Un programma, quindi, viene definito come composizione di moduli, ed è possibile riutilizzare la sua rappresentazione come componente base per definire applicazioni più complesse.

2.2 Il Parmod

Il Parmod rappresenta la soluzione adottata in ASSIST per risolvere i problemi di potenza espressiva e flessibilità degli skeletons classici. Le

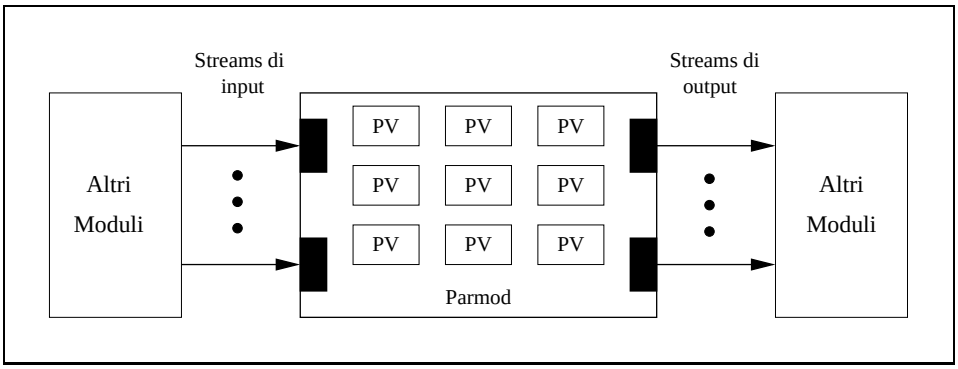


Figura 2.1: Schema grafico del Parmod.

caratteristiche di questo nuovo costrutto permettono di specializzare e modellare il suo funzionamento in base alle richieste di una applicazione. Può, quindi, essere considerato lo skeleton generico, tramite il quale definire forme di parallelismo ad-hoc.
La figura 2.1 mostra lo schema generale del Parmod. Il modulo parallelo possiede un'interfaccia di ingresso/uscita che gli permette di comunicare con gli altri moduli dell'applicazione. La computazione viene eseguita da un insieme di *processori virtuali*, i quali interagiscono e cooperano tramite uno *stato* interno. Un processore virtuale rappresenta un'entità di computazione generica che esegue una sequenza di *tasks* in maniera sequenziale. Un *task* corrisponde ad una elaborazione sequenziale dell'algoritmo da parallelizzare. È possibile definire insiemi disgiunti di processori virtuali ai quali associare elaborazioni distinte.
Un'insieme di sequenze di tasks correlate logicamente tra loro ed associate ad un'insieme di processori virtuali costituisce una *Elaborazione Parallela*. È possibile definire più elaborazioni, la cui attivazione è guidata dalla ricezione dei valori sugli streams di input ad esse associati.

2.2.1 Struttura interna

La dichiarazione della struttura interna del Parmod può essere divisa in quattro parti:

1. *Dichiarazioni*. Contiene la definizione del tipo di topologia del Parmod, dello stato interno, delle variabili di controllo e degli streams interni.

2. *Sezione di input*. Permette di modellare il comportamento del Parmod nel momento in cui arrivano i dati dagli streams di input. È possibile definire il tipo di distribuzione dei dati per i processori virtuali e specificare particolari funzioni di pre-elaborazione.

3. *Sezione processori virtuali*. Vengono definite le elaborazioni parallele.

4. *Sezione di output*. In questa sezione possiamo specificare il modo in cui collezionare i risultati prodotti durante l'esecuzione e se eseguire funzioni di post-elaborazione prima di inviare i valori sugli streams di output.

La specializzazione di ogni sezione permette di definire il comportamento del modulo parallelo.

Dichiarazioni

Il primo passo per la definizione del Parmod è quello di dichiarare il tipo di topologia. Essa rappresenta lo schema di *naming* dei processori virtuali e non è legata al modo in cui questi effettivamente cooperano o comunicano. Ogni processore virtuale ha un identificatore unico, che può essere utilizzato per definire particolari forme di parallelismo.
Nella versione attuale di ASSIST esistono tre topologie:

- **one**: il Parmod è costituito da un solo processore virtuale. Questa configurazione viene utilizzata per gestire il nondeterminismo sugli streams di input e specializzare l'esecuzione in base ai dati ricevuti.

- **none**: in questo caso il nome dei processori virtuali non è importante ai fini della computazione; questi sono anonimi e non in relazione tra loro. Tale configurazione viene utilizzata per definire elaborazioni in cui è necessario replicare l'esecuzione di una funzione pura senza dipendenza tra i dati (come in un farm).

- **array**: i processori virtuali sono logicamente strutturati come un array multidimensionale e possono essere identificati tramite indici. Tale topologia viene impiegata se lo stato interno deve essere condiviso tra i processori virtuali oppure se questo deve essere mantenuto indipendentemente dall'attivazione del modulo.

Una volta definita la topologia, è possibile dichiarare lo stato interno, le variabili di controllo e gli streams interni.

Lo stato interno del Parmod è rappresentato da *attributi* logicamente *partizionati* e/o *replicati* tra i processori virtuali. Nel primo caso ogni processore virtuale dispone di una parte dello stato e l'accesso o la modifica dei dati è permesso in base alla *owner compute rule*: il processore virtuale proprietario del dato può accedervi in lettura e scrittura, tutti gli altri solo in lettura. Nel secondo ogni processore virtuale dispone di una copia dell'attributo ed è l'unico che può accedervi.
Il tipo di topologia determina il tipo di stato del Parmod. Nel caso di topologia one si potranno definire solo attributi singoli per il processore virtuale, che non sono né partizionati né replicati; un Parmod con topologia none può disporre solo di attributi replicati; per uno con topologia array è possibile definire sia attributi replicati che partizionati.

Le variabili di controllo rappresentano *variabili di stato* che vengono utilizzate per gestire le comunicazioni dagli streams di input e verso gli streams di output e sono condivise tra la sezione di ingresso e quella di uscita del Parmod. Grazie ad esse è possibile definire condizioni di terminazione aggiuntive o creare loop impliciti su streams, cioè terminare l'esecuzione se si verifica una determinata condizione oppure attivare un modulo per un certo numero di volte.

Gli streams interni vengono utilizzati nel caso in cui il tipo dello stream di output del Parmod non coincida con quello del dato prodotto durante l'esecuzione e sia necessario collezionare ed aggregare tutti i valori prima di poter comunicare il risultato.

Sezione di input

In questa parte è possibile definire la procedura di inizializzazione dello stato e delle variabili di controllo, dichiarare le guardie di ingresso e le politiche di distribuzione dei dati di input.

L'inizializzazione viene effettuata nel momento in cui il Parmod è istanziato. È necessario selezionare i processori virtuali che devono eseguire tale operazione, utilizzando lo stesso schema di indicizzazione con il quale vengono specificate le elaborazioni parallele.

Le guardie sono definite da un identificatore unico che riferisce gli streams controllati, un valore di priorità, una condizione di abilitazione/disabilitazione ed una espressione sugli streams di input. In questo modo possiamo modellare un comportamento sia di tipo *data-flow* che *nondeterministico*. Nel primo caso il Parmod attende di ricevere i valori da tutti gli streams di input, mentre nell'altro viene selezionato un sottoinsieme di streams ad ogni attivazione.
Ad ogni stream di input è associata una *strategia di distribuzione* indipendente, che permette di trasmettere i dati ricevuti ai processori virtuali oppure di selezionare una variabile di stato nella quale salvare i dati. Sono permesse le seguenti strategie:

- **scatter**: il dato viene distribuito ai processori virtuali in base alla regola di mapping utilizzata. È anche possibile specificare, al posto dei processori virtuali, un attributo partizionato e distribuire il dato su di esso.

- **broadcast**: il dato è inviato a tutti i processori virtuali.

- **on_demand**: il dato viene inviato al primo processore virtuale libero.

- **scheduled**: il dato viene inviato al processore virtuale selezionato tramite indici.

Sezione processori virtuali

Una volta specificata la topologia e lo stato interno ed aver definito come distribuire i dati di input, la fase successiva consiste nella definizione delle elaborazioni parallele, nelle quali si associano agli streams di input le computazioni da eseguire. Ogni elaborazione è rappresentata da un nome unico che la identifica, una lista di guardie, utilizzata per attivare l'esecuzione una volta ricevuti i dati, e una lista di stream di output, sui quali inviare i risultati.

Nel corpo dell'elaborazione si definisce la computazione da eseguire tramite una selezione dei processori virtuali e dei tasks ad essi associati. In generale, è possibile definire sottoinsiemi disgiunti di processori virtuali che eseguono differenti sequenze di tasks all'interno di una determinata elaborazione.

Oltre a queste funzionalità, se necessario, si può parallelizzare la sequenza di tasks assegnati ai processori virtuali utilizzando i costrutti iterativi *for* e *while*, che permettono di definire computazioni di tipo data-parallel.

Sezione di output

In questa sezione vengono definite le strategie per collezionare i valori prodotti dai processori virtuali. Quelle messe a disposizione da ASSIST sono le seguenti:

- **from ANY**: si seleziona in maniera nondeterministica uno dei processori virtuali.

- **from ALL**: si raccolgono i valori da tutti i processori virtuali e si crea il dato aggregato.

Dopo aver recuperato i dati, questi possono essere rielaborati (*post-elaborazione*) prima di inviarli sugli streams di output.

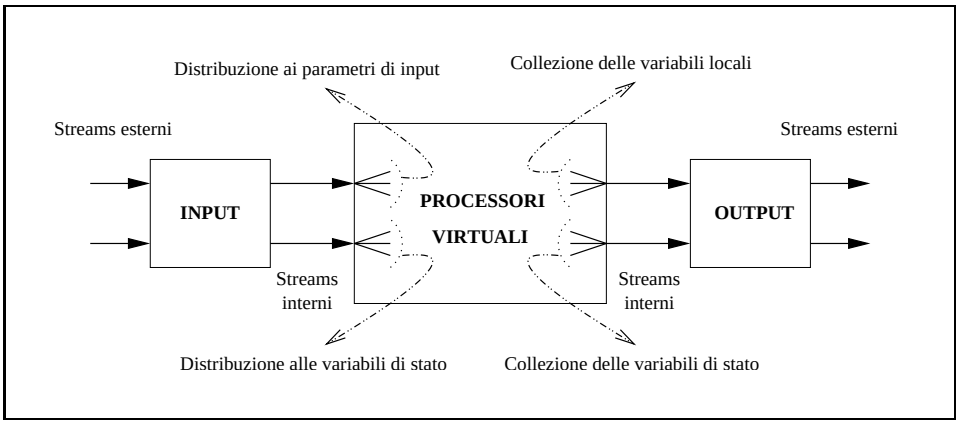


Figura 2.2: Semantica delle sezioni del Parmod.

2.2.2 Semantica delle sezioni

Le tre sezioni del Parmod che abbiamo descritto implementano le funzionalità del nuovo costrutto e cooperano seguendo uno schema di tipo pipeline (figura 2.2). Questa caratteristica permette di poter sovrapporre più attività ed è molto importante ai fini della performance e della scalabilità. È possibile, così, mascherare il tempo dovuto ad operazioni di pre/post-elaborazione dei dati oppure mascherare il tempo di comunicazione rispetto al tempo di calcolo dei processori virtuali.
Esistono streams interni tra la sezione di input e quella dei processori virtuali e tra questa e quella di output, utilizzati per lo scambio dei valori di stato e dei parametri di input/output.
Il supporto a run-time gestisce le comunicazioni ed i problemi di consistenza di tali streams, in completa trasparenza per il programmatore.

2.3 Il compilatore

In questa sezione analizziamo il compilatore ASSIST (`astCC`) tramite il quale un programma sorgente viene tradotto in un insieme di codici oggetto eseguibili.
Il compilatore [3, 4] è stato progettato in modo tale da permettere la generazione efficiente del codice; in ogni fase si possono prendere scelte in-

dipendenti legate al tipo di codice da generare. L'utilizzo di varie tecniche di sviluppo software, tra cui quella dei *design patterns* [20], permette di isolare ogni fase del compilatore in modo tale che modifiche in una sua parte non influenzino l'intero processo e non richiedano cambiamenti nelle altre. Possiamo dividere l'esecuzione del compilatore in tre fasi:

1. *Front-end*: si esegue l'analisi sintattica del programma sorgente e si crea una rappresentazione interna della sua struttura. Dopodiché vengono eseguiti i controlli semantici per determinare la correttezza dei costrutti utilizzati.

2. *Middle-end*: la rappresentazione interna viene tradotta nel codice intermedio, il *task-code*, una specie di linguaggio assembly parallelo ad lato livello basato sui template.

3. *Back-end*: il task-code viene ulteriormente tradotto in un insieme di classi C++, dalle quali si ottiene il codice eseguibile dell'applicazione.

Il risultato della compilazione produce:

- Un insieme di codici oggetto necessari per eseguire i moduli applicativi e per fornire il supporto a run-time. Ogni modulo può essere compilato in diverse versioni, ognuna per una diversa piattaforma.

- I makefiles necessari per la compilazione di tale oggetti.

- Un file XML di configurazione, contenente le informazioni sui moduli per il coordinamento a run-time.

I dati contenuti nel file di configurazione vengono utilizzati da un processo CLAM (`Coordination Language Abstract Machine`) master, che abilita il caricamento e l'esecuzione del codice oggetto da parte di processi CLAM slave presenti sulle macchine target. Inoltre, dal momento che la CLAM poggia sulla libreria ACE [1], sfruttandone le funzionalità, e che questa è disponibile per diverse piattaforme, è garantito un alto grado di portabilità dell'intero ambiente di sviluppo di ASSIST.

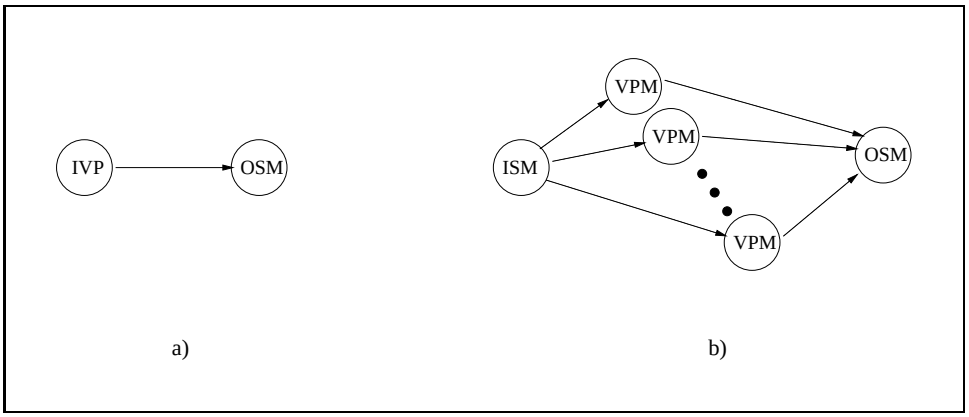


Figura 2.3: Rete di nodi task-code. a) Parmod one. b) Parmod none e array.

2.3.1 Il task-code

Il task-code viene utilizzato per descrivere la struttura interna dei moduli di una applicazione. Ogni modulo viene tradotto in una rete di nodi task-code e l'insieme di tutti i nodi rappresenta il grafo dell'applicazione.
Esistono due tipi di nodi task-code: quello sequenziale e quello parallelo. Quest'ultimo, in particolare, è in grado di esprimere parallelismo a livello di threads e può essere organizzato in gruppi omogenei di repliche secondo il paradigma SPMD.
Il Parmod viene rappresentato in due modi distinti. Nel caso di topologia one viene tradotto come una rete di due nodi sequenziali (figura 2.3a), mentre negli altri casi come una rete di tre o più nodi, di cui due sequenziali (ISM e OSM) e gli altri paralleli (figura 2.3b).
I nodi IVP, ISM e OSM contengono codice di supporto (per la gestione delle politiche di distribuzione nel caso del nodo ISM oppure delle strategie di collezione nel caso del nodo OSM). Il nodo VPM contiene codice utente (i tasks che devono essere eseguiti) e permette di gestire l'insieme dei processori virtuali.

Ogni nodo task-code viene successivamente tradotto in un insieme di classi ed oggetti C++ che definiscono un processo del supporto a run-time. La compilazione dei nodi permette di passare da una rete di nodi task-code ad una rete di processi, ognuno dei quali gestisce un processing element.

Ogni nodo task-code è fornito di due metodi principali che vengono chiamati in seguenza e determinano il suo comportamento: *init* e *run*. Il metodo *init* permette di effettuare le inizializzazioni ed ha il compito di configurare le parti dell'architettura del nodo che dipendono dalle impostazioni dinamiche del sistema; nel nostro caso questo aspetto riguarda il numero di VPM presenti nella rete di processi. In questa fase si identificano i rank dei nodi e si aprono i canali di comunicazione.
Il metodo *run* descrive il comportamento del nodo; la sua terminazione corrisponde alla terminazione dell'applicazione. L'esecuzione prosegue finché ci sono streams attivi o almeno una guardia è abilitata.

CAPITOLO 3

Topologia ad albero: progetto logico

Passiamo adesso a descrivere il progetto logico della nuova topologia. L'analisi delle funzionalità da integrare nel Parmod viene fatta descrivendo le modifiche al linguaggio di coordinamento e le scelte fatte.
L'obiettivo principale è ottenere un compromesso tra espressività e modifiche da apportare in fase di implementazione. Lo studio effettuato in questa fase si concentra sulla ricerca di caratteristiche che permettano di scrivere ed esprimere programmi significativi, limitando il numero di cambiamenti da effettuare sul linguaggio di coordinamento e sul compilatore.

Per la nostra analisi viene presa come punto di riferimento la topologia array ed i costrutti del linguaggio ad essa legati, perché le sue caratteristiche si avvicinano maggiormente a quelle della topologia ad albero.

La discussione si concentra sui seguenti punti: dichiarazione della topologia, identificazione dei processori virtuali, definizione dello stato interno, politiche di distribuzione e strategie di collezione dei dati. La parte finale del capitolo contiene un semplice esempio che mostra un utilizzo tipico delle nuove funzionalità.

3.1 Dichiarazione della topologia

La definizione del Parmod inizia con la scelta della topologia, quindi per prima cosa è necessario essere in grado di definire una topologia ad albero. ASSIST mette a disposizione la seguente sintassi:

```
topology array[indice:dim]{[indice:dim]} vps;
```

con cui si dichiara una topologia array multidimensionale, di nome vps. La sintassi prevede di poter definire un numero qualsiasi di dimensioni.

Questo costrutto può essere riutilizzato per supportare la dichiarazione della nuova topologia. Innanzitutto è necessario aggiungere al linguaggio di coordinamento la parola chiave `tree`, e successivamente associare alle varie dimensioni i dati che permettono di definire la struttura dell'albero. In particolare abbiamo bisogno di specificare il numero di *livelli* [1] e l'*arietà* dell'albero. Tramite questi due valori è possibile definire un albero k-ario completo di profondità arbitraria, ma comunque finita. Tale scelta è una conseguenza di un vincolo molto forte presente nell'attuale versione di ASSIST: la dimensione e la struttura dei dati, così come quella della topologia, non può cambiare dinamicamente durante l'esecuzione.
La sintassi, quindi, viene estesa nel seguente modo:

```
topology tree[livelli:valore][arieta:valore] vps;
```

Il frammento di code 1 definisce lo scheletro di un Parmod con topologia ad albero, che in seguito utilizzeremo per mostrare le varie caratteristiche introdotte.

Frammento di codice 1. Parmod ad albero: dichiarazione della topologia.

```
parmod esempio_albero(input_stream ... output_stream ...) {
  topology tree[l:3][a:2] vps;

  input_section { ... }
  virtual_processors { ... }
  output_section { ... }
}
```

[1] Il numero di livelli è inteso come profondità dell'albero, cioè non comprende la radice.

La dichiarazione della topologia utilizzata nell'esempio definisce un albero binario completo con quattro livelli.

Mettiamo in evidenza che la dichiarazione obbliga ad utilizzare due dimensioni ed è opposta a quella per la topologia array, nella quale non esiste un limite superiore al numero di dimensioni usate (basta usarne almeno una affinché la sintassi sia corretta). Sarà necessario, durante la fase di modifica del parser, effettuare un controllo aggiuntivo per stabile la correttezza della nuova dichiarazione.

3.2 Identificazione dei processori virtuali

Il metodo con il quale i processori virtuali vengono identificati è importante per la selezione dei sottoinsiemi ai quali assegnare una sequenza di tasks da eseguire e deve essere molto espressivo. Nel caso di un Parmod array ogni processore virtuale viene identificato utilizzando un indice per ogni dimensione ed è possibile selezionare particolari elementi lavorando sui valori associati a tali indici.

Lo stesso meccanismo può essere utilizzato per la topologia ad albero, impiegando due indici. Il primo permette di selezionare il livello al quale appartiene il processore virtuale, mentre il secondo identifica il processore virtuale tra tutti gli altri del suo livello. Tale soluzione risulta molto intuitiva e permette di identificare in maniera semplice un qualsiasi elemento dell'albero. Il vantaggio immediato è dato dal fatto che non è necessario modificare lo scanner ed il parser. Inoltre, permette di effettuare diversi tipi di selezione che possono essere utilizzati per la definizione delle elaborazioni parallele, sfruttando completamente le regole e la sintassi del linguaggio di coordinamento.
Nell'esempio contenuto nel frammento di codice 2 vengono selezionati tre diversi sottoinsiemi di processori virtuali, ai quali è assegnata una diversa elaborazione:

- Un unico elemento: la selezione viene definita specificando un valore per entrambi gli indici, in modo da identificare un unico proces-

Frammento di codice 2. Parmod ad albero: selezione dei processori virtuali.

```
1  parmod esempio_albero(input_stream ... output_stream ...) {
2    topology tree[l:3][a:2] vps;
3
4    input_section { ... }
5
6    virtual_processors {
7      elab elaborazione(...) {
8        VP l = 0, a = 0 { //Elaborazione 1 }
9
10       VP l = 1..2, a { //Elaborazione 2 }
11
12       VP l = 3, a = 0..8 { //Elaborazione 3 }
13     }
14   }
15
16   output_section { ... }
17 }
```

sore virtuale. Nel nostro esempio (riga 8) viene selezionata la radice dell'albero.

- Un insieme di livelli: questo tipo di selezione permette di specificare un sottoinsieme di elementi definiti utilizzando uno o più livelli dell'albero; al primo indice si assegna un range di valori, che identifica i livelli di interesse, mentre il secondo viene lasciato libero, per indicare che devono essere selezionati tutti i nodi dei livelli scelti. Nell'esempio fatto (riga 10) vengono selezionati gli elementi intermedi dell'albero.

- Un insieme di elementi di un livello: la selezione viene definita specificando un valore per il primo indice, che permette di identificare il livello di interesse, ed un range per il secondo, che determina il sottoinsieme dei nodi del livello che devono essere utilizzati. Nel nostro esempio (riga 12) la selezione avviene per tutte le foglie dell'albero.

3.3 Dichiarazione dello stato interno

Il passo successivo nella definizione del Parmod consiste nella dichiarazione degli attributi, che permettono di determinare lo stato associato ai

processori virtuali. È necessario, in primo luogo, valutare la possibilità di utilizzare i tipi di attributi disponibili nel linguaggio e successivamente decidere se introdurne altri per esigenze specifiche.

Gli attributi replicati permettono di definire variabili che vengono assegnate ai processori virtuali in modo che ognuno ne abbia una copia; il costrutto utilizzato è il seguente:

```
attribute tipo nome replicated;
```

Tali attributi possono essere impiegati anche per la topologia ad albero, dal momento che la dichiarazione non necessita di alcun dettaglio relativo ad essa.

Gli attributi partizionati permettono di definire uno stato che viene distribuito tra i processori virtuali in base alla regola scatter specificata. Ad esempio:

```
attribute long a[N] scatter a[*i] onto vps[i];
```

definisce un vettore di N elementi, ognuno assegnato ad un processore virtuale. La parte sinistra della regola è rappresentata da a[*i], mentre la destra da vps[i]. La *variabile libera* i, introdotta nella parte sinistra, permette di mappare lo stato sui processori virtuali. È necessario che ogni variabile dichiarata a sinistra venga utilizzata a destra. Inoltre, è possibile distribuire una intera dimensione della variabile su un processore virtuale, utilizzando l'espressione "`[]`":

```
attribute long a[N][M] scatter a[*i][] onto vps[i];
```

In questo caso, ad ogni processore virtuale spetta una riga della matrice a. Con la topologia ad albero è necessario disporre di diversi tipi di attributi partizionati:

- *Sulla radice*: lo stato è associato interamente al processore virtuale radice. È un caso particolare di partizionamento perché in realtà il dato non è distribuito tra un insieme di nodi. Inoltre, con la politica

di distribuzione **scheduled** è sempre possibile assegnare una variabile presente sullo stream di input al processore virtuale (0,0). Il suo utilizzo, però, aumenta la potenza espressiva associata alla nuova topologia; ciò può risultare importante nell'implementazione di alcune forme di parallelismo.

- *Sulle foglie*: lo stato è distribuito tra i processori virtuali che rappresentano le foglie dell'albero. Anche questo è un caso particolare di attributo partizionato perché si considera solo un sottoinsieme dei possibili nodi dell'albero. Il suo utilizzo, come nel caso precedente, può risultare importante per la definizione di forme di parallelismo specifiche per l'albero.

- *Sull'albero*: lo stato è distribuito tra tutti i processori virtuali. La semantica ed il tipo di utilizzo sono gli stessi di quelli degli attributi partizionati per la topologia array.

Gli attributi partizionati sulla radice e sulle foglie possono essere supportati utilizzando la regola di mapping vista. Mantenendo inalterata la sintassi dovrebbe essere possibile specificare, per la parte destra, dei valori numerici per identificare i processori virtuali. Ad esempio:

- Radice: `scatter a[*i] onto vps[0][0];`

- Foglie: `scatter a[*i] onto vps[3][*i];`

  Il valore 3 rappresenta la profondità dell'albero e corrisponde al valore specificato per il primo indice nella dichiarazione della topologia.

Nell'attuale versione di ASSIST non è possibile utilizzare tali espressioni, quindi sarebbe necessario modificare la sintassi e la semantica della regola. L'altra soluzione è quella di sostituire la parte destra con le parole chiave `root` e `leaves`: ció comporta lo stesso una modifica dello scanner, del parser e dei controlli semantici. In definitiva, quindi, con entrambe le soluzioni sono necessari dei cambiamenti.
Anche per quanto riguarda gli attributi partizionati sull'albero possiamo

utilizzare diverse strategie. Come prima soluzione potremmo usare la forma contratta di partizionamento, per la quale non è necessario specificare la regola di mapping:

```
attribute long a[N] scatter a onto vps;
```

Tuttavia, tale costrutto può essere utilizzato solo se la struttura della variabile è isomorfa a quella dei processori virtuali. Dal momento che la struttura ad albero può essere "schiacciata" su una sola dimensione, tale scelta non è adottabile se la variabile è multidimensionale.
La seconda soluzione potrebbe essere quella di utilizzare il costrutto esteso in modo da esprimere il mapping desiderato. Anche in questo caso incontriamo dei problemi legati alla semantica della regola di mapping. Infatti, per identificare il generico processore virtuale sul quale mappare lo stato sono necessari due indici, quindi la parte sinistra della regola deve contenere almeno due variabili libere, che saranno impiegate nella parte destra. Ad esempio:

```
attribute long a[N][M] scatter a[*i][*j] onto vps[i][j];
```

Nel caso di un attributo monodimensionale tale soluzione non è praticabile, per cui la regola non potrebbe essere definita correttamente.

Le soluzioni analizzate hanno tutte degli svantaggi e non forniscono un'espressività sufficiente per coprire tutti i casi. Per tale motivo la scelta è stata di introdurre un nuovo costrutto, simile per sintassi a quello già presente, con una semantica adatta ai nuovi tipi di attributi partizionati:

```
scatter_tree a[{var_libera}]{[]} onto vps{[valore][var_libera | valore]};
```

Per prima cosa viene introdotta la parola chiave `scatter_tree`, che permette di identificare la definizione degli attributi partizionati per l'albero. Il secondo passo è quello di specificare la semantica della regola di mapping. Adesso la parte sinistra può contenere, per la prima dimensione, una variabile libera oppure essere selezionata completamente. La variabile li-

bera viene utilizzata con gli attributi partizionati sulle foglie o sull'albero, per definire il mapping degli elementi sui processori virtuali. Le altre dimensioni dell'attributo, se presenti, devono essere selezionate completamente. Invece, la parte destra deve specificare obbligatoriamente il nome della topologia. Nel caso di attributi partizionati sulla radice o sulle foglie è necessario utilizzare anche gli indici per la selezione dei processori virtuali. Consideriamo il seguente frammento di codice.

Frammento di codice 3. Parmod ad albero: attributi partizionati.

```
1  parmod esempio_albero(input_stream ... output_stream ...) {
2    topology tree[l:3][a:2] vps;
3    attribute long m[5][6] scatter_tree m[][] onto vps[0][0];
4    attribute long n[8] scatter_tree n[*h] onto vps[3][h];
5    attribute long v[15][3] scatter_tree v[*i][] onto vps;
6
7    input_section { ... }
8
9    virtual_processors {
10     elab elaborazione(...) {
11       VP l = 0, a = 0 { //Elaborazione 1 }
12
13       VP l = 1..2, a { //Elaborazione 2 }
14
15       VP l = 3, a = 0..8 { //Elaborazione 3 }
16     }
17   }
18
19   output_section { ... }
20 }
```

Le righe 3-5 contengono la definizione dei nuovi tipi di attributo partizionato. Nel primo caso (riga 3) viene definita una matrice distribuita per intero al processore virtuale radice. Nella parte sinistra della regola di mapping ogni dimensione della variabile viene selezionata completamente, per indicare che tutto lo stato deve essere associato ad un unico elemento, mentre la parte destra contiene due indici, entrambi a 0, che identificano il processore virtuale di interesse.

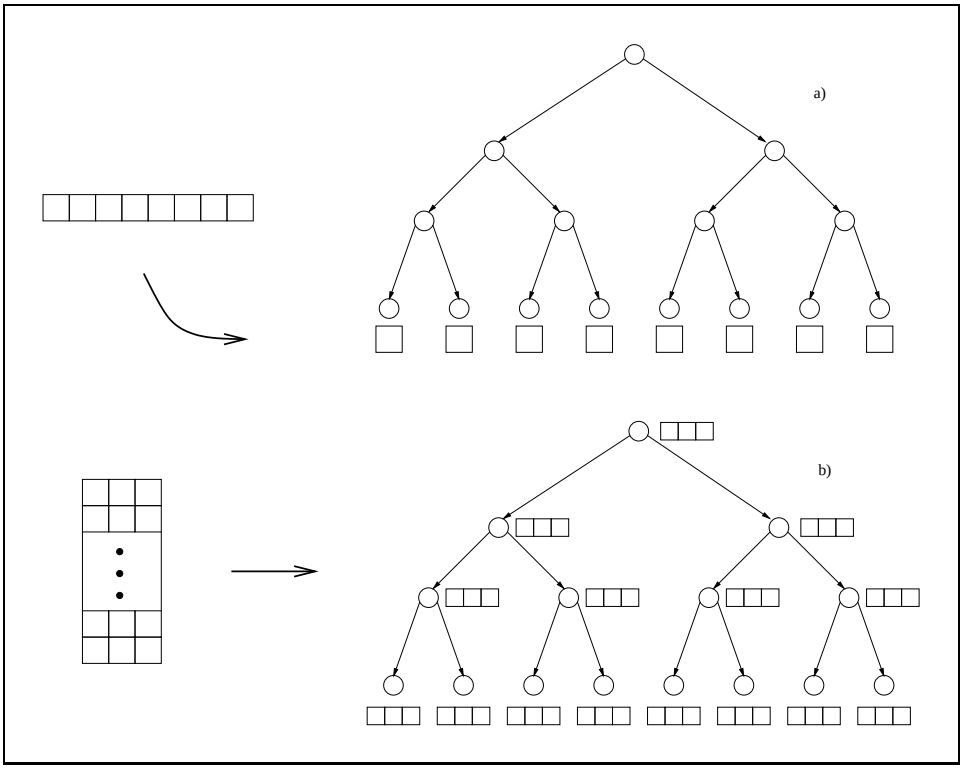


Figura 3.1: Attributo partizionato: a) sulle foglie; b) sull'albero.

Nel secondo (riga 4) si dichiara un vettore che sarà distribuito tra i processori virtuali che appartengono alle foglie dell'albero (figura 3.1a). La parte sinistra della regola contiene una variabile libera, utilizzata per identificare il generico elemento per il quale effettuare il mapping. Nella parte destra si specifica il generico processore virtuale foglia: il primo indice contiene la profondità dell'albero, corrispondente al valore inserito nella dichiarazione della topologia (riga 2), mentre il secondo contiene la variabile libera, che identifica il processore virtuale rispetto agli altri del suo livello.

Nell'ultimo caso (riga 5) viene dichiarata una seconda matrice che verrà distribuita tra tutti i processori virtuali (figura 3.1b). Nella parte sinistra della regola si utilizza, per la prima dimensione, una variabile libera, la quale servirà per selezionare la parte di stato da associare ad ogni nodo, mentre la seconda dimensione viene selezionata completamente: in questo modo la

matrice è distribuita per righe ai processori virtuali. La parte destra contiene semplicemente il nome della topologia.

Un altro aspetto da analizzare, legato agli attributi partizionati, è il tipo di accesso agli elementi dello stato effettuato dai processori virtuali. Le modalità di cooperazione e di interazione durante l'esecuzione di un task dipendono da questo punto.
Con una struttura logica ad albero i patterns di comunicazione tra due processori virtuali sono limitati; in particolare lo scambio delle informazioni avviene a livello padre-figlio. Questa caratteristica influenza il tipo di accesso allo stato distribuito.
Con attributi partizionati sulle radice e sulle foglie non abbiamo problemi nell'esprimere i riferimenti agli elementi. Nel primo caso lo stato è visibile solo al processore virtuale radice, il quale può utilizzare qualsiasi tipo di espressione. Nel secondo, ogni processore virtuale foglia gestisce una parte dello stato; dal momento che non sono ammessi accessi diretti agli elementi dei fratelli, ogni processore virtuale può riferire solo lo stato locale e per farlo deve specificare un'espressione che permetta di identificare la sua parte. Invece, nel caso di attributi partizionati sull'albero abbiamo un problema legato allo schema di naming utilizzato. Riprendiamo l'esempio precedente e consideriamo il processore virtuale (l,a). Per poter riferire lo stato locale è necessario determinare l'indice della riga ad esso associata. I valori l ed a non sono molti utili se usati separatamente, ma possono essere impiegati congiuntamente. In base allo schema di distribuzione dei dati utilizzato e ad alcune semplici operazioni su questi due valori possiamo ricavare l'indice della riga. Questo tipo di traduzione deve essere effettuato ogni volta che si accede allo stato locale o a quello remoto; in questo ultimo caso è necessario calcolare l'indice di riga associato al padre o al figlio. Tale inconveniente può essere risolto introducendo nel linguaggio di coordinamento delle macro da utilizzare per i riferimenti:

- `NODE_INDEX`: permette di riferire l'elemento dell'attributo associato al processore virtuale che la utilizza. Per ricavare tale valore si procede

nel seguente modo: in base al livello si calcola il numero di nodi ai quali è già stata assegnata una parte dello stato (secondo la regola di distribuzione scelta); quindi si somma a questo valore il contenuto del secondo indice usato per la selezione.

- `CHILD_INDEX(n)`: permette di calcolare l'indice dell'elemento associato al figlio del processore virtuale che ha fatto la chiamata. `n` indica il numero del figlio al quale riferirsi. Per ottenere il valore dell'indice si determina la coppia (l,a) che identifica il figlio e si procede alla traduzione utilizzando il metodo descritto al punto precedente.

- `PARENT_INDEX`: permette di riferire l'elemento gestito dal padre del processore virtuale che la utilizza. Anche in questo caso determiniamo per prima cosa la coppia (l,a) che identifica il processore virtuale padre, quindi si effettua il calcolo descritto sopra.

Vediamo alcuni semplici esempi di utilizzo di tali macro:

Frammento di codice 4. Parmod ad albero: utilizzo delle macro.

```
1  parmod esempio_albero(input_stream ... output_stream ...) {
2    topology tree[l:3][a:2] vps;
3    attribute long m[5][6] scatter_tree m[][] onto vps[0][0];
4    attribute long n[8] scatter_tree n[*h] onto vps[3][h];
5    attribute long v[15][3] scatter_tree v[*i][] onto vps;
6
7    input_section { ... }
8
9    virtual_processors {
10     elab elaborazione(...) {
11       VP l = 0, a = 0 {
12         esecuzione1(in v[CHILD_INDEX(1)][], v[CHILD_INDEX(2)][]
13                     out v[NODE_INDEX][]);
14       }
15
16       VP l = 1..2, a {
17         esecuzione2(in v[NODE_INDEX][] out v[NODE_INDEX][]);
18       }
19
```

```
20        VP l = 3, a = 0..8 {
21          esecuzione3(in v[PARENT_INDEX][] out v[NODE_INDEX][]);
22        }
23      }
24    }
25
26    output_section { ... }
27  }
```

Il processore virtuale radice esegue una funzione sullo stato associato ai figli e memorizza il risultato dell'elaborazione sul proprio stato locale (righe 12-13). I processori virtuali intermedi elaborano separatamente le informazioni contenute nel proprio stato locale, senza accedere allo quello remoto di altri (riga 17). I processori virtuali foglia, infine, recuperano il dato memorizzato dal padre e lo utilizzano per aggiornare lo stato locale (riga 21).

Resta adesso da valutare il possibile utilizzo delle variabili di controllo e degli streams interni. Questi, non essendo legati al tipo di topologia scelta, possono essere utilizzati anche con quella ad albero senza creare problemi, quindi non è necessario effettuare alcuna modifica o aggiornamento.

3.4 Politiche di distribuzione

A questo punto l'analisi si sposta sulle politiche di distribuzione dei valori ricevuti dagli streams di input. Ancora una volta, verranno prima valutate le politiche presenti in ASSIST e la possibilità di utilizzarle insieme con la nuova topologia. In caso contrario verranno descritte le aggiunte da effettuarsi.
Il linguaggio di coordinamento permette di definire quattro diverse politiche: `broadcast`, `on_demand`, `scheduled` e `scatter`. Nel caso di distribuzione `broadcast`, il dato viene distribuito a tutti i processori virtuali e ognuno ne riceve una copia. Tale politica può essere utilizzata anche con la topolo-

gia ad albero perché non richiede una regola particolare per selezionare i processori virtuali, ma solo il nome della topologia:

```
distribute nome_stream broadcast to vps;
```

La medesima cosa vale per le politiche `on_demand` e `scheduled`. Con quest'ultima, in particolare, la selezione del processore virtuale al quale spedire il dato può essere fatta utilizzando due indici, come descritto nella sezione 2, quindi rispettando le semantica della politica:

```
distribute nome_stream on_demand to vps;
```

```
distribute nome_stream scheduled to vps[ind1][ind2];
```

Per quanto concerne la distribuzione di tipo `scatter`, in questo caso potremmo avere dei problemi. La politica prevede la possibilità di distribuire il dato in input direttamente ai processori virtuali oppure su un attributo partizionato. Ad esempio:

```
distribute dato scatter dato[*i] to vps[i];
```

```
distribute dato scatter dato[*i] to attributo[i];
```

La prima possibilità obbligherebbe all'introduzione di un nuovo tipo di scatter per gestire gli attributi partizionati per l'albero, mentre la seconda non pone alcun problema.
La soluzione adottata è di permettere l'utilizzo della scatter solo per distribuire il dato in input su un attributo partizionato. In questo modo non è necessario modificare ulteriormente lo scanner ed il parser.
Riprendiamo il Parmod di esempio presentato in questo capitolo ed aggiungiamo la parte legata alle politiche di distribuzione (frammento di codice 5).

Il primo passo è stato quello di definire parte dell'interfaccia del Parmod, aggiungengo gli streams di input. Quindi abbiamo definito la sezione di input del modulo, nella quale vengono definite le politiche di distribuzione utilizzate. Per ogni stream di ingresso abbiamo utilizzato una diversa strategia di distribuzione.

Frammento di codice 5. Parmod ad albero: politiche di distribuzione.

```
1  parmod esempio_albero(input_stream long dato1[15][3], long dato2
2                        output_stream ...) {
3    topology tree[l:3][a:2] vps;
4    attribute long m[5][6] scatter_tree m[][] onto vps[0][0];
5    attribute long n[8] scatter_tree n[*h] onto vps[3][h];
6    attribute long v[15][3] scatter_tree v[*i][] onto vps;
7
8    input_section {
9      guardia: on, , dato1 && dato2 {
10       distribution dato1 scatter dato1[*j1][*j2] to v[j1][j2];
11       distribution dato2 broadcast to vps;
12     }
13   }
14
15   virtual_processors {
16     elab elaborazione(in guardia) {
17       VP l = 0, a = 0 {
18         esecuzione1(in v[CHILD_INDEX(1)][], v[CHILD_INDEX(2)][],
19                     dato2 out v[NODE_INDEX][]);
20       }
21
22       VP l = 1..2, a {
23         esecuzione2(in v[NODE_INDEX][], dato2
24                     out v[NODE_INDEX][]);
25       }
26
27       VP l = 3, a = 0..8 {
28         esecuzione3(in v[PARENT_INDEX][], dato2
29                     out v[NODE_INDEX][]);
30       }
31     }
32   }
33
34   output_section { ... }
35 }
```

Il dato in arrivo sul primo stream viene distribuito sull'attributo v (riga 10). Ogni elemento di dato1 viene mappato sul corrispondente elemento

di v. Successivamente, ogni riga di v viene associata ad un processore virtuale. Il dato ricevuto dal secondo stream viene inviato a tutti i processori virtuali (riga 11), i quali ne gestiscono una copia locale e la utilizzano come parametro di input per le proprie elaborazioni.

3.5 Strategie di collezione

L'analisi del progetto logico si conclude con una descrizione delle strategie di collezione, seguendo il filo conduttore delle precedenti sezioni.
ASSIST mette a disposizione due distinte strategie di collezione: `from ANY` e `from ALL`. Data la loro semantica, entrambe possono essere utilizzate con la topologia ad albero:

```
collects nome_stream from ANY vps;
```

```
collects nome_stream from ALL vps[l][a];
```

Oltre a queste soluzioni il programmatore potrebbe richiedere di effettuare una collezione per i valori prodotti dalle foglie o dalla radice dell'albero. È utile, quindi, introdurre due nuove strategie: `from ROOT` e `from LEAVES`. Ognuna di esse può essere considerata un caso particolare delle rispettive strategie già presenti in ASSIST; per questo motivo le modifiche si concentrano principalmente sul parser e sui controlli semantici.

Completiamo l'esempio di Parmod utilizzato fino ad ora con la parte relativa alla sezione di output (frammento di codice 6).
L'interfaccia del Parmod è stata completata aggiungendo due streams di output. La sezione di output contiene i costrutti utilizzati per la raccolta dei risultati prodotti durante l'esecuzione e sui quali streams inviare tali valori. Alla riga 34 si sfrutta la strategia di collezione `from ROOT` per recuperare il dato prodotto dalla radice: la sintassi del costrutto è identica a quella della strategia `from ANY`. Nella riga successiva si utilizza la collezione di tipo `from LEAVES`.

Frammento di codice 6. Parmod ad albero: collezione dei risultati.

```
parmod esempio_albero(input_stream long dato1[15][3], long dato2
                      output_stream long ris1[5][6], long ris2[8]) {
  topology tree[l:3][a:2] vps;
  attribute long m[5][6] scatter_tree m[][] onto vps[0][0];
  attribute long n[8] scatter_tree n[*h] onto vps[3][h];
  attribute long v[15][3] scatter_tree v[*i][] onto vps;
  stream long ris;

  input_section {
    guardia: on, , dato1 && dato2 {
      distribution dato1 scatter dato1[*j1][*j2] to v[j1][j2];
      distribution dato2 broadcast to vps;
    }
  }

  virtual_processors {
    elab elaborazione(in guardia out ris1, ris) {
      VP l = 0, a = 0 {
        esecuzione1(in v[CHILD_INDEX(1)][], v[CHILD_INDEX(2)][],
                    dato2 out ris1);
      }
      VP l = 1..2, a {
        esecuzione2(in v[NODE_INDEX][], dato2
                    out v[NODE_INDEX][]);
      }
      VP l = 3, a = 0..8 {
        esecuzione3(in v[PARENT_INDEX][], dato2
                    out ris2);
      }
    }
  }

  output_section {
    collects ris1 from ROOT vps;
    collects ris from LEAVES vps[a] { //Aggregazione }<>;
  }
}
```

La sintassi, anche in questo caso, è identica a quella della strategia `from ALL` ma con una piccola modifica: invece di due indici per la selezione del generico processore virtuale, si utilizza un unico indice, il cui valore viene definito dalla variabile di topologia che definisce l'arietà dell'albero. Tale indice è sufficiente per identificare correttamente i processori virtuali foglia.

3.6 Esempio di utilizzo

Consideriamo il seguente frammento di codice che descrive la struttura di un semplice Parmod con topologia ad albero.

```
parmod top_albero(input_stream long elementi[8]
                  output_stream long risultato[8]) {
  topology tree[l:N][a:2] vps;

  attribute long tmp[8] scatter_tree copia[] onto vps[0][0];
  attribute long stato[15][8] scatter_tree v[*j][] onto vps;
  stream long s;

  input_section {
    guardia: on, , elementi {
      distribution elementi scatter elementi[*h] to tmp[h];
    }
  }

  virtual_processors {
    elab elaborazione(in guardia out s) {
      VP l = 0, a = 0 {
        for(int ind1 = 0; ind1 <= N + 1; ind1++) {
          copia(in l, ind1, tmp[] out stato[NODE_INDEX][]);
        };
      }
      VP l = 1..N-1, a {
        for(int ind2 = 0; ind2 <= N + 1; ind2++) {
          fun(in l, ind2, stato[PARENT_INDEX][] out stato[NODE_INDEX][]);
        };
      }
```

```
      VP l = N, a {
        for(int ind3 = 0; ind3 <= N + 1; ind3++) {
          fun(in l, ind3, stato[PARENT_INDEX][] out stato[NODE_INDEX][]);
          assist_out(s, stato[NODE_INDEX][0]);
        }
      }
    }
  }

  output_section {
    collects s from LEAVES vps[a] {
      //Aggregazione e invio su risultato;
    }<>;
  }
}
```

L'esempio fatto prende in esame un tipico schema di elaborazione che è possibile realizzare con la topologia ad albero: una computazione strutturata secondo il paradigma divide et impera. Il dato da elaborare viene ricevuto sullo stream *elementi* ed è distribuito al processore virtuale radice attraverso una copia dei suoi valori sull'attributo *tmp*. A questo punto inizia l'elaborazione, effettuata in diverse fasi. Ognuna di queste permette di eseguire le operazioni definite da *fun* su un sottoinsieme dei dati di partenza. Nella prima fase il processore virtuale radice copia il dato ricevuto nella parte locale dell'attributo *stato*, in modo che anche i figli possano accedere a tali valori. Le successive fasi vengono attivate di volta in volta per uno specifico livello dell'albero; l'attivazione viene controllata dai parametri *l*, *ind2* ed *ind3* di *fun*. In ognuna i processori virtuali interessati recuperano il dato prodotto dal padre e lo rielaborano copiando il risultato ottenuto nel proprio stato locale. L'esecuzione termina quando siamo arrivati alle foglie dell'albero. A questo punto gli ultimi valori calcolati vengono inviati sullo stream interno del Parmod (*s*), raccolti ed aggregati per produrre il risultato finale.

CAPITOLO 4

Topologia ad albero: implementazione

In questa parte l'attenzione si sposta sull'implementazione della topologia ad albero. L'integrazione delle nuove funzionalità all'interno del Parmod si può riassumere in due punti: estensione del linguaggio di coordinamento con i costrutti descritti nel capitolo precedente; introduzione nella libreria ASSIST del supporto necessario per la gestione del nuovo tipo di parallelismo.
L'analisi che faremo prende in esame la struttura interna del compilatore; per ogni fase si descrivono le modifiche e le aggiunte apportate e le scelte di progetto effettuate.
La parte principale del capitolo è rivolta al back-end, in cui si passa da una rappresentazione del programma come rete di nodi task-code ad una come rete di processi. È in questa fase, infatti, che si prendono le decisioni più importanti legate alla strategia di implementazione del parallelismo "ad albero": rappresentazione dei processori virtuali, mapping dei processori virtuali sui processori fisici, gestione dello stato distribuito.

4.1 Front-end

In questa fase vengono attivati lo scanner ed il parser, che analizzano la struttura del programma sorgente e ne creano una rappresentazione interna. A partire da questa vengono effettuati i controlli semantici per verificare la correttezza dei costrutti utilizzati. Le modifiche apportate in questa fase della compilazione servono principalmente per aggiungere il supporto dei nuovi costrutti all'interno del linguaggio di coordinamento.

4.1.1 Scanner

Sono state aggiunte nuove parole chiave nel file `assistKwords.gperf`, contenente la lista di quelle riconosciute dal linguaggio di coordinamento. Ogni parola viene definita dalla coppia (*nome*, *token*):

- `tree`, **_TREE_**
- `scatter_tree`, **_SCATTER_TREE_**
- `NODE_INDEX`, **_NODE_INDEX_**
- `CHILD_INDEX`, **_CHILD_INDEX_**
- `PARENT_INDEX`, **_PARENT_INDEX_**
- `ROOT`, **_ROOT_**
- `LEAVES`, **_LEAVES_**

4.1.2 Parser

Il parser raccoglie le informazioni provenienti dallo scanner e le utilizza per ricostruire la struttura sintattica del programma, ne verifica la correttezza e ne crea la rappresentazione interna. È necessario modificare la grammatica utilizzata dal parser in modo che in essa vengano inserite le produzioni per il riconoscimento dei nuovi costrutti del linguaggio.

Il primo passo è stato quello di aggiungere i token associati alle nuove parole chiave nella lista dei simboli terminali. A questo punto è stato possibile procedere con le modifiche alla grammatica.

Dichiarazione della topologia

Le produzioni che permettono di riconoscere la topologia del Parmod sono le seguenti:

topology_decl: _**TOPOLOGY**_ *topology_type*
topology_type: *array_topology* | *none_topology* | *one_topology*

topology_type è stata arricchita con la regola *tree_topology*, che permette di riconoscere la dichiarazione della nuova topologia (A.1):

tree_topology: _**TREE**_ *levels_dimension children_dimension* _**ID**_ _**SEMICOLON**_
level_dimension: _**LSB**_ _**ID**_ _**COLON**_ *ide_expr* _**RSB**_
children_dimension: _**LSB**_ _**ID**_ _**COLON**_ *ide_expr* _**RSB**_

levels_dimension permette di identificare il primo indice, che contiene il valore sul numero di livelli dell'albero, mentre *children_dimension* viene usata per riconoscere il secondo indice, che contiene il valore sull'arietà dell'albero.
Tale soluzione permette di assegnare un tipo diverso ad ognuna delle variabili usate per le dimensioni, in modo che durante i controlli semantici sia possibile verificare che gli indici utilizzati per la selezione dei processori virtuali siano stati impiegati nell'ordine giusto. Per questo motivo alla prima variabile si assegna tipo `TOPOLOGY_LEVELS`, mentre alla seconda `TOPOLOGY_CHILDREN`.
Il controllo relativo al numero di dimensioni specificate viene lasciato nelle mani del parser, il quale solleverà un errore nel caso in cui non vengano definite esattamente due dimensioni.

La produzione *tree_topology* raccoglie le informazioni sulla topologia e le memorizza nella classe `AST_Tree` (A.2), aggiunta a quelle utilizzate per gestire la rappresentazione interna del programma.

Attributi partizionati

Come abbiamo visto nel capitolo precedente, gli attributi replicati possono essere utilizzati anche con la topologia ad albero senza creare problemi. La nostra attenzione, quindi, si concentra sui nuovi attributi partizionati. Partiamo dalla produzione utilizzata per riconoscerne la dichiarazione:

partitioned_variables_decl: *attribute_declaration* **_SCATTER_** *part_expr* **_ONTO_** *part_expr* **_SEMICOLON_** |
attribute_declaration **_ONTO_** **_ID_** **_SEMICOLON_**

Ad essa è stata aggiunta la regola (A.3):

attribute_declaration **_SCATTER_TREE_** *part_expr* **_ONTO_** *destination* **_SEMICOLON_**

destination: *part_expr* | **_ID_**

La classe `AST_Attribute` memorizza tutte le informazioni sull'attributo dichiarato. Ad essa sono stati aggiunti i valori `ISO_TREE` e `SUBSET_TREE` relativi al tipo di mapping usato. Il primo si riferisce ad un attributo partizionato su tutto l'albero, mentre il secondo agli attributi partizionati sulle foglie e sulla radice.
Nel caso di attributo con mapping di tipo `SUBSET_TREE`, il parser userà la seguente regola per riconoscerne la dichiarazione:

attribute_declaration **_SCATTER_TREE_** *part_expr* **_ONTO_** *part_expr* ...

in cui la produzione *part_expr* permette di identificare sia la parte sinistra

che quella destra della regola di mapping ed il tipo di indici utilizzati.
Invece, per gli attributi con mapping di tipo `ISO_TREE`, la regola per il riconoscimento della dichiarazione è la seguente:

attribute_declaration **_SCATTER_TREE_** *part_expr* **_ONTO_ _ID_** ...

in cui la parte sinistra della regola di mapping viene ancora identificata tramite la produzione *part_expr*, mentre la parte destra è rappresentata da un identificatore, nel nostro il nome della topologia.
Per tali attributi è necessario aggiungere il supporto per il riconoscimento delle macro `NODE_INDEX`, `CHILD_INDEX` e `PARENT_INDEX`, utilizzate per riferire gli elementi dello stato partizionato. La produzione di riferimento è *dim_value*, che permette di recuperare le informazioni sul tipo di espressione usata come indice per la dimensione di una variabile. Ad essa sono state aggiunte le regole (A.4):

_NODE_INDEX_
_CHILD_INDEX_ _LRB_ *const_value* **_RRB_**
_PARENT_INDEX_

Le informazioni raccolte durante il parsing vengono memorizzate nella classe `AST_MacroExpr` (A.5) e vengono effettuati i primi controlli per verificare il corretto utilizzo delle macro: `NODE_INDEX` può essere usata come espressione sia per parametri di input che di output, mentre le altre due solo per quelli di input. Tale limitazione è dovuta alla *owner compute rule*, secondo la quale ogni processore virtuale può avere accesso solo in lettura allo stato remoto.

Strategie di collezione

Anche in questo caso è stato necessario modificare le produzioni che permettono di riconoscere le strategie di collezione usate:

collection_statement: **_COLLECTS_** *RValue* **_FROM_** *collection_kind*
collection_kind: **_ANY_** | **_ALL_**

Sono stati aggiunti i simboli terminali **_ROOT_** e **_LEAVES_** alla regola *collection_kind*. Le informazioni vengono raccolte nella classe `AST_OutputSection`, alla quale sono stati aggiunti i valori `COLLECT_ROOT` e `COLLECT_LEAVES` come tipo per le nuove strategie.

4.1.3 Controlli Semantici

Una volta costruita la rappresentazione interna del programma, vengono effettuati i controlli semantici per stabilire la correttezza dei costrutti utilizzati. In questa fase sono stati aggiunti diversi controlli molto rigidi legati alla nuova topologia e all'utilizzo che viene fatto delle sue funzionalità. È importante stabilire con estrema accuratezza che ogni costrutto impiegato rispetti le regole semantiche che vedremo in seguito, perché nelle fasi successive della compilazione si fanno alcune assunzioni basate su tali vincoli.

Innanzitutto è stata aggiornata la procedura che inizializza le variabili utilizzate per definire le caratteristiche semantiche del Parmod (ad esempio quali politiche di distribuzione è possibile usare oppure se i processori virtuali devono essere indicizzati) ed impiegate durante i controlli. La configurazione della topologia ad albero viene impostata nello stesso modo di quella array, perché entrambe hanno le medesime caratteristiche e funzionalità.
Le parti di cui ci occuperemo adesso riguardano la selezione dei processori virtuali, la dichiarazione e l'utilizzo degli attributi partizionati e le politiche di collezione.

Selezione dei processori virtuali

Lo schema di naming adottato per la topologia ad albero si basa sull'utilizzo di due indici, impiegati per identificare il livello ed il numero del nodo all'interno del livello. È necessario verificare che tale schema venga

rispettato ogni volta che si effettuata una selezione dei processori virtuali. Consideriamo la seguente dichiarazione di topologia:

`topology tree`[l:3][a:2] vps.

Ad esempio, per identificare il processore virtuale (1,1), il figlio destro della radice, utilizziamo il costrutto `VP` l=1, a=1. Tale selezione è corretta per tre motivi:

1. Utilizza due indici.
2. L'ordine con il quale gli indici sono specificati è corretto.
3. I valori associati agli indici sono validi.

Ognuna delle tre condizioni è una diretta conseguenza dello schema di naming utilizzato. Le condizioni 2 e 3 sono ulteriormente legate dal fatto che i due indici non sono completamente indipendenti, come accade per la topologia array, ma i valori che possono assumere dipendono fortemente l'uno dall'altro; in particolare i valori assunti dal primo influenzano il secondo.
Per tale motivo è necessario che i due indici vengano utilizzati nell'ordine che fin qui abbiamo descritto. In questo modo risulta più semplice verificare la correttezza di ogni singola selezione. Inoltre, le operazioni di controllo sono facilitate anche dal fatto che ad ogni indice è stato assegnato un tipo diverso.
Il controllo sulla validità dei valori specificati va effettuato per essere sicuri che l'insieme selezionato venga definito in modo da rispettare la struttura dell'albero ed il numero dei suoi elementi. I valori che è possibile associare ad ogni indice sono riassunti dallo schema seguente:

Primo indice	*Secondo indice*
libero	libero
costante	libero / costante / range
range	libero

Se il primo indice viene lasciato libero (`VP` l, ...) la selezione viene effettuata per tutti i livelli dell'albero, per cui anche il secondo indice deve essere lasciato libero. Questa condizione deriva dal fatto che, non avendo specificato un determinato livello di riferimento per gli elementi, non è possibile determinare un range da associare al secondo indice. Il tipo di selezione che otteniamo è `VP` l, a, cioé per tutto l'albero.
Se al primo indice è associato un unico livello (`VP` l=3, ...) allora il secondo può assumere qualsiasi valore. Se questo è libero, otteniamo una selezione per un intero livello dell'albero (`VP` l = 3, a); se è stata utilizzata una costante la selezione avviene per un unico nodo (`VP` l=3, a=4); ma se è stato specificato un range la selezione riguarda un sottoinsieme dei nodi del livello (`VP` l=3, n=2..5).
Se per il primo indice utilizziamo un range di valori (`VP l=0..2`) la selezione dovrà avvenire per un sottoinsieme dei livelli dell'albero, per cui il secondo indice deve essere lasciato libero (`VP` l =0..2, a). La situazione è la stessa che si presenta nel caso di selezione con primo indice libero.

Attributi partizionati

I controlli semantici sugli attributi partizionati permettono di valutare la correttezza della regola di mapping e di controllare il tipo di espressioni usate per l'accesso allo stato distribuito.
La regola di mapping viene rappresentata da una espressione "sorgente", che identifica la generica parte di stato da distribuire, e una espressione "destinazione", che identifica il generico processore virtuale al quale assegnare la porzione selezionata. Nel caso di attributi partizionati per la topologia ad albero possiamo avere diversi tipi di espressioni e la loro combinazione dipende dal tipo di mapping scelto:

- *Mapping sulla radice*: in questo caso l'intero stato viene associato al processore virtuale radice, quindi l'espressione sorgente non deve contenere nessuna forma di selezione; ogni dimensione della variabile deve essere considerata nel suo complesso:

attributo[][]...[]

L'espressione destinazione deve contenere il riferimento al processore virtuale (0,0), cioé deve specificare il nome della topologia seguito da due indici, entrambi a 0:

nome_topologia[0][0]

Un esempio di tale regola è il seguente:

```
attribute long m[5] scatter_tree m[] onto vps[0][0];
```

- *Mapping sulle foglie*: lo stato viene distribuito tra i processori virtuali foglia. Il partizionamento avviene rispetto alla prima dimensione dell'attributo, lasciando libere le altre, quindi l'espressione sorgente deve contenere, per tale dimensione, una variabile libera, usata per identificare la generica porzione della variabile:

  attributo[*variabile_libera*][]...[]

  L'espressione destinazione deve contenere il riferimento al generico nodo foglia, al quale associare lo stato selezionato:

  nome_topologia[*livelli*][*variabile_libera*]

  Il primo indice specifica il livello di appartenenza delle foglie, mentre il secondo la variabile libera introdotta nell'espressione sorgente, usata per identificare il generico processore virtuale foglia tra tutti quelli del suo livello.
  Un esempio di tale regola è il seguente:

```
attribute long n[5][5] scatter_tree n[*i][] onto vps[4][i];
```

- *Mapping sull'albero*: la variabile viene distribuita tra tutti i nodi dell'albero. Anche in questo caso il partizionamento avviene solo per la prima dimensione, quindi l'espressione sorgente è la stessa di quella precedente. L'espressione destinazione deve contenere solo

il nome della topologia, senza specificare nessun indice di selezione; tale soluzione viene adottata per identificare l'insieme di tutti i processori virtuali che appartengono alla topologia.
Un esempio di questa regola viene fornito dalla seguente dichiarazione:

```
attribute long s[7][10] scatter_tree s[*h][] onto vps;
```

Per quanto riguarda il controllo sul tipo di espressioni utilizzate per l'accesso alle variabili di stato, è importante verificare che vengano rispettati i patterns di comunicazione permessi all'interno dell'albero. Una comunicazione tra due processori virtuali avviene tramite un accesso in lettura da parte di un processore allo stato remoto dell'altro. Nel nostro caso lo scambio di informazioni può avvenire solo tra padre e figlio. Tale tipo di comunicazione è implementato utilizzando un attributo partizionato sull'albero e mettendo a disposizione di un processore virtuale la possibilità di riferire gli elementi di interesse: quello locale, quello del padre, quello di un figlio.
Oltre a questa limitazione generale, esistono una serie di condizioni che devono essere rispettate affinché i nuovi attributi partizionati siano utilizzati correttamente:

1. Un attributo partizionato sulla radice o sulle foglie non può essere utilizzato come parametro di una funzione che inizializza lo stato. La chiamata a tale funzione può essere definita solo se l'insieme dei processori virtuali che la eseguono comprende tutti gli elementi della topologia, pertanto non è permesso selezionare un particolare sottoinsieme. Per questo motivo la variabile non potrebbe essere inizializzata correttamente. Tale condizione viene posta per evitare di modificare la semantica della chiamata e semplifica il processo di integrazione.

2. Un attributo partizionato sulla radice può essere usato solo per elaborazioni associate al processore virtuale radice. Questa condizione è una conseguenza del fatto che tale tipo di attributo è visibile solo alla radice dell'albero, la quale ha un accesso esclusivo ai suoi elementi. Un esempio di tale utilizzo è fornito dal seguente codice:

```
topology tree[l:3][a:2] vps;
attribute long m[5] scatter_tree m[] onto vps[0][0];

virtual_processors {
  elaborazione(...) {
    VP l=0,a=0 { f(in m[] out m[2]); }
  }
}
```

La radice può accedere allo stato selezionandolo completamente oppure riferendo un elemento preciso tramite una costante o la variabile di iterazione del for.

3. Un attributo partizionato sulle foglie può essere impiegato solo per elaborazioni associate ai processori virtuali foglia. Ognuno di essi ha una visibilità legata alla porzione di stato assegnatogli, del quale è l'unico gestore, e non può accedere a quello remoto appartenente ad un fratello. Vediamo un esempio:

```
topology tree[l:3][a:2] vps;
attribute long n[8][6] scatter_tree n[*i][] onto vps[3][i];

virtual_processors {
  elaborazione(...) {
    VP l=3,a { f(in n[a][2] out n[a][]); }
  }
}
```

L'accesso allo stato locale avviene specificando come prima dimensione la variabile di topologia che identifica il numero del processore virtuale rispetto a tutti gli altri del suo livello (a); in questo modo è possibile riferire la porzione di stato che è stata associata al processore virtuale. Per quanto riguarda le altre dimensioni, dal momento che queste vengono assegnate completamente ad ogni foglia, non ci sono limitazioni sul tipo di espressione utilizzata per riferire gli elementi: questa può essere una costante o la variabile di iterazione del for.

4. Un attributo partizionato sull'albero può essere utilizzato, senza in-

contrare particolari problemi legati alla selezione dei processori virtuali, in qualsiasi elaborazione. L'accesso allo stato avviene tramite le macro NODE_INDEX, CHILD_INDEX e PARENT_INDEX. La prima può essere utilizzata sia per l'accesso in lettura che in scrittura perché permette di riferire gli elementi locali di un processore virtuale, mentre le altre possono essere impiegate solo per l'accesso in lettura, dal momento che riferiscono lo stato remoto di un altro processore virtuale. Inoltre, la macro CHILD_INDEX non può essere usata se sono stati selezionati i processori virtuali foglia (non avendo figli), mentre la macro PARENT_INDEX non può essere usata per il processore virtuale radice (non avendo padre). Vediamo due semplici esempi:

```
topology tree[l:3][a:2] vps;
attribute long s1[15][10] scatter_tree s1[*i][] onto vps;

virtual_processors {
  elaborazione(...) {
    VP l=0..2,a {
      f1(in s1[CHILD_INDEX(1)][], s1[CHILD_INDEX(2)][]
         out s1[NODE_INDEX][]); }
    VP l=3,a {
      f2(in s1[NODE_INDEX][] out s1[NODE_INDEX][]); }
  }
}


topology tree[l:3][a:2] vps;
attribute long s2[15] scatter_tree s2[*i] onto vps;

virtual_processors {
  elaborazione(...) {
    VP l=0,a=0 {
      f1(in s2[NODE_INDEX] out s2[NODE_INDEX]); }
    VP l=1..3,a {
      f2(in s2[PARENT_INDEX] out s2[NODE_INDEX]); }
  }
}
```

Strategie di collezione

I controlli semantici legati alla sezione di output permettono di verificare il corretto utilizzo delle strategie di collezione.
Ogni strategia può essere impiegata per la raccolta dei risultati sia per gli streams di output che per quelli interni del Parmod, quindi è necessario verificare, per prima cosa, che ogni stream sia utilizzato con un'unico costrutto collect.
Le nuove strategie introdotte hanno la stessa semantica di utilizzo di quelle già presenti in ASSIST: la strategia `from ROOT` è identica a quella `from ANY`, mentre la strategia `from LEAVES` è identica a quella `from ALL`. Entrambe possono essere considerate delle specializzazioni di tali strategie e per tale motivo è necessario effettuate dei controlli più specifici.
Nel caso in cui la strategia di collezione usata sia di tipo `from ROOT`, la semantica ad essa associata suppone che il dato raccolto verrà spedito dal processore virtuale radice. Per tale motivo si deve controllare che lo stream dal quale si riceve i dati venga utilizzato soltanto in una elaborazione in cui è stata selezionata solo la radice. L'esempio che segue mostra l'utilizzo corretto della strategia:

```
parmod collect_root(... output_stream long risultato) {
topology tree[l:3][a:2] vps;
attribute long s replicated;

virtual_processors {
  elaborazione(in ... out risultato) {
    VP l=0,a=0 { f1(in s output_stream risultato); }
    VP l=1..3,a { f2(out s); }
  }

  output_section {
    collects risultato from ROOT vps;
  }
}
```

Nel caso di collezione `from LEAVES` la semantica della strategia suppone che i dati ricevuti vengano inviati dai processori virtuali foglia. Per tale

motivo deve essere effettuato un controllo per verificare che lo stream usato per la raccolta sia impiegato in una elaborazione associata solo alle foglie dell'albero. Eccone un esempio:

```
parmod collect_root(... output_stream long risultato[8]) {
topology tree[l:3][a:2] vps;
attribute long s replicated;
stream long risultato_parziale;

virtual_processors {
  elaborazione(in ... out risultato_parziale) {
    VP l=0..2,a { f1(out s); }
    VP l=1..3,a { f2(in s output_stream risultato_parziale); }
  }

  output_section {
    collects risultato_parziale from LEAVES vps[a];
    {
      //Viene restituito il dato aggregato memorizzato
      //in risultato.
    }
  }
}
```

La strategia prevede l'utilizzo di uno stream interno per la raccolta di tutti i risultati parziali prodotti dai processori virtuali foglia; successivamente viene eseguita una operazione di aggregazione di tali valori in un unico dato inviato sullo stream di output del Parmod.

4.2 Middle-end

In questa fase la rappresentazione interna del programma viene tradotta nel codice task-code, il linguaggio intermedio adottato dal compilatore. Il Parmod viene rappresentato come una rete di nodi task-code, che descrivono il supporto a run-time di ASSIST.
La nostra attenzione è rivolta alla rete che rappresenta il Parmod parallelo (topologie none e array), costituita dai nodi ISM, VPM e OSM. A livello logico il suo funzionamento è il seguente: i dati ricevuti dagli streams di

input vengono gestiti dal nodo ISM, il quale, a seconda della politica di distribuzione utilizzata, li invia ai VPM, che hanno il compito di gestire i processori virtuali e le elaborazioni parallele; possiamo avere una o più istanze di tale nodo, in base al grado di parallelismo del Parmod; terminata l'esecuzione i risultati vengono comunicati al nodo OSM, il quale si occupa di collezionare i dati e inviarli sugli streams di output.

4.2.1 Nodo ISM

Rappresenta uno dei nodi di supporto della rete. Le elaborazioni ad esso associate costituiscono una parte del supporto a run-time messo a disposizione da ASSIST. Le principali operazioni svolte riguardano la gestione delle guardie associate agli streams di input e l'esecuzione delle politiche per la distribuzione dei dati.
Su questo ultimo aspetto si sono concentrate le modifiche effettuate al task-code. Con la topologia ad albero possono essere utilizzate tutte le politiche di distribuzione definite nel linguaggio di coordinamento. Tra queste, la **scatter** è l'unica che impiega i nuovi attributi partizionati; in base alla regola di mapping utilizzata deve essere effettuata una diversa operazione di distribuzione. Per questo motivo sono stati aggiunti nuovi tipi per le politiche di distribuzione, ognuno dei quali dovrà essere integrato all'interno del supporto a run-time attraverso una classe: `scatter_root`, `scatter_leaves` e `scatter_tree`. A queste politiche viene affiancato uno stream che supporta comunicazioni 1:N, definito all'interno del task-code, in cui il nodo ISM si comporta da produttore ed i nodi VPM da consumatori.

4.2.2 Nodo VPM

Rappresenta il nodo parallelo appartenente alla rete di nodi task-code e si occupa dell'esecuzione delle elaborazioni parallele. Fornisce il supporto per la gestione dei processori virtuali e per l'accesso allo stato condiviso.
Per ogni elaborazione definita nel Parmod viene memorizzato l'insieme dei nodi selezionati e la lista dei tasks ad essi associati.

L'integrazione con la nuova topologia viene realizzata introducendo un metodo per rappresentare le informazioni sulla struttura dell'albero e per definire correttamente gli insiemi di processori virtuali selezionati per l'esecuzione.

La topologia viene rappresentata tramite due range: il primo contiene il numero di livelli dell'albero, mentre il secondo la sua arietà ed il grado di parallelismo associato al Parmod. Quest'ultimo parametro è molto importante e viene utilizzato per effettuare il mapping dei processori virtuali sui VPM.

Anche per la definizione dell'insieme dei processori virtuali selezionati per una elaborazione si utilizzano due range. Combinando opportunamente i valori possibili siamo in grado di definire l'insieme desiderato:

- *Sottoinsieme di nodi di un livello*: [l...l][i...j]. Per il primo range si usa un unico valore, che identifica il livello scelto, mentre per il secondo si usano due valori diversi, che identificano il sottoinsieme dei processori virtuali del livello.

- *Nodi appartenenti a più livelli*: [l1...l2][n1...n2]. Per il primo indice si utilizzano due valori distinti, che identificano il livello iniziale e quello finale selezionati. Anche per il secondo indice si usano due valori distinti, che rappresentano rispettivamente il numero di nodi del livello iniziale e di quello finale.

Oltre alla topologia ad albero è necessario introdurre all'interno del nodo il supporto per i nuovi attributi partizionati. Per fare questo sono stato aggiunti i valori `root_partitioned`, `leaves_partitioned` e `tree_partitioned` alla lista dei tipi di attributi presenti nel task-code.

4.2.3 Nodo OSM

Anche questo rappresenta un nodo di supporto tra quelli che definiscono la rete di nodi task-code. Il suo compito principale è quello di collezionare i risultati delle elaborazioni, ricevuti dai nodi VPM, produrre il dato aggregato ed inviarlo sullo stream di output del Parmod.

L'integrazione delle nuove funzionalità all'interno di tale nodo si limita all'aggiunta dei tipi `from_root` e `from_leaves` a quelli delle strategie di collezione già definite nel task-code.
Per entrambe è necessario utilizzare uno stream che supporti comunicazioni N:1, in cui i nodi VPM rappresentano i produttori ed il nodo OSM il consumatore. Il task-code mette a disposizione due tipi di stream con tali caratteristiche: **any** ed **all**. Il primo è utilizzato per la strategia `from ANY`, mentre il secondo per quella `from ALL`. Se questi venissero impiegati anche per le nuove strategie non avremmo abbastanza informazione per distinguere la politica effettivamente utilizzata. Questo problema nasce dal fatto che la topologia ad albero supporta tutte e quattro le strategie descritte. Di conseguenza ciascun stream verrebbe impiegato per due strategie, creando problemi per la selezione di quella corretta.
Per tale motivo sono stati introdotti due nuovi tipi di streams, `root` e `leaves`, utilizzati esclusivamente per le nuove politiche di collezione.

4.3 Back-end

La fase finale della compilazione consiste nella traduzione della rete di nodi task-code in una rete di processi. I singoli nodi vengono "scritti" dal modulo builder come un insieme di classi ed oggetti C++ che costituiscono il supporto a run-time.
In questa parte vengono effettuate le scelte principali legate alla strategia di implementazione del parallelismo ad albero. Gli aspetti principali da considerare riguardano la rappresentazione dei processori virtuali e lo schema utilizzato per il mapping sui processori fisici, la gestione dello stato distribuito e delle elaborazioni parallele. Ogni scelta influenza il tipo di modifiche ed aggiunte che devono essere apportate alla libreria ASSIST.

4.3.1 Gestione dei processori virtuali

All'interno della libreria ASSIST i processori virtuali vengono rappresentati utilizzando la classe `TopologyGroup`. In essa viene definito

una vettore di dimensione pari alla cardinalità della struttura dichiarata per la topologia. Ad esempio, una topologia array con due dimensioni viene rappresentata come un vettore di due elementi, ad ognuno dei quali è possibile associare un range continuo di valori. In questo modo un generico sottoinsieme di processori virtuali può essere definito specificando opportunamente i valori di ogni dimensione.

Il primo passo da compiere verso l'integrazione della nuova topologia all'interno del supporto a run-time consiste nel valutare le possibilità che abbiamo di utilizzare tale classe per rappresentare i processori virtuali strutturati ad albero. Le scelte possibili sono le seguenti:

1. Utilizzare un vettore di dimensione uguale al numero di livelli dell'albero e specificare per ognuno un range relativo ai nodi.

2. Utilizzare un sola dimensione ed effettuare un mapping logico dell'albero su un array monodimensionale, identificando ogni processore virtuale tramite un solo indice.

Con la rappresentazione descritta al punto 1 la struttura dell'albero resterebbe inalterata, permettendo l'accesso ai singoli elementi utilizzando l'indicizzazione vista fino ad ora. È anche possibile selezionare particolari insiemi dell'albero specificando i range dei singoli livelli in maniera indipendente (figura 4.1). Il range nullo [0,-1] assegnato ad un indice viene utilizzato per specificiare che i nodi del livello ad esso corrispondente non devono far parte della selezione.
Il problema di tale soluzione è dato dal fatto che la classe `TopologyGroup` è stata studiata per rappresentare insiemi allineati di indici per array multidimensionali e parte dei metodi dovrebbero essere modificati o riadattati completamente per supportare la rappresentazione dell'albero.

Con la seconda soluzione riusciamo a risolvere il problema appena descritto perché rappresentiamo l'albero utilizzando un array monodimensionale. Ció comporta la perdita completa delle informazioni sulla sua struttura, con conseguenze molto importanti per l'integrazione di alcune funzionalità. In particolare, non siamo più in grado di determinare il val-

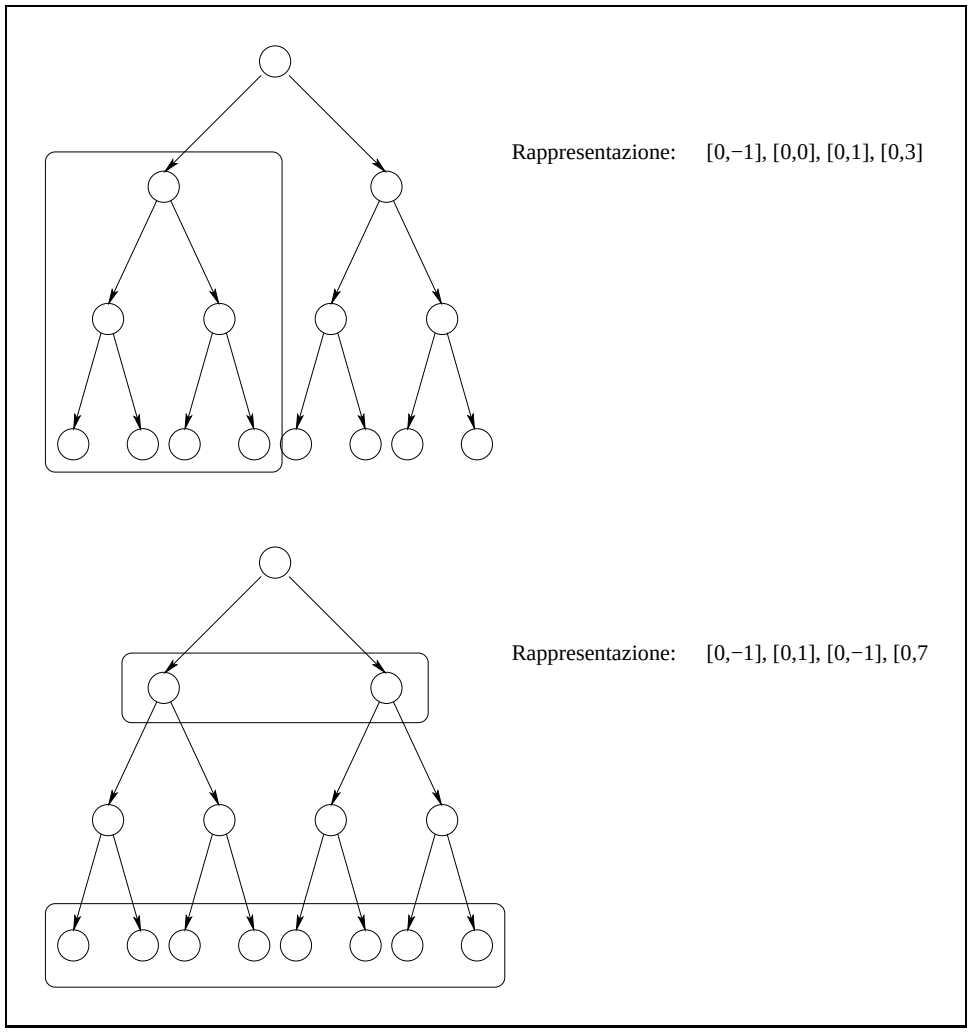


Figura 4.1: Esempi di rappresentazioni di alcuni insiemi.

ore dei due indici utilizzati per identificare un processore virtuale. Per comprendere meglio il problema consideriamo il seguente frammento di codice:

```
VP l=0, a=0  f(in l out ...);
```

nel quale si definisce una semplice elaborazione sulla radice, che utilizza come parametro di input l'indice del livello.
Utilizzando la rappresentazione vista, il processore virtuale (0,0) viene riferito tramite un unico indice (i). Una volta effettuato questo tipo di mapping

non è più possibile tornare indietro e riottenere i due valori di partenza. Di conseguenza il valore di l non può essere calcolato e la funzione non viene eseguita.
Se estendiamo questo discorso al caso del secondo indice usato per l'identificazione, è facile concludere come tale soluzione sia impraticabile e non possa essere adottata per l'implementazione della nuova topologia.

Dal momento che entrambe le possibilità analizzate comportano dei problemi, la scelta finale è stata di implementare una nuova classe che permette di definire le operazioni necessarie per la rappresentazione e la gestione dei processori virtuali ad albero. La classe è stata chiamata `TopologyTreeGroup` (A.6).
Viene definita come template rispetto a due parametri: il numero di livelli e l'arietà dell'albero. La rappresentazione dei processori virtuali viene realizzata secondo lo schema descritto precedentemente per la soluzione 1; questo risulta essere molto espressivo e permette di identificare diversi tipi di insiemi in maniera molto compatta.
Sono stati definiti particolari operatori per l'intersezione, la disgiunzione, la cardinalità e la relazione "contenuto in".

L'introduzione di questa classe comporta la defizione di un nuovo insieme di templates adatto alla gestione del tipo di rappresentazione scelto. Le nuove classi dovranno essere integrate all'interno della libreria ASSIST; in particolare gli oggetti utilizzati per gestire il mapping dei processori virtuali, gli attributi partizionati e le elaborazioni parallele dovranno essere sostituiti con quelli nuovi, che forniscono il supporto a run-time specifico per la nuova topologia.

La gestione della distribuzione dei processori virtuali sui VPM viene realizzata tramite la classe `VPMapperTree` (A.7). Anche questa viene definita come un template rispetto agli stessi parametri della classe `TopologyTreeGroup`. Il metodo principale è quello che permette di mappare un insieme di processori virtuali su un determinato VPM (metodo `distribute`). Lo schema generale di funzionamento prevede di associare ad ogni processo un sottoinsieme di nodi dell'albero in modo tale che questi

rappresentino un sottoalbero della struttura originale. Questo garantisce di avere una forte località dei dati, con pochi accessi fatti in remoto verso altri processi.
Il parametro principale utilizzato per la divisione in sottoinsiemi è costituito dal grado di parallelismo del Parmod: esso rappresenta il numero di VPM presenti nella rete di processi. La procedura di distribuzione opera, in base a tale valore, nel seguente modo (figura 4.2). Si determina il livello dell'albero il cui numero di nodi corrisponde al grado di parallelismo specificato (*l* figura 4.2a). I nodi al di sopra di tale livello vengono partizionati, livello per livello, ed assegnati in maniera ciclica ai VPM (figura 4.2b), mentre gli altri vengono raggruppati in n sottoalberi, ognuno con radice in uno dei nodi del livello *l*, e ciascuno è assegnato ad un VPM (figura 4.2c). Il risultato finale dell'intero procedimento è mostrato in figura 4.2d. Ad ogni VPM viene associato un sottoinsieme di elementi che rappresenta un particolare sottoalbero: questo permette di mantenere molte relazioni padre-figlio all'interno dello stesso nodo e quindi di ottene un'alta località dei dati. Gli accessi remoti sono limitati (figura 4.2d, linee tratteggiate) e vengono effettuati solo per pochi elementi dell'albero.
All'aumentare del grado di parallelismo del modulo diminuisce la dimensione degli insiemi di nodi associati ad ogni VPM, quindi aumenta il numero di accessi remoti. Per cercare di ottenere un buon compromesso tra numero di accessi remoti e numero di elementi associati ad ogni VPM, per default il grado di parallelismo del Parmod viene settato uguale al numero di nodi del livello intermedio dell'albero.

Le forme di parallelismo che vengono privilegiate da questo schema di mapping sono quelle di tipo data parallel, nelle quali ogni nodo esegue un certo insieme di tasks sui propri dati locali, senza effettuare accessi agli elementi degli altri (ad esempio elaborazioni indipendenti associate ai vari livelli dell'albero).
Per altre forme di parallelismo che sfruttano maggiormente la struttura ad albero, è importante limitare il numero di richieste remote ed accedere il più possibile allo stato locale assegnato al VPM (ad esempio tramite

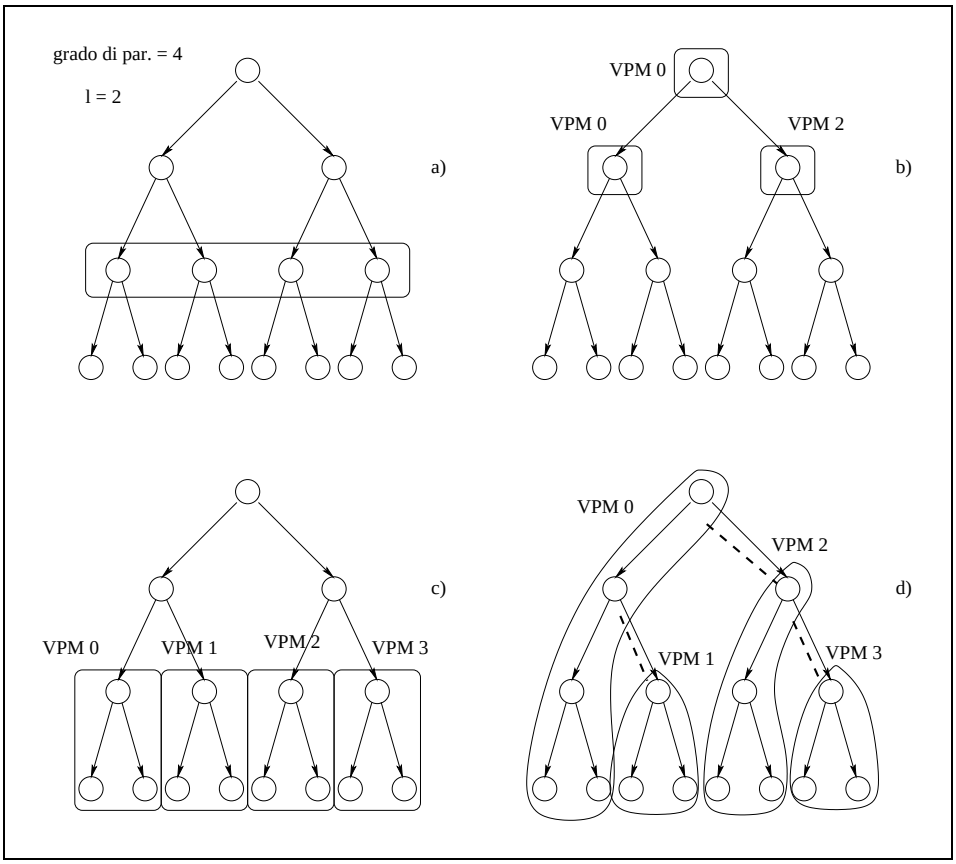


Figura 4.2: Funzionamento del metodo `distribute`.

elaborazioni distinte associate a diversi sottoalberi).

4.3.2 Gestione dello stato distribuito

Per ogni attributo definito nel Parmod, ogni VPM crea un'istanza del gestore della memoria distribuita (`SMU`, *State Management Unit*), che si occupa di allocare la memoria per lo stato locale e di richiedere ed effettuare il fetch dello stato remoto. L'insieme delle `SMU` istanziate per un certo attributo fornisce il supporto per la gestione dello stato distribuito.
I nuovi attributi partizionati utilizzati per la topologia ad albero vengono rappresentati utilizzando la classe `partTreeAttr` (A.8), derivata dalla classe `SMU`, dalla quale eredita le principali funzionalità.
Grazie ad essa ogni VPM determina la parte di stato di sua competenza,

calcolata a seconda dei processori virtuali gestiti e alla regola di mapping utilizzata (metodo `init`):

- Mapping sulla radice: l'intero stato deve essere gestito dal VPM 0 perché detiene il controllo del processore virtuale radice.
- Mapping sulle foglie: lo stato viene suddiviso in parti uguali, ognuna gestita da un VPM diverso in base all'insieme dei processori virtuali foglia che gli sono stati assegnati.
- Mapping sull'albero: ad ogni VPM compete una diversa parte dello stato, calcolata considerando il numero di processori virtuali gestiti.

Per quanto riguarda gli attributi partizionati su tutto l'albero è importante definire un metodo per distribuire gli elementi tra tutti i processori virtuali. Questo influenza il calcolo della parte di stato associata ad un certo VPM.
Il metodo utilizzato deve tener conto della rappresentazione che viene fatta all'interno della libreria ASSIST degli attributi. Ogni riferimento ad una variabile di stato viene effettuata tramite la classe `TopologyGroup`, specificando un determinato intervallo per ogni dimensione; in questo modo è possibile definire solo porzioni continue dello stato. La possibilità di sostituire tale classe con una definita espressamente per la topologia ad albero non è praticabile per il fatto che questa viene utilizzata pesantemente all'interno del supporto. La soluzione, quindi, consiste nel trovare un metodo di distribuzione che permetta di utilizzare anche per l'albero tale tipo di rappresentazione.
Riprendendo l'esempio che abbiamo fatto in precedenza per mostrare il comportamento dello schema di mapping, notiamo come gli elementi associati ad un certo VPM possano essere identificati tramite una visita in profondità dell'albero (figura 4.2d).
Questa semplice constatazione ci suggerisce di utilizzare un metodo che distribuisca gli elementi dello stato secondo tale schema di visita (figura 4.3).

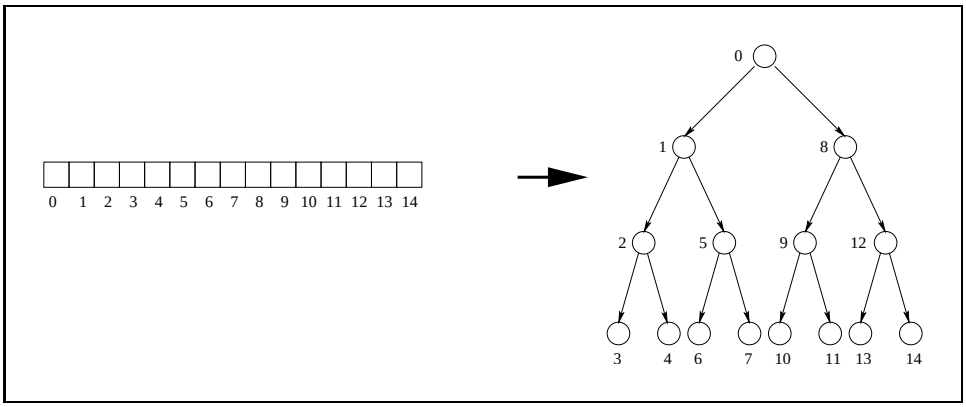


Figura 4.3: Distribuzione degli elementi tra i processori virtuali.

In questo modo lo stato associato ad un certo VPM costituisce una porzione continua dell'intero attributo ed è possibile utilizzare la classe `TopologyGroup` per definire i riferimenti.

Una volta gestita la rappresentazione degli attributi partizionati, è importante fornire il supporto per l'accesso agli elementi. Per tale scopo è stata introdotta la classe `TreeAttr` (A.9), che permette di definire un attributo come parametro per una elaborazione. L'accesso ai dati viene effettuato specificando lo stencil utilizzato per indicizzare gli elementi.
È possibile identificare la parte di stato richiesta per l'esecuzione (metodi `identify` e `discern`), e nel caso in cui questa sia remota, impostare un messaggio di fetch da inviare al VPM interessato. In seguito, al momento in cui vengono preparati tutti i valori (metodo `prepare`), viene effettuato il vero e proprio recupero dei dati.
L'utilizzo di stencil statici e dinamici permette di definire particolari accessi agli elementi. Oltre a quelli già presenti nella libreria (costante, variabile, intervallo, indice della topologia array) ne sono stati aggiunti altri (A.10):

- Indice del livello: si utilizza quando viene impiegato, come parametro di una elaborazione, l'indice della topologia che identifica il livello del processore virtuale.

- Indice del nodo: viene utilizzato quando, come parametro di una

elaborazione, si specifica l'indice che identifica un processore virtuale tra quelli appartenenti al suo stesso livello.

- Macro: viene utilizzato per gestire l'accesso agli elementi locali e remoti effettuato tramite le macro `NODE_INDEX`, `CHILD_INDEX` e `PARENT_INDEX`.

4.3.3 Esecuzione delle elaborazioni parallele

Ogni elaborazione parallela definita nella sezione dei processori virtuali del Parmod viene rappresentata, all'interno della libreria ASSIST, come una lista di tasks che devono essere eseguiti. Nel momento in cui una elaborazione viene attivata, vengono effettuate le seguenti operazioni: per ogni task si identificano i dati locali e remoti necessari all'esecuzione; una volta recuperati tutti i valori viene attivato il task per ognuno dei processori virtuali ai quali è stato assegnato; infine si procede con il commit dei cambiamenti effettuati sullo stato e la scrittura dei risultati sugli streams di output.

Il supporto per la topologia ad albero viene realizzato implementando due nuove classi, utilizzate per gestire le elaborazioni e supportare il tipo di rappresentazione scelta per i processori virtuali: `ProcTree` e `ProcTreeList` (A.11). La prima memorizza le informazioni sul generico task da eseguire e sui processori virtuali associati con tale esecuzione. La seconda permette di gestire la lista dei tasks che fanno parte di una elaborazione parallela. Grazie ad essa è possibile recuperare i dati necessari per l'esecuzione, effettuare il commit dello stato locale ed attivare i singoli task.

4.3.4 Politiche di distribuzione e di collezione

Il supporto alle nuove politiche di distribuzione e di collezione è stato implementato introducendo cinque nuove classi: `ScatterRoot_Poly`, `ScatterLeaves_Poly` e `ScatterTree_Poly` (A.12), `CollRoot_Poly` e `CollLeaves_Poly` (A.13).

Le prime tre forniscono il supporto alle nuove strategie di distribuzione.

`ScatterRoot_Poly` supporta la distribuzione di un dato verso la radice dell'albero. Dal momento che tale processore virtuale è associato al VPM 0, a questo viene inviato il dato completo, mentre agli altri VPM un dato nullo. `ScatterLeaves_Poly` viene utilizzata per gestire la distribuzione di un dato verso le foglie dell'albero. Il dato è suddiviso in n parti, della stessa dimensione, ed ognuna viene inviata ad un VPM. `ScatterTree_Poly` permette di distribuire un dato verso tutti i processori virtuali. Questo è ancora suddiviso in n parti, ma la loro dimensione varia in base al numero di processori virtuali associati ad ogni VPM.

Le altre classi supportano le nuove strategie di collezione. `CollRoot_Poly` viene utilizzata quando la radice dell'albero invia il risultato su uno stream (interno o di output); il suo funzionamento è identico a quello della strategia **from ANY**. A differenza di questa, avendo controllato in precedenza il corretto uso dello stream, siamo sicuri che il dato ricevuto provenga dal processore virtuali radice, quindi dal VPM 0. La seconda permette di raccogliere i dati prodotti dai processori virtuali foglia ed aggregarli per restituire un unico risultato.

CAPITOLO 5

Risultati sperimentali

Una fase molto importante nello sviluppo e nell'integrazione di nuove funzionalità all'interno di un sistema parallelo è la valutazione delle prestazioni, in termini di performance ed overhead dell'implementazione. Queste valutazioni vengono effettuate testando un certo campione di applicazioni, stimando la loro risposta rispetto al *tempo di completamento*, all'*efficienza* e alla *scalabilità* e calcolando se sono stati introdotti dei ritardi rispetto alla versione precedente.

In questo capitolo viene presentato il campione di programmi utilizzati per valutare le prestazioni della topologia ad albero e del Parmod, analizzando i risultati ottenuti.

5.1 Ambiente di lavoro

I tests presentati sono stati eseguiti utilizzando Pianosa [25], il cluster del laboratorio di architetture parallele del nostro dipartimento. Pianosa è costituito da 33 nodi generici più uno utilizzato unicamente per l'accesso e le operazioni di amministrazione. Tutti hanno la stessa configurazione:

- Processore Intel Pentium III 800 Mhz con 32 Kb di cache L1.

- 1GB di memoria.
- Due hard disk da 18 GB.
- Tre schede di rete Ethernet Pro 100.

Il software installato e di interesse per le prove è il seguente:

- Sistema operativo Redhat 8.0, con kernel 2.4.18-27rlx4.
- gcc 3.2-7.
- ACE 5.3 e TAO 1.3.
- ASSIST 1.2.1.

5.2 Valutazione delle prestazioni

Al fine di valutare le prestazioni di una applicazione è necessario disporre di un meccanismo di profiling che permetta di stimare il tempo di esecuzione delle varie componenti.
Il compilatore ASSIST permette di attivare il supporto per il profiling fornito dalla libreria tramite l'opzione -P [13]. Per ogni nodo della rete di processi viene creato un file con estensione `prf` contenete le principali informazioni sul tempo di esecuzione: istante nel quale è iniziata la computazione (calcolato a partire dall'inizio dell'intera esecuzione), tempo speso in sincronizzazioni e comunicazioni, tempo speso per il calcolo.

Il tempo di completamento è stato stimato considerando il tempo riportato dal nodo OSM nel momento in cui il dato viene scritto sullo stream di output del Parmod. Tale valore è stato utilizzato per una prima valutazione delle prestazioni e per calcolare l'efficienza e la scalabilità. Queste ultime sono state valutate nel seguente modo [10, 27]:

- *Efficienza*:

$$\varepsilon_c = \frac{T_{c-id}^{(n)}}{T_c^{(n)}}$$

- *Scalabilità*:

$$s_c = \frac{T_c^{(1)}}{T_c^{(n)}} = \varepsilon_c \cdot n$$

dove

- n identifica il numero dei PE, gli esecutori. Nel nostro caso sono rappresentati dai VPM.
- $T_c^{(n)}$ rappresenta il tempo di completamento utilizzando n PE.
- $T_c^{(1)}$ rappresenta il tempo di completamento nel caso sequenziale, con un solo PE.
- $T_{c-id}^{(n)}$ rappresenta il tempo di completamento nel caso ideale, dato dal tempo di completamento sequenziale diviso il numero di PE:

$$T_{c-id}^{(n)} = \frac{T_c^{(1)}}{n}$$

5.3 Tipologia delle prove

Il primo scopo delle prove effettuate è stato quello di controllare il corretto funzionamento della topologia ad albero in alcuni semplici casi di utilizzo delle sue funzionalità; in particolare è stato valutato il controllo del parallelismo, la gestione dei processori virtuali e delle elebarazioni ad essi associate. Successivamente, sono state analizzate le prestazioni ottenute dalle varie applicazioni al fine di valutare la performance raggiunta, in termini di tempo di completamento, efficienza e scalabilità, e l'overhead introdotto nella nuova versione del Parmod rispetto a quella originale.

Le applicazioni utilizzate per effettuare le prove sono le seguenti:

- *Massimo*: i processori virtuali cooperano ed interagiscono per effettuare il calcolo del valore massimo di un vettore.
- *Data parallel*: ogni processore virtuale esegue, indipendentemente dalle altre, una elaborazione sulla parte di stato ad essa assegnata.

Sono state realizzate diverse versioni di questi programmi, sia per la topologia ad albero che per quella array.
Data la grana computazionale molto piccola di tali applicazioni, per poter meglio analizzare il loro comportamento al variare del carico di lavoro è stato associato un peso ad ogni processore virtuale, espresso da un ciclo di iterazioni con chiamate a funzioni trigonometriche.
Inoltre, per differenziare ulteriormente i valori ottenuti e migliorare l'analisi dei risultati, ogni prova è stata effettuata variando i principali parametri che ne influenzano l'esecuzione:

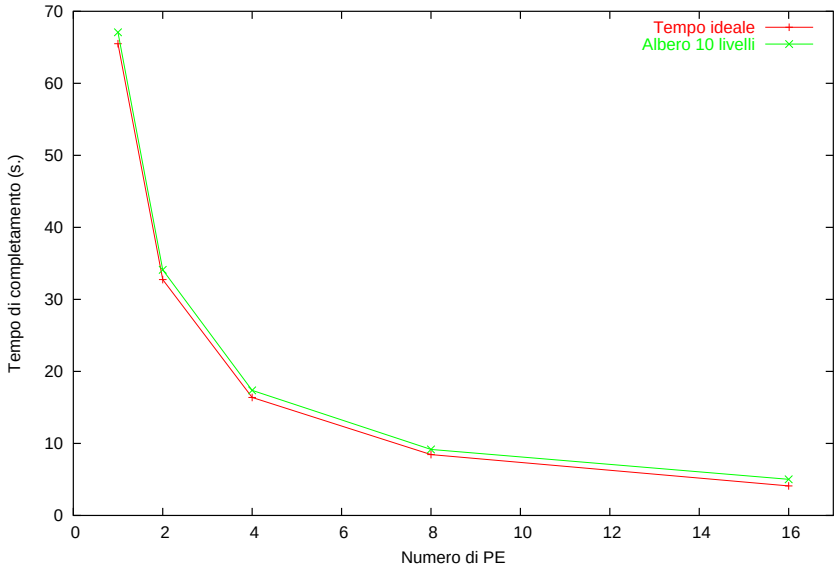
1. Livelli dell'albero: il numero di livelli dell'albero è legato alla sua dimensione e al numero di processori virtuali che ogni PE deve gestire.

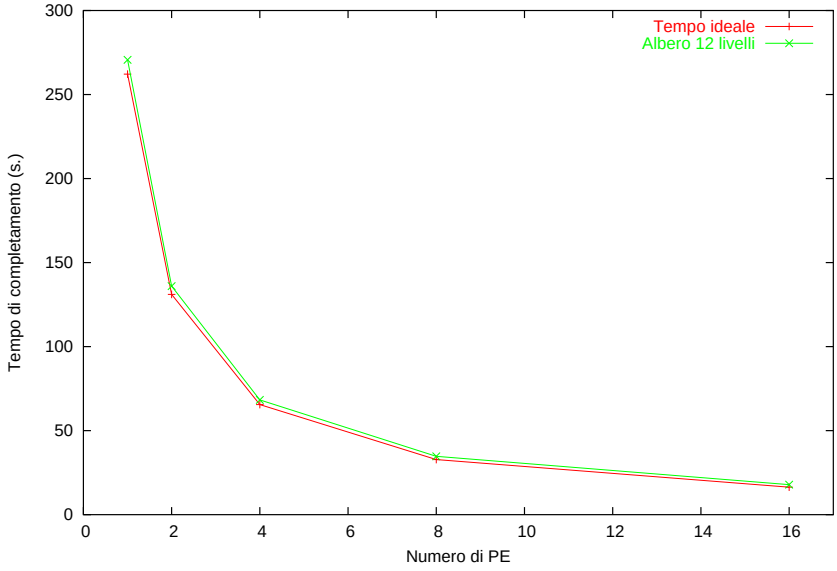
2. Dimensione dell'input: la dimensione del dato di input dipende principalmente dal numero di elementi dell'albero e dal tipo di distribuzione utilizzata nella sezione di input del Parmod.

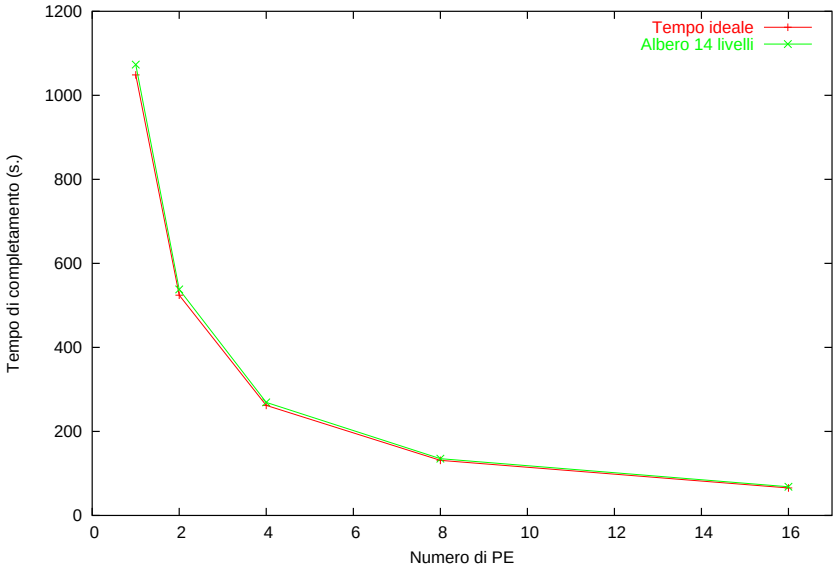
3. Numero di PE: variando il numero di esecutori è possibile stimare l'efficienza e la scalabilità dell'implementazione.

5.3.1 Massimo

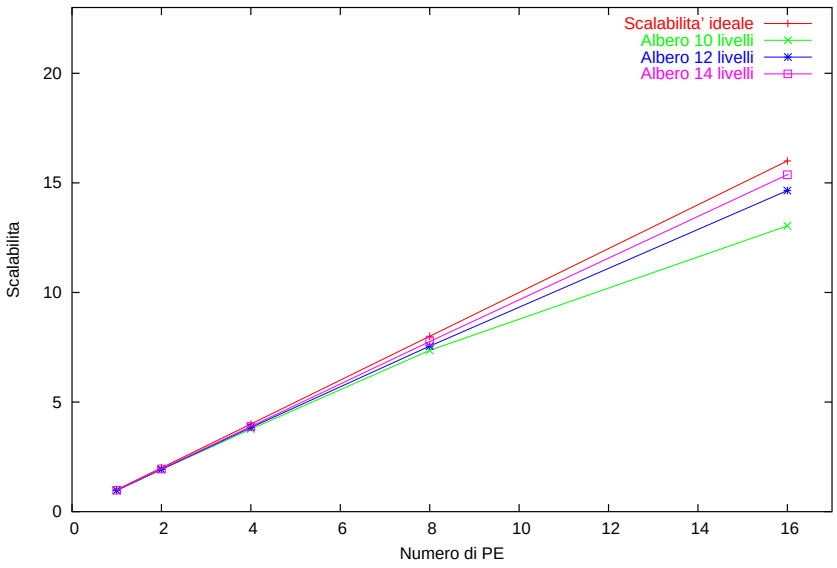
Il problema di riferimento preso in considerazione è il seguente: calcolare il valore massimo di un vettore. Il tipo di problema si presta bene ad essere parallelizzato utilizzando un algoritmo che sfrutta una struttura ad albero per l'esecuzione (B.1.1).
La computazione viene effettuata in diverse fasi, in ognuna delle quali lavorano soltanto i nodi di un livello, che calcolano il massimo dei valori associati ai figli. Il vettore da ordinare ricevuto in input viene trasmesso alle foglie e a partire da queste inizia l'elaborazione. Il risultato finale viene calcolato dal nodo radice, che lo invia sullo stream di output.
I risultati delle prove effettuate vengono mostrati nei grafici che seguono.

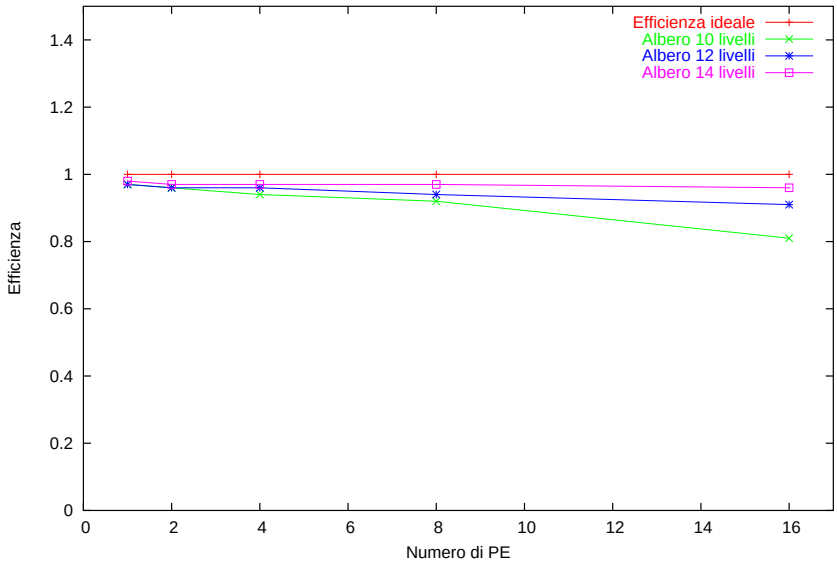

Tempo ideale
Albero 10 livelli
Tempo di completamento (s.)
Numero di PE
0
10
20
30
40
50
60
70
0
2
4
6
8
10
12
14
16


Tempo ideale
Albero 12 livelli
Tempo di completamento (s.)
Numero di PE
0
50
100
150
200
250
300
0
2
4
6
8
10
12
14
16



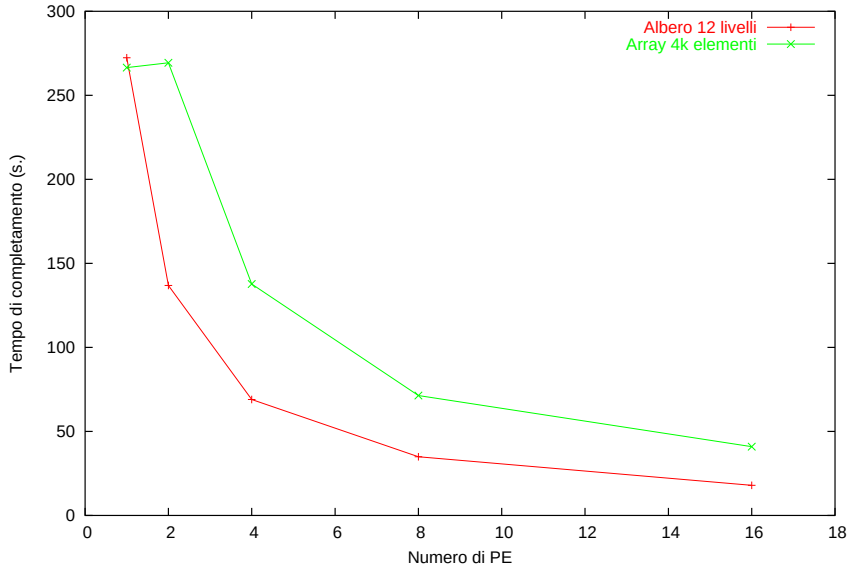
Analizzando questi primi risultati possiamo notare come le prestazioni dell'applicazione migliorino all'aumentare del numero di nodi dell'albero; l'andamento della curva relativa al tempo di completamento per la versione con topologia ad albero tende a schiacciarsi sempre di più verso quella del tempo ideale, segno che la performance tende ad aumentare. Tale andamento è confermato anche dai risultati di scalabilità ed efficienza.





Le due curve mostrano ancora meglio l'andamento delle prestazioni dell'applicazione e risulta evidente come, all'aumentare del numero di processori virtuali, efficienza e scalabilità migliorino, tendendo ai valori idelai. In particolare, nel caso con 16 PE, si passa da una efficienza pari all'80%, relativa all'albero con 10 livelli, fino ad una del 98%, relativa all'albero con 14 livelli.
Nel complesso, quindi, la performance dell'applicazione testata risulta molto buona.

Per una valutazione più completa e differenziata delle prestazioni della topologia è stata implementata una versione dell'algoritmo tramite un Parmod array, il quale simula l'esecuzione a livelli descritta precedentemente (B.1.2). La prima differenza con la versione ad albero è nel numero processori virtuali che devono essere gestiti: la metà rispetto al caso visto in precedenza.
I risultati hanno mostrato il seguente andamento per entrambe le versioni.



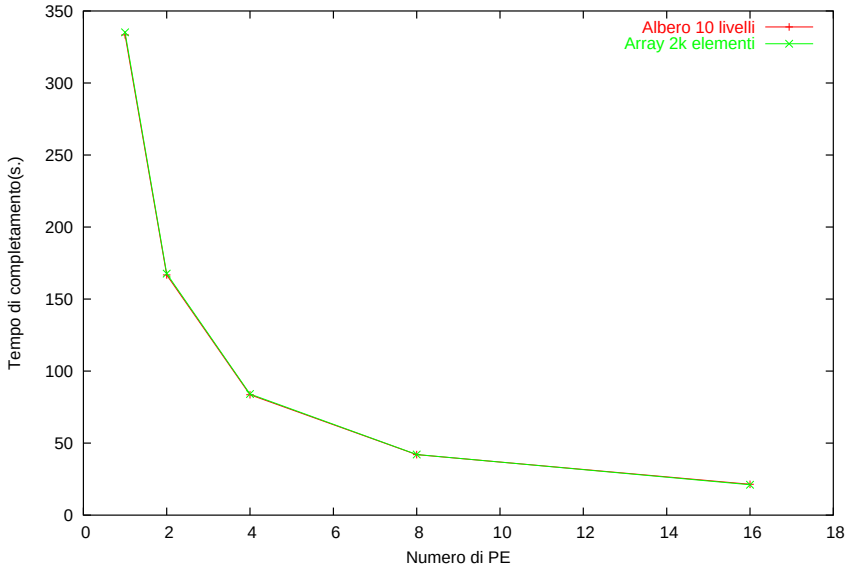
Il confronto è stato fatto tra i tempi di completamento delle rispettive implementazioni. Come si può notare le prestazioni della versione che utilizza la topologia ad albero sono molto migliori rispetto a quelle con topologia array (circa il 50% più veloce). Il motivo principale di un tale andamento è dovuto alla differenza di espressività esistente tra le due topologie. Nel caso di topologia ad albero ogni processore virtuale riesce ad accedere allo stato del figlio utilizzando le macro messe a disposizione del linguaggio di coordinamento, mentre con la topologia array l'accesso ai singoli elementi può essere fatto solo utilizzando semplici espressioni (costante o variabile di iterazione). A causa di questo, per poter accedere ad un elemento particolare, il cui indice deve essere calcolato a tempo di esecuzione, è necessario specificare l'interno vettore come parametro di input dell'elaborazione. Di conseguenza, aumentando il numero di PE lo stato diventa sempre più distribuito e sono necessari più accessi remoti per ricostruire la sua struttura completa, aumentando significativamente l'overhead.

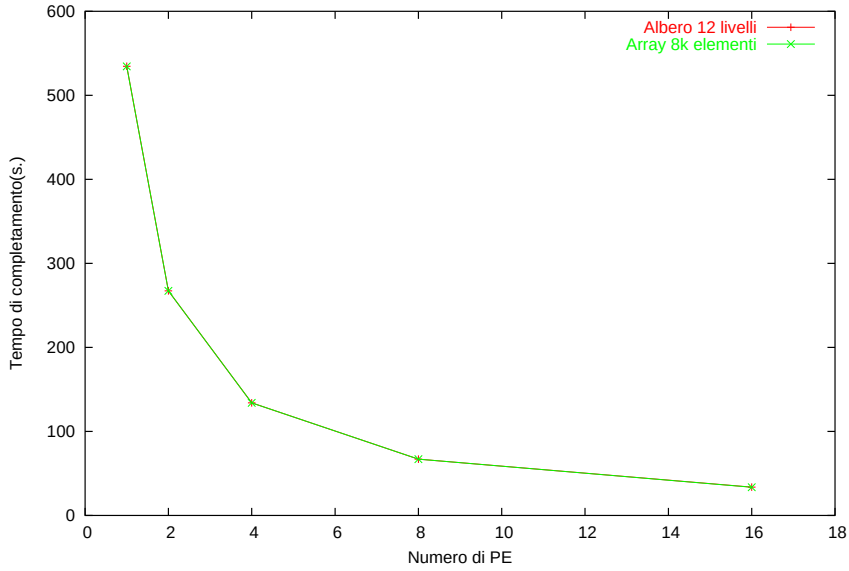
5.3.2 Data parallel

L' algoritmo utilizzato è una semplice applicazione di tipo data parallel: il dato ricevuto sullo stream di input viene distribuito tra i processori

virtuali del Parmod; ognuno di essi esegue, indipendentemente dagli altri, un'elaborazione per il dato locale.
Sono state realizzate due diverse implementazioni dell'algoritmo, una con topologia ad albero (B.2.1) ed una con topologia ad array (B.2.2), e sono state effettuate due distinte serie di prove. Con la prima è stata valutata la performance della nuova topologia rispetto a quella array, in un caso in cui entrambe sfruttano la stessa espressività, mentre con la seconda abbiamo confrontato il tempo di completamento dell'implementazione che utilizzata la topologia array, compilando i sorgenti con la versione originale del compilatore e con quella modificata, per stimare l'overhead introdotto.

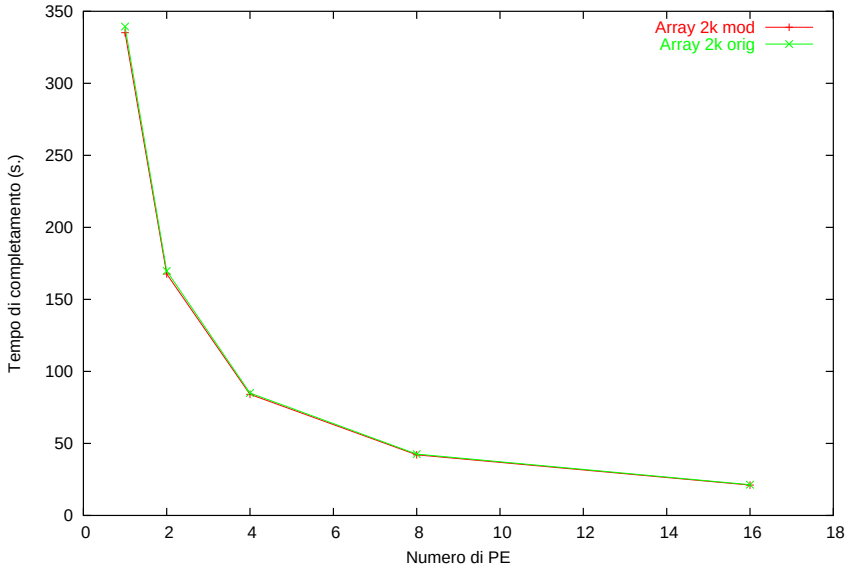
La prima serie di prove ha fornito i seguenti risultati.

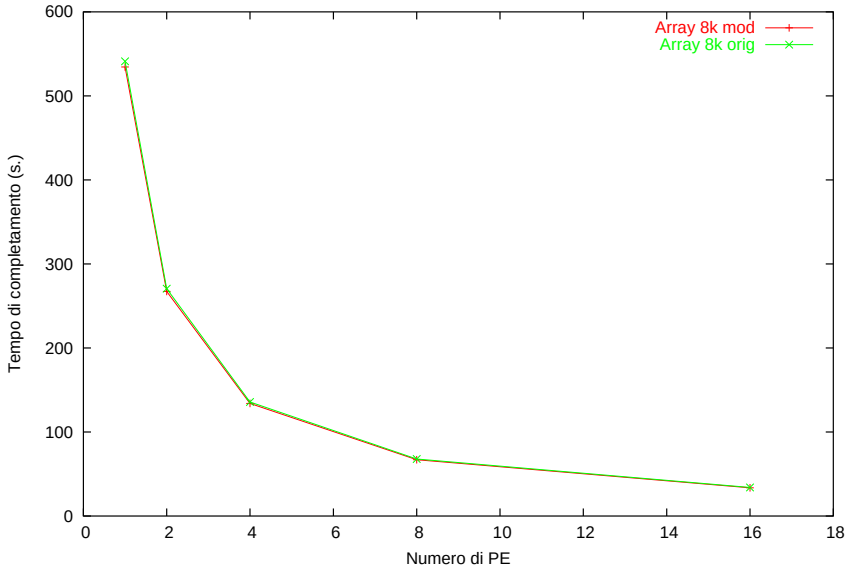




Con entrambe le implementazioni otteniamo gli stessi identici risultati. Una prima conseguenza di ciò è data dal fatto che, nel caso in cui la topologia ad albero possa essere utilizzata al posto di quella array, le prestazioni che si ottengono sono paragonabili a quest'ultima; in pratica non si sono introdotti overhead. La seconda conclusione conferma quanto affermato in precedenza con il confronto tra le due topologie: le prestazioni di quella ad albero sono migliori esclusivamente per il fatto che la sua potenza espressiva permette di ottimizzare gli accessi allo stato; a parità di potenza espressiva entrambe le topologie offrono lo stesso grado di performance.

La seconda serie di prove ha confermato le aspettative ed ha offerto gli stessi risultati del caso precedente.





I risultati confermano il fatto che l'implementazione della nuova topologia e le modifiche apportate al compilatore non hanno influenzato le prestazioni del Parmod e non hanno introdotto ritardi.

5.4 Analisi conclusiva

Le serie di prove che abbiamo effettuato ci hanno permesso di valutare le prestazioni della nuova topologia sia nei confronti dell'efficienza e della scalabilità che nei confronti dell'overhead introdotto dall'implementazione. I risultati possono essere riassunti come segue:

- L'implementazione funziona correttamente rispettando i requisiti iniziali del progetto.
- A parità di espressività, la nuova topologia può essere utilizzata al posto di quella array con prestazioni paragonabili a quest'ultima.
- Quando si utilizza una espressività maggiore, rispetto a quella fornita dalla topologia array, si riesce a sfruttare al meglio le caratteristiche dell'albero ottenendo un buon comportamento dell'applicazione.
- Nel complesso la performance della nuova forma di parallelismo risulta essere buona ed in linea con i risultati ottenuti dall'attuale versione di ASSIST.

CAPITOLO 6

Conclusioni

La tesi ha permesso di lavorare ed utilizzare un ambiente di sviluppo che si è dimostrato completo ed espressivo per la realizzazione di programmi paralleli e distribuiti.

Lo scopo principale che ci siamo prefissati è stato quello di integrare all'interno del costrutto parallelo Parmod il supporto per forme di parallelismo strutturate ad albero e valutare le prestazioni offerte. Il percorso di sviluppo seguito ha permesso di acquisire una certa pratica nello studio di un sistema e dei passi necessari per l'estensione delle sue caratteristiche.
Dopo una prima fase di analisi del modello di programmazione di ASSIST, il lavoro si è concentrato interamente sul compilatore, ponendoci due obiettivi:

1. Estendere il linguaggio di coordinamento con i costrutti necessari per esprimere il parallelismo ad albero.

2. Fornire il supporto a run-time per le funzionalità introdotte.

Lo studio effettuato sulla struttura del compilatore è stato molto interessante ed ha permesso di valutare nel dettaglio i singoli passi eseguiti:

analisi sintattica e semantica, generazione del codice intermedio, creazione degli eseguibili.
L'implemenatazione ha interessato ogni fase del processo di compilazione: in ognuna sono state prese le scelte necessarie per estendere il supporto alla topologia ad albero, evitando di apportare un numero eccessivo di modifiche.

Terminata la parte implementativa è stato possibile valutare le prestazioni della nuova topologia. I tests effettuati hanno mostrato un buon comportamento del supporto a run-time, offrendo un alto grado di efficienza e scalabilità. L'espressività fornita dalla topologia ad albero può essere paragonata a quella fornita dalla topologia array, ed in certi casi è anche maggiore. Tale estensione della potenza espressiva del linguaggio di coordinamento si traduce in un miglioramento delle prestazioni delle applicazioni che la sfruttano e quindi dell'intero ambiente di sviluppo di ASSIST.

Alcuni aspetti legati alle funzionalità del Parmod non sono stati trattati in fase di sviluppo, in particolare la gestione e l'interazione con gli oggetti esterni. Oltre a questi, alcuni dei possibili sviluppi futuri potrebbero prevedere l'introduzione di alberi con un numero qualsiasi di livelli, in modo da adattare l'esecuzione del modulo alla dimensione dell'input, oppure estendere il linguaggio di coordinamento per il supporto di un costrutto in grado di selezionare direttamente un sottoalbero, aumentando ulteriormente la potenza espressiva della topologia.

La realizzazione di tutte le funzionalità e le caratteristiche descritte è stata effettuata "offline" rispetto al gruppo che lavora allo sviluppo di ASSIST. A tal proposito ho ricevuto una proposta per integrare a tutti gli effetti la topologia ad albero all'interno dell'attuale versione dell'ambiente di sviluppo.

Bibliografia

[1] ACE. The Adaptive Communication Environment homepage. URL: http://www.cs.wustl.edu/schmidt/ACE-papers.html, 2004.

[2] M.Aldinucci, S. Campa, P.Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi e C. Zoccolo. ASSIST Demo: A High Level, High Performance, Portable, Structured Parallel Programming Environment ar Work. In *Euro-Par 2003 Parallel Processing*, pagine 1295-1300, 2003.

[3] M. Aldinucci, S.Campa, P.Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L.Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi e C. Zoccolo. A framework for experimenting with structured parallel programming environment design. In *Procs. od Intl. Conference ParCo 2003 Parallel Computing*, Dresden, Germany, 2003.

[4] M. Aldinucci, S.Campa, P.Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L.Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi e C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In *Euro-Par 2003 Parallel Processing*, pagine 712-721, 2003.

[5] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi e C. Zoccolo. Components for High

performance Grid programming in the Grid.it project. In *Intl. Workshop on Component Models and Systems for Grid Applications*, Saint-Malo, France, 2004. ACM ICS 2004.

[6] M. Aldinucci, S. Campa, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati e C. Zoccolo. Targeting Heterogeneous Architecture in ASSIST: Experimental Results. In *Euro-Par 2004 Parallel Processing*, pagine 638-643.

[7] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi e C. Zoccolo. ASSIST as a Research Framework for High performance Grid Programming Environment. Technical Report TR-04-09, Dipertimento di Informartica, Università di Pisa, 2004.

[8] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi e C. Zoccolo. WP8 Component Model, Grid.it project, 2004.

[9] ASSIST homepage. URL: http://www.di.unipi.it/groups/architecture/Assist.htm, 2005.

[10] F. Baiardi, A. Tomasi e M. Vanneschi. *Architettura dei Sistemi di Elaborazione, volume I e II*. Franco Angeli, 1987.

[11] R. Baraglia, M. Danalutto, D. Laforenza, S. Orlando, P. Palmieri, P. Pesciullesi, R. Perego e M. Vanneschi. AssistConf: a Grid configuration tool for the ASSIST parallel programming environment.

[12] P. Ciullo, M. Danelutto, S. Magini, L. Potiti, R. Ravazzolo, D. Guerri e M. Vanneschi. ASSIST-CL User Manual. ASI-PQE2000, 2002.

[13] P. Ciullo, S. Magini, L. Potiti e C. Zoccolo. Tutorial ASSIST-Cl ver. 1.2, 2004.

[14] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[15] Murray Cole. Bringing skeletons ot of closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comp.* 30(3):389-406, 2004.

[16] Murray Cole. eSkel homepage. URL: http://homepages.inf.ed.ac.uk/mic/eSkel, 2004.

[17] M. Coppola, M. Pasquali, L. Presti e M. Vanneschi. En Experiment with High Performance Components for Grid Applications.

[18] CORBA. CORBA homepage. URL: http://www.corba.org, 2004.

[19] M. Danelutto, F. Pasqualetti e S. Pelagatti. Skeletons for Data Parallelism in p3l. In *Euro-Par '97 Parallel Processing*, pagine 619-628, 1997.

[20] E. Gamma, R. Helm, R. Johnson e J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[21] Grid.it project homepage. URL: http://www.grid.it, 2004.

[22] S. Magini, P. Pesciullesi e C.Zoccolo. Parallel Software Interoperability by Means of CORBA in ASSIST Programming Environment. In *Euro-Par 2004 Parallel Processing*, pagine 679-688.

[23] Object Management Group. CORBA Component Model version 3.0 specification, 2002, URL: http://www.omg.org.

[24] Object Management Group. The Common Object Request Broken: Architecture and Specification, 2002, Minor revision 2.4.1, URL: http://www.omg.org.

[25] Pianosa homepage. URL: http://pianosa.di.unipi.it, 2004.

[26] Skeletons homepage. URL: http://homepages.inf.ed.ac.uk/mic/Skeletons/index.html, 2004.

[27] Marco Vanneschi. *Appunti di Architettura degli Elaboratori II*. SEU, Pisa, 2001.

[28] Marco Vanneschi. ASSIST: an Environment for Parallel and Distributed Portable Applications. Technical Report TR-02-07, Dipartimento di Informatica, Università di Pisa, 2002.

[29] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Comp.*, 28(12):1709-1732, 2002.

[30] W3C. Web Services homepage. URL: http://www.w3c.org/2002/ws, 2003.

APPENDICE A

Codice prodotto

A.1 Dichiarazione della topologia

Produzioni utilizzate per riconoscere la dichiarazione della topologia ad albero; inserite all'interno della grammatica del parser.

```
tree_topology:
_TREE_
{
  dimList = new AST_Iterator<int *>();
  varList = new AST_Iterator<AST_LocalDecl *>();
}
levels_dimension children_dimension _ID_ _SEMICOLON_
{
  int levels;
  int children;
  int middle;

  dimList->First();
  levels = *(dimList->currentItem());
  dimList->Next();
  children = *(dimList->currentItem());
  middle = ((levels % 2) == 0) ? levels / 2 : (levels / 2) + 1;

  AST_Tree * t = new AST_Tree(levels, children);
  t->setName($5->tokenValue);
  t->setLineNum($5->getLineNumber());
  t->setFileName($5->getFileName());

  ((AST_Parmod*) currModule)->setTopologyDecl(t, AST_Parmod::TREE);
  ((AST_Parmod*) currModule)->setArrayIndex(varList);
```

```
  ((AST_Parmod*) currModule)->setParDegree(t->getNodesByLevel(middle));
}
;

levels_dimension:
_LSB_ _ID_ _COLON_ ide_expr _RSB_
{
  AST_BasicType * b = new AST_BasicType(AST_BasicType::LONG);

  b->setName($2->tokenValue);
  b->setLineNum($2->getLineNumber());
  b->setFileName($2->getFileName());

  AST_LocalDecl * lD =
      new AST_LocalDecl(AST_LocalDecl::TOPOLOGY_LEVELS,
                        b->getName(), b);

  if(!currModule->localTable.insertUnique(lD->getName(), lD))
  {
    cerr << "Error in file " << $2->getFileName() << " line ";
    cerr << $2->getLineNumber() << ": already declared name ";
    cerr << $2->tokenValue << endl;
    exit(-1);
  }

  AST_EvalExpr * evExpr = new AST_EvalExpr();
  AST_Value * res = evExpr->eval($4);
  if(res->getValueKind() != AST_Value::INT_VALUE)
  {
    cerr << "Errore indicizzazione con valore non intero" << endl;
    exit(-1);
  }

  int * val = new int();
  *val= res->getValue().lval;
  dimList->addElement(val);
  lD->setRangeValue(*val);
  lD->setArrayPosition(0);
  varList->addElement(lD);
}
;

children_dimension:
_LSB_ _ID_ _COLON_ ide_expr _RSB_
{
  AST_BasicType * b = new AST_BasicType(AST_BasicType::LONG);

  b->setName($2->tokenValue);
  b->setLineNum($2->getLineNumber());
  b->setFileName($2->getFileName());

  AST_LocalDecl * lD =
      new AST_LocalDecl(AST_LocalDecl::TOPOLOGY_CHILDREN,
                        b->getName(), b);

  if(!currModule->localTable.insertUnique(lD->getName(), lD))
  {
    cerr << "Error in file " << $2->getFileName() << " line ";
    cerr << $2->getLineNumber() << ": already declared name ";
    cerr << $2->tokenValue << endl;
    exit(-1);
  }
```

```
  AST_EvalExpr * evExpr = new AST_EvalExpr();
  AST_Value * res = evExpr->eval($4);
  if(res->getValueKind() != AST_Value::INT_VALUE)
  {
    cerr << "Errore indicizzazione con valore non intero" << endl;
    exit(-1);
  }

  int * val = new int();
  *val= res->getValue().lval;
  dimList->addElement(val);
  lD->setRangeValue(*val);
  lD->setArrayPosition(1);
  varList->addElement(lD);
}
;
```

A.2 Classe AST_Tree

Classe utilizzata per rappresentare le informazioni sull'albero raccolte durante il parsing; aggiunta a quelle utilizzate per la rappresentazione interna di un programma.

```
#ifndef _AST_TREE_H
#define _AST_TREE_H

#include "AST_Type.h"
#include "AST_Visitor.h"
#include <cmath>

class AST_Tree: public AST_Type
{
 public:

  AST_Tree(int numLevels, int numChildren):
    AST_Type(AST_Type::TREE_TYPE)
  {
    _levels = numLevels;
    _children = numChildren;
  };

  ~AST_Tree() { };

  int getNumLevels(void)
  {
    return _levels;
  };

  int getNumChildren(void)
  {
    return _children;
  };
```

```
  int getNodesByLevel(int level)
  {
    if(level >= 0)
    {
      double result = pow((double) _children, (double) level);
      return((int) result);
    }
    else
      return(-1);
  }

  virtual void accept(AST_Visitor *visitor)
  {
    visitor->visit_tree(this);
  }

 private:
  int _levels;
  int _children;
};
#endif
```

A.3 Attributi partizionati

Produzioni usate per riconoscere la dichiarazione degli attributi partizionati per l'albero; aggiunte alla grammatica del parser.

```
partitioned_variables_decl:
attribute_declaration
{

...

}
_SCATTER_ part_expr _ONTO_ part_expr _SEMICOLON_
{

...

}
| attribute_declaration  _ONTO_ _ID_ _SEMICOLON_
{

...

}
| attribute_declaration
{
  currAtt = new AST_Attribute(AST_Attribute::PARTITIONED);
  currAtt->setFileName($1->getFileName());
  currAtt->setLineNum($1->getLineNumber());
  currAtt->setName($1->getName());
  currAtt->setType($1);
  isIso = false;
```

```
  if($1->getTypeKind() != AST_Type::ARRAY_TYPE)
  {
    string errMsg = "Only array attribute can be partitioned.";
    AST_Error::instance()->addError(AST_Error::ERROR,
                                    AST_Error::WRONG_VARIABLE_DECL,
                                    $1, errMsg);
  }

  ((AST_Parmod*) currModule)->addAttributeDecl(currAtt,
                                               AST_Attribute::PARTITIONED);
}
_SCATTER_TREE_ part_expr _ONTO_ destination _SEMICOLON_
{
  if(isIso)
  {
    AST_VariableExpr *varExpr = new
        AST_VariableExpr(AST_VariableExpr::LOCAL_VARIABLE,
    ((AST_Parmod *) currModule)->getTopologyDecl());
    currAtt->setMappingRule(AST_Attribute::ISO_TREE, $4, varExpr);
  }
  else
    currAtt->setMappingRule(AST_Attribute::SUBSET_TREE, $4, $6);

  if(((AST_Parmod*) currModule)->getTopologyKind() != AST_Parmod::TREE)
  {
    string errMsg = "Partitioned attribute allowed only for parmod tree";
    AST_Error::instance()->addError(AST_Error::ERROR,
                                    AST_Error::WRONG_VARIABLE_DECL,
                                    $1, errMsg);
  }
}
;

destination:
part_expr
{
  $$ = $1;
}
|
_ID_
{
  $$ = 0;
  isIso = true;

  AST_LocalDecl *lD = currModule->localTable.getElement($1->tokenValue);

  if(lD == NULL)
  {
    cerr << "Error in file " << $1->getFileName() << " line ";
    cerr << $1->getLineNumber() << " variable " << $1->tokenValue;
    cerr << " not declared" << endl;
    exit(-1);
  }
  else
  {
    if(lD->getLocalDecl() != AST_LocalDecl::TOPOLOGY_VAR)
    {
      cerr << "Error in file " << $1->getFileName() << " line ";
      cerr << $1->getLineNumber() << " " << $1->tokenValue;
      cerr << " not declared as topology variable" << endl;
      exit(-1);
    }
```

```
  }
}
;
```

A.4 Macro di indicizzazione

Produzioni utilizzate per riconoscere le macro di indicizzazione dello stato distribuoto; aggiunte alla grammatica del parser.

```
dim_value:
{

...

}
| RValue
{

...

}
| _STAR_
{

...

}
| _STAR_ _ID_
{

...

}
| _NODE_INDEX_
{
  $$ = new AST_MacroExpr(AST_MacroExpr::NODE_IDX, 0);
  $$->setFileName($1->getFileName());
  $$->setLineNum($1->getLineNumber());

  if(!macroUse || (macroUse && !isIso))
  {
    string errMsg = string("Macro NODE_INDEX not allowed");
    AST_Error::instance()->addError(AST_Error::ERROR,
                                    AST_Error::WRONG_EXPRESSION,
                                    $$, errMsg);
  }
}
| _CHILD_INDEX_ _LRB_ const_value _RRB_
{
  AST_Value *cVal = (AST_Value*) $3;
  $$ = new AST_MacroExpr(AST_MacroExpr::CHILD_IDX, cVal->getValue().lval);
  $$->setFileName($1->getFileName());
  $$->setLineNum($1->getLineNumber());

  if(!macroUse || (macroUse && !isInputList) ||
```

```
    (macroUse && isInputList && !isIso))
  {
    string errMsg = string("Macro CHILD_INDEX not allowed");
    AST_Error::instance()->addError(AST_Error::ERROR,
                                    AST_Error::WRONG_EXPRESSION,
                                    $$, errMsg);
  }
  else
  {
    AST_LocalDecl *topoDecl = ((AST_Parmod *) currModule)->getTopologyDecl();
    AST_Tree *topoType = (AST_Tree *) topoDecl->getType();
    int children = topoType->getNumChildren();

    if((cVal->getValue().lval == 0) || (cVal->getValue().lval > children))
    {
      string errMsg = string("Wrong child number for macro CHILD_INDEX");
      AST_Error::instance()->addError(AST_Error::ERROR,
                                      AST_Error::WRONG_EXPRESSION,
                                      $$, errMsg);

    }
  }
}
| _PARENT_INDEX_
{
  $$ = new AST_MacroExpr(AST_MacroExpr::PARENT_IDX, -1);
  $$->setFileName($1->getFileName());
  $$->setLineNum($1->getLineNumber());

  if(!macroUse || (macroUse && !isInputList) ||
    (macroUse && isInputList && !isIso))
  {
    string errMsg = string("Macro PARENT_INDEX not allowed");
    AST_Error::instance()->addError(AST_Error::ERROR,
                                    AST_Error::WRONG_EXPRESSION,
                                    $$, errMsg);
  }
}
;
```

A.5 Classe AST_MacroExpr

Classe utilizzata per memorizzare le informazioni su ogni macro raccolte durante il parsing; aggiunta a quelle utilizzate per creare la rappresentazione interna di un programma.

```
#ifndef _AST_MACRO_EXPR_H_
#define _AST_MACRO_EXPR_H_

#include <string>

class AST_MacroExpr: public AST_Expr
{
 public:
```

```
  typedef enum
  {
    NODE_IDX,
    CHILD_IDX,
    PARENT_IDX
  } idx_kind;

  AST_MacroExpr(AST_MacroExpr::idx_kind kind, int child):
    AST_Expr(AST_Expr::MACRO_IDX), _indexType(kind), _childNum(child) {  }

  AST_MacroExpr::idx_kind getIndexKind()
  {
    return(_indexType);
  }

  int getChildNum()
  {
    return(_childNum);
  }

  virtual string getName()
  {
    return(string(""));
  }

  virtual void accept(AST_Visitor *visitor)
  {
    visitor->visit_macro_expr(this);
  }

 private:
  int _childNum;
  AST_MacroExpr::idx_kind _indexType;
};
#endif
```

A.6 Classe TopologyTreeGroup

Classe utilizzata per rappresentare insiemi di processori virtuali appartenenti alla topologia ad albero; aggiunta alla libreria ASSIST.

```
#ifndef _TOPOLOGY_TREE_GROUP_HPP_
#define _TOPOLOGY_TREE_GROUP_HPP_

#include <iostream>
#include <Range.hpp>
#include <TopologyGroup.hpp>

template<int LEVELS, int CHILDREN> class TopologyTreeGroup
{
 public:

  enum
  {
```

```
  dims = LEVELS
};

Range range[LEVELS];

void clear()
{
  for(int i = 0; i < LEVELS; i++)
    range[i].clear();
}

bool empty()
{
  bool ret = true;

  for(int i = 0; i < LEVELS; i++)
    ret &= range[i].empty() ? true : false;

  return(ret);
}

int cardinality() const
{
  int card = 0;

  for(int i = 0; i < LEVELS; i++)
  {
    if(!range[i].empty())
      card += range[i].cardinality();
  }
  return(card);
}

void firstIndex(Index<LEVELS>& index) const
{
  for(int i = 0; i < LEVELS; i++)
    index.idx[i] = range[i].empty() ? -1 : range[i].b;
}

bool nextIndex(Index<LEVELS>& index)
{
  return(false);
}

bool nextIndex0(Index<LEVELS>& index) const
{
  for(int i = 0; i < LEVELS; i++)
    if(index.idx[i] != -1)
      return((++index.idx[i]) <= range[i].e);

  return(false);
}

bool nextIndexR(Index<LEVELS>& index) const
{
  return(false);
}

static TopologyTreeGroup<LEVELS, CHILDREN> fromIndex(Index<LEVELS>& index)
{
  TopologyTreeGroup<LEVELS, CHILDREN> tmp;
```

```
  tmp.clear();
  return(tmp);
}

bool intersect(const TopologyTreeGroup<LEVELS, CHILDREN>& set)
{
  bool ret = false;
  TopologyTreeGroup<LEVELS, CHILDREN> tmp = *this;

  for(int i = 0; i < LEVELS; i++)
  {
    if(!range[i].empty())
    {
      if(range[i].intersect(set[i]))
        ret = true;
    }
  }
  return(ret);
}

void split(TopologyTreeGroup<LEVELS, CHILDREN>& remain, int elems)
{
  int rest = cardinality() - elems;

  clear();
  remain = *this;

  for(int i = 0; i < LEVELS; i++)
  {
    int rng = range[i].cardinality();

    if(rng > 0)
    {
      if(rest > 0)
      {
        if(rest >= rng)
          remain[i].clear();
        else
        {
          remain[i].bound(rest, range[i].e);
          range[i].bound(range[i].b, rest - 1);
        }
        rest -= rng;
      }
      else
        range[i].clear();
    }
  }
}

bool isWrapped(TopologyTreeGroup<LEVELS, CHILDREN> set)
{
  for(int i = 0; i < LEVELS; i++)
    if(!range[i].empty())
      if(!range[i].isWrapped(set[i]))
        return(false);

  return(true);
}

void buildDiffSet(TopologyTreeGroup<LEVELS, CHILDREN>& set_a,
                  TopologyTreeGroup<LEVELS, CHILDREN>& set_b)
```

```
{
  clear();
  for(int i = 0; i < LEVELS; i++)
  {
    if(set_a[i].empty() && !set_b[i].empty())
      range[i] = set_b[i];
    else if(!set_a[i].empty() && !set_b[i].empty())
    {
      range[i].innerJoin(set_a[i], set_b[i]);

      if(range[i].empty())
        range[i] = set_b[i];
      else
      {
        if(range[i] == set_b[i])
          range[i].clear();
        else
        {
          if(range[i].b <= set_a[i].e)
            range[i].bound(set_a[i].e + 1, set_b[i].e);
          else if(range[i].e <= set_a[i].e)
            range[i].bound(set_b[i].b, set_a[i].b - 1);
        }
      }
    }
  }
}

void macro(int displ)
{
  if(displ > 0)
  {
    TopologyTreeGroup<LEVELS, CHILDREN> tmp;

    tmp.clear();
    for(int i = 0; i < LEVELS; i++)
    {
      if(!range[i].empty())
      {
        int node1 = (range[i].b * CHILDREN) + displ - 1;
        int node2 = (range[i].e * CHILDREN) + displ - 1;

        tmp[i + 1].bound(node1, node2);
      }
    }
    *this = tmp;
  }

  if(displ < 0)
  {
    TopologyTreeGroup<LEVELS, CHILDREN> tmp;

    tmp.clear();
    for(int i = 0; i < LEVELS; i++)
    {
      if(!range[i].empty())
      {
        int pos1 = range[i].b;
        int pos2 = range[i].e;
        int parent1;
        int parent2;
```

```
        while(pos1 >= CHILDREN)
          pos1 = pos1 / 2;

        while(pos2 >= CHILDREN)
          pos2 = pos2 / 2;

        parent1 = (range[i].b - pos1) / CHILDREN;
        parent2 = (range[i].b - pos2) / CHILDREN;

        tmp[i - 1].bound(parant1, parent2);
      }
    }
    *this = tmp;
  }
}

void mapToOneDim(TopologyGroup<1>& tmp, string scatter)
{
  int index1;
  int index2;

  tmp.clear();

  if(scatter == "root")
  {
  }
  else if(scatter == "tree")
  {
    for(int i = 0; i < LEVELS; i++)
    {
      if(!range[i].empty())
      {
        index1 = mapToIndex(i, range[i].b);
        index2 = mapToIndex(i, range[i].e);

        if(tmp.empty())
          tmp.range[0].bound(index1, index2);
        else
        {
          if(index1 < tmp[0].b)
            tmp.range[0].b = index1;
          if(index2 > tmp[0].e)
            tmp.range[0].e = index2;
        }
      }
    }
  }
  else if(scatter == "leaves")
    tmp.range[0] = range[LEVELS - 1];
}

void buildElemsList(vector<TopologyTreeGroup<LEVELS, CHILDREN> >& vect,
                    int displ)
{
  vect.clear();

  for(int i = 0; i < LEVELS; i++)
  {
    if(!range[i].empty())
    {
      for(int j = range[i].b; j <= range[i].e;)
      {
```

```
        TopologyTreeGroup<LEVELS, CHILDREN> tmp;

        tmp.clear();
        tmp.range[i].bound(j, j);
        vect.push_back(tmp);
        j += displ;
      }
    }
  }
}

void buildElemsList(vector<TopologyTreeGroup<LEVELS, CHILDREN> >* set)
{
  set->clear();

  for(int i = 0; i < LEVELS; i++)
  {
    if(!range[i].empty())
    {
      for(int j = range[i].b; j <= range[i].e; j++)
      {
        TopologyTreeGroup<LEVELS, CHILDREN> tmp;

        tmp.clear();
        tmp[i].bound(j, j);
        set->push_back(tmp);
      }
    }
  }
}

int mapToIndex(int level, int node)
{
  int subtree = (int) (pow((double) children, (double) (dims - level))) - 1;
  int parent = node;
  int elems = subtree * node;

  if(level == 0)
    return(0);

  for(int l = level - 1; l >= 0; l--)
  {
    parent = ((parent % children) == 0) ? parent / children :
    (parent - 1) / children;
    elems += (parent + 1);
  }
  return(elems);
}

Range& operator[](int index)
{
  return(range[index]);
}

const Range& operator[](int index) const
{
  return(range[index]);
}

void operator=(const TopologyTreeGroup<LEVELS, CHILDREN>& copy)
{
  for(int i = 0; i < LEVELS; i++)
```

```
      range[i] = copy.range[i];
  }
};

template<int LEVELS, int CHILDREN>
  ostream & operator<<(ostream &stream, TopologyTreeGroup<LEVELS, CHILDREN> &set)
{
  for(int i = 0; i < LEVELS; i++)
    stream << set[i];
  return(stream);
}

template<int LEVELS, int CHILDREN>
  ostream & operator<<(ostream &stream, const TopologyTreeGroup<LEVELS, CHILDREN> &set)
{
  for(int i = 0; i < LEVELS; i++)
    stream << set[i];
  return(stream);
}
#endif
```

A.7 Classe VPMapperTree

Implementa le funzionalità per il mapping dei processori virtuali della topologia ad albero su quelli fisici; aggiunta alla libreria ASSIST.

```
#ifndef _VPMAPPER_TREE_HPP
#define _VPMAPPER_TREE_HPP

#include <cmath>
#include <vector>
#include <TopologyTreeGroup.hpp>

template <int LEVELS, int CHILDREN> class VPMapperTree
{
 public:

  enum
  {
    dims = LEVELS
  };

  VPMapperTree() {}

  VPMapperTree(TopologyTreeGroup<LEVELS, CHILDREN> vpset)
  {
    vps = vpset;
  }

  int getVPMCard()
  {
    return(map.size());
  }

  void setAllVp(TopologyTreeGroup<LEVELS, CHILDREN> vpset)
```

```
{
  vps = vpset;
}

TopologyTreeGroup<LEVELS, CHILDREN>& allVp()
{
  return(vps);
}

TopologyTreeGroup<LEVELS, CHILDREN> getVp(int vpm)
{
  return(map[vpm]);
}

TopologyGroup<1> mapVpLevel(int vpm, int index)
{
  Range rng = (map[vpm])[index];
  TopologyGroup<1> set = {{rng}};

  return(set);
}

void distribute(int procs, int level)
{
  int nodes = 1;

  for(int i = 0; i < procs; i++)
    map.push_back(vps);

  for(int i = 0; i < LEVELS; i++)
  {
    int base = 0;

    if(i <= level)
    {
      int index = 0;
      int displ = procs / nodes;

      for(int j = 0; j < procs; j++)
      {
        if(j == index)
        {
          (map[j])[i].bound(base, base);
          base++;
          index += displ;
        }
        else
          (map[j])[i].clear();
      }
    }
    else
    {
      int displ = nodes / procs;

      for(int j = 0; j < procs; j++)
      {
        map[j][i].bound(base, base + displ - 1);
        base += displ;
      }
    }
    nodes *= children;
  }
```

```
  }

 private:

  vector<TopologyTreeGroup<LEVELS, CHILDREN> > map;
  TopologyTreeGroup<LEVELS, CHILDREN> vps;
};
#endif
```

A.8 Classe `partTreeAttr`

Permette di rappresentare gli attributi partizionati sull'albero; aggiunta alla libreria ASSIST.

```
template <class T, int DIM, int LEVELS, int CHILDREN, int FOLDER> struct partTreeAttr:
  SMU<T, DIM, FOLDER>
{
  typedef enum
  {
    root,
    tree,
    leaves
  } distr_t ;

  bool pers[FOLDER];

  partTreeAttr(SMUSystem& sys): SMU<T, DIM, FOLDER>(sys, SMU_DOUBLE), fetched(true), sem(0) {}

  ~partTreeAttr() { }

  void setState(const TopologyGroup<DIM> &state)
  {
    local_state = state;
  }

  const TopologyGroup<DIM>& getState() const
  {
    return(local_state);
  }

  string get_type()
  {
    string type;

    if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::root)
      type = "root";
    else if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::tree)
      type = "tree";
    else if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::leaves)
      type = "leaves";

    return(type);
  }

  void switch_folder(int folder)
```

```
{
  fetched = !pers[folder];
  if((u_short) folder != current_folder())
    clear_memory();
  SMU<T,DIM,FOLDER>::switch_folder(folder);
}

void require(TopologyTreeGroup<LEVELS, CHILDREN>& set_a, TopologyGroup<DIM>& set_b,
             int displ, wish_t mode = SMU_VOLATILE)
{
  typedef vector<TopologyTreeGroup<LEVELS, CHILDREN> > vtg;
  vtg tmp;
  u_short folder = SMU<T, DIM, FOLDER>::current_folder();

  fetched = false;
  if(mode != SMU_VOLATILE)
    pers[current_folder()] = true;

  set_a.buildElemsList(tmp, displ);

  for(typename vtg::const_iterator it = tmp.begin(); it != tmp.end(); it++)
  {
    for(int i = 0; i < mapper.getVPMCard(); i++)
    {
      TopologyTreeGroup<LEVELS, CHILDREN> copy = *it;

      if((i != myVpm) && (copy.isWrapped(mapper.getVp(i))))
      {
        TopologyGroup<1> elem;
        TopologyGroup<DIM> req;

        if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::root)
          copy.mapToOneDim(elem, "root");
        else if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::tree)
          copy.mapToOneDim(elem, "tree");
        else if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::leaves)
          copy.mapToOneDim(elem, "leaves");
        partial_assign(req, elem, set_b);
        SMU<T, DIM, FOLDER>::require(folder, req, i, mode);
      }
    }
  }
}

void init(VPMapperTree<LEVELS, CHILDREN>& vpmapper, distr_t type,
          char mode = SMU_DOUBLE)
{
  vps.clear();
  mapper = vpmapper;
  scatter_type = type;

  if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::root)
  {
    if(myVpm == 0)
      locate(myVpm, local_state);
    else
    {
      locate(0, local_state);
      local_state.clear();
    }
    vps = mapper.getVp(myVpm);
  }
```

```
  else if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::tree)
  {
    for(int i = 0; i < mapper.getVPMCard(); i++)
    {
      TopologyTreeGroup<LEVELS, CHILDREN> copy = mapper.getVp(i);
      TopologyGroup<1> tree;
      TopologyGroup<DIM> tmp;

      copy.mapToOneDim(tree, "tree");
      partial_assign(tmp, tree, local_state);
      locate(i, tmp);

      if(i == myVpm)
      {
        local_state = tmp;
        vps = copy;
      }
    }
  }
  else if(scatter_type == partTreeAttr<T, DIM, LEVELS, CHILDREN, FOLDER>::leaves)
  {
    for(int i = 0; i < mapper.getVPMCard(); i++)
    {
      TopologyTreeGroup<LEVELS, CHILDREN> copy = mapper.getVp(i);
      TopologyGroup<1> leaves;
      TopologyGroup<DIM> tmp;

      leaves.range[0] = copy[LEVELS - 1];
      partial_assign(tmp, leaves, local_state);
      locate(i, tmp);

      if(i == myVpm)
      {
        local_state = tmp;
        vps = copy;
      }
    }
  }
  state_card = local_state.cardinality();
}

void set_buffer(T *buffer)
{
  SMU<T, DIM, FOLDER>::set_buffer(buffer);
}

void set_buffer(dynbuf<T>& buffer)
{
  SMU<T, DIM, FOLDER>::set_buffer(buffer.begin());
}

template<typename U> void set_buffer(U buffer[])
{
  set_buffer(*buffer);
}

template<typename U> void set_buffer(dynbuf<U> &buffer)
{
  set_buffer(buffer.begin());
}

bool get_out(StateReference<T, DIM> &state)
```

```
  {
    CHOOSE_SEMANTIC(dirty = true, out_regions.add(state.group));
    return(SMU<T, DIM, FOLDER>::get_out(state));
  }

  err_t commit_changes()
  {
#ifndef SWAP_BUF_OPT
    if(dirty)
    {
      dirty = false;
      return(SMU<T, DIM, FOLDER>::commit_changes());
    }
#else
    if(out_regions.size() > 0)
    {
      for(int i = 0; i < out_regions.size(); i++)
      {
        StateReference<T, DIM> sr(out_regions[i]);
        get_out(sr);
        put(sr);
      }
      swap_buffers();
      out_regions.clear();
    }
#endif
    return AST_SUCCESS;
  }

  void fetch()
  {
    if(!fetched)
    {
      SMU<T,DIM,FOLDER>::fetch();
      fetched = true;
    }
  }

 private:
  bool fetched;
  int state_card;
  distr_t scatter_type;
  ACE_Thread_Semaphore sem;
  TopologyGroup<DIM> local_state;
  VPMapperTree<LEVELS, CHILDREN> mapper;
  TopologyTreeGroup<LEVELS, CHILDREN> vps;
  CHOOSE_SEMANTIC(bool dirty, TopologyCompactGroup<DIM> out_regions);
};
```

A.9 Classe `TreeAttr`

Classe utilizzata per rappresentare un parametro di input o di output di una elaborazione; aggiunta alla libreria ASSIST.

```
template <typename Attr, int dimVP, int children, direction_t dir, typename ExprT,
  bool use_threads = true> struct TreeAttr: Reference<Attr>
{
  enum
  {
    dimS = Attr::dimensions
  };

  typedef TopologyGroup<dimS> tgd;
  typedef TopologyGroup<1> tg1;
  typedef TopologyTreeGroup<dimVP, children> tg;
  typedef vector<tg > vtg;
  typedef typename buffer_t<typename Attr::base_t, ExprT>::type user_t;
  typedef Walker<typename Attr::base_t, dimS> Walk_t;
  typedef Walk_t local_t;
  ExprT exp;
  dynbuf<typename Walk_t::sr_t> srs;
  tg local_vps;

  void open(local_t& walker, int index)
  {
    walker.open(srs.getElem(index));
  }

  user_t & get_fast(local_t& walker, const Index<dimVP>& index)
  {
    return(walker.template get_data<user_t>());
  }

  user_t & get(local_t& walker, const Index<dimVP>& index)
  {
    return(walker.template get_data<user_t>());
  }

  void next(local_t& walker, bool nil)
  {
    if(nil)
      walker.inner_inc();
    else
      walker.invalidate();
  }

  void done()
  {
    if(dir == P_OUT)
      get_ref().commit_changes();
  }

  void identify(vtg& local, vtg& remote, const tg& vps)
  {
    local_vps = vps;

    if(dir == P_IN)
    {
      if(ExprT::isstatic)
      {
        typename vtg::iterator it = local.begin();

        if(it != local.end())
        {
          tg tmp = *it;
          tg diff;
```

```
        exp.unapply(tmp);
        diff.buildDiffSet(local_vps, tmp);
        if(!diff.empty())
        {
          tg1 elem;
          tgd request;

          elem.clear();
          request.clear();
          exp.apply(elem, request);
          get_ref().require(diff, request, exp.get_displ(), SMU_PERSISTENT);
        }
      }
    }
  }
}

void discern(vtg &local, vtg &remote, vtg &remain)
{
  if(dir == P_IN)
  {
    if(!ExprT::isstatic)
    {
      typename vtg::iterator it = local.begin();

      if(it != local.end())
      {
        tg tmp = *it;
        tg diff;

        exp.unapply(tmp);
        diff.buildDiffSet(local_vps, tmp);
        if(!diff.empty())
        {
          tg1 elem;
          tgd request;

          elem.clear();
          request.clear();
          exp.apply(elem, request);
          get_ref().require(diff, request, exp.get_displ(), SMU_VOLATILE);
        }
      }
    }
  }
}

void prepare(const vtg *vect)
{
  typename vtg::const_iterator it = vect->begin();

  if(it != vect->end())
  {
    vtg tmp;
    int i = 0;
    tg copy = *it;

    if(exp.is_macro())
      copy.macro(exp.get_macro());
    copy.buildElemsList(tmp, exp.get_displ());
    srs.reserve(tmp.size());
```

```
      for(it = tmp.begin(); it != tmp.end(); it++, i++)
      {
        tg1 elem;
        typename Walk_t::sr_t& sr = srs.getElem(i);

        copy = *it;
        copy.mapToOneDim(elem, get_ref().get_type());
        exp.apply(elem, sr.group);

        if(dir != P_IN)
          get_ref().get_out(sr);
        else
          get_ref().get(sr);
      }
    }
  }

  void fetch()
  {
    if(dir == P_IN)
      get_ref().fetch();
  }
};
```

A.10 Stencil

Rappresentano i nuovi stecil da utilizzare per l'accesso agli elementi dello stato distribuito; aggiunti nella libreria ASSIST.

```
template <N LEVELS, N CHILDREN> struct l
{
  enum
  {
    dims = 1,
    vpsdim = LEVELS,
    chld = CHILDREN,
    isstatic = 1,
    isinterval = 0,
    isintervalc = 0,
    isunitaryinc = 0,
    isindex = 0
  };

  static bool unapply(TopologyTreeGroup<vpsdim, chld>& s)
  {
    return(true);
  }

  static interval apply(const II<dims>& s)
  {
    return(s[0]);
  }

  static int eval(const MI<vpsdim>& index)
```

```
  {
    for(int i = 0; i < vpsdim; i++)
      if(index[i] != -1)
        return(i);
    return(0);
  }

  static int get_displ()
  {
    return(1);
  }

  static bool is_macro()
  {
    return(false);
  }

  static int get_macro()
  {
    return(0);
  }
};

template <N LEVELS, N CHILDREN> struct n
{
  enum
  {
    dims = 1,
    vpsdim = LEVELS,
    chld = CHILDREN,
    isstatic = 1,
    isinterval = 0,
    isintervalc = 0,
    isunitaryinc = 0,
    isindex = 0
  };

  static bool unapply(TopologyTreeGroup<vpsdim, chld>& s)
  {
    return(true);
  }

  static interval apply(const II<dims>& s)
  {
    return(s.range[0]);
  }

  static int eval(const Index<vpsdim>& index)
  {
    for(int i = 0; i < vpsdim; i++)
      if(index[i] != -1)
        return(index[i]);
    return(0);
  }

  static int get_displ()
  {
    return(1);
  }

  static bool is_macro()
  {
```

```
    return(false);
  }

  static int get_macro()
  {
    return(0);
  }
};

template <N_LEVELS, N_CHILDREN, N_MACRO_IDX> struct m
{
  enum
  {
    dims = 1,
    vpsdim = LEVELS,
    chld = CHILDREN,
    macro = MACRO_IDX,
    isstatic = 1,
    isinterval = 0,
    isintervalc = 0,
    isunitaryinc = 0,
    isindex = 0
  };

  static bool unapply(TopologyTreeGroup<vpsdim, chld>& s)
  {
    s.macro(macro);
    return(true);
  }

  static interval apply(const II<dims>& s)
  {
    return(s.range[0]);
  }

  static int get_displ()
  {
    return((macro == 0)? 1 : chld);
  }

  static bool is_macro()
  {
    return(true);
  }

  static int get_macro()
  {
    return(macro);
  }
};
```

A.11 Lista dei tasks

Implementano il supporto per la gestione delle liste di tasks associati ai processori virtuali; aggiunte alla libreria ASSIST.

```
template<typename ProcTree> class ProcTreeList
{
 public:
  typedef TopologyTreeGroup<ProcTree::vpsdim, ProcTree::chld> tg;
  typedef vector<tg> vtg;
  typedef vector<vtg> vvtg;

  ProcTreeList& operator,(ProcTree *proc)
  {
    procs.push_back(proc);
    return(*this);
  }

  ProcTreeList& operator=(ProcTree *proc)
  {
    procs.push_back(proc);
    return(*this);
  }

  void identify(const vvtg &elems, const tg &vps)
  {
    loc_set = elems;
    rem_set.resize(elems.size());
    typename vvtg::iterator loc_it = loc_set.begin();
    typename vvtg::iterator rem_it = rem_set.begin();

    for(typename vector<ProcTree *>::iterator proc = procs.begin(); proc != procs.end();
        proc++, loc_it++, rem_it++)
      if(*proc)
        (*proc)->identify(*loc_it, *rem_it, vps);
  }

  void discern(vvtg &loc_set, vvtg &rem_set, vvtg &remain)
  {
    typename vvtg::iterator loc_it = loc_set.begin();
    typename vvtg::iterator rem_it = rem_set.begin();
    typename vvtg::iterator it = remain.begin();

    for(typename vector<ProcTree *>::iterator proc = procs.begin(); proc != procs.end();
        proc++, loc_it++, rem_it++, it++)
      if(*proc)
        (*proc)->discern(*loc_it, *rem_it, *it);
  }

  void prepare(vvtg &vect)
  {
    typename vvtg::iterator it = vect.begin();

    for(typename vector<ProcTree *>::iterator proc = procs.begin(); proc != procs.end();
        proc++ , it++)
      if(*proc)
        (*proc)->prepare(&*it);
  }

  template <class PE_t> void start(PE_t& pe)
  {
    for(typename vector<ProcTree *>::iterator proc = procs.begin(); proc != procs.end();
        proc++)
      if(*proc)
        (*proc)->start(pe);
  }
```

```
  void done()
  {
    for(typename vector<ProcTree *>::iterator proc = procs.begin(); proc != procs.end();
        proc++)
      if(*proc)
        (*proc)->done();
  }

  void fetch()
  {
    for(typename vector<ProcTree *>::iterator proc = procs.begin(); proc != procs.end();
        proc++)
      if(*proc)
        (*proc)->fetch();
  }

  template <class PE_t> void cycle(PE_t &pe, OutputController *oc, bool not_all_at_once = true)
  {
    vvtg loc(loc_set);
    vvtg rem(rem_set);
    vvtg rest;

    discern(loc, rem, rest);
    fetch();
    prepare(loc);
    start(pe);
    if(oc)
      oc->run();
  }

  template <class PE_t> void cycle(PE_t &pe, OutputController *oc, int i)
  {
    vvtg loc(loc_set);
    vvtg rem(rem_set);
    vvtg rest;

    discern(loc, rem, rest);
    fetch();
    prepare(loc);
    start(pe);
    if(oc)
      oc->run();
  }

 private:
  vector<ProcTree *> procs;
  vvtg loc_set;
  vvtg rem_set;
};

template <int LEVELS, int CHILDREN> class ProcTree: AstTask
{
 public:

  enum
  {
    dims = 1,
    vpsdim = LEVELS,
    chld = CHILDREN
  };

  typedef TopologyTreeGroup<LEVELS, CHILDREN> tg;
```

```
 typedef vector<tg> vtg;

 ProcTree() {}

 template <typename Elab_t> ProcTree(Elab_t *elab, bool all_thr, unsigned div = 1)
 {
   oc = elab->OutController;
   if(all_thr)
     complete();
   else
     adaptive(div);
 }

 virtual void identify(vtg &loc_set, vtg &rem_set, const tg &vps) {}

 virtual void discern(vtg &loc_set, vtg &rem_set, vtg &remain) {}

 virtual void fetch() {}

 virtual void done() {}

 virtual void prepare(vtg *vect)
 {
   typename vtg::const_iterator it = vect->begin();

   tgs = new(vtg);

   if(it != vect->end())
   {
     tg tmp = *it;

     bound(0, tmp.cardinality() - 1);
     tmp.buildElemsList(tgs);
   }
   else
     bound(0, -1);
 }

 virtual void start()
 {
   if(oc)
     oc->new_task(tgs->size());
 }

 void execute()
 {
   execute(0, tgs->size() - 1);
 }

 template<class PE_t> void start(PE_t &pe)
 {
   start();
   pe.wish_execute(this);
 }

protected:

 vtg *tgs;
 OutputController * oc;

 virtual void execute(int min, int max)
 {
```

```
    if(oc)
      oc->dec_tasks(max - min + 1);
  }
};
```

A.12 Politiche di distribuzione

Implementano le politiche di distribuzione associate ad i nuovi attributi partizionati; aggiunte nella libreria ASSIST.

```
template <int LEVELS, int CHILDREN> class ScatterRoot_Poly
{  VPMapperTree<dims, chld> mapper;

 public:

  enum
  {
    dims = LEVELS,
    chld = CHILDREN
  };

  void setup(const TopologyTreeGroup<dims, chld> vps, int vpms, int level)
  {
    mapper.setAllVp(vps);
    mapper.distribute(vpms, level);
  }

  template<typename sent_t, typename coll> int put(const sent_t& elems, coll& scatter)
  {
    //Da implementare.
  }
};

template <int LEVELS, int CHILDREN> class ScatterLeaves_Poly
{
  VPMapperTree<dims, chld> mapper;

 public:

  enum
  {
    dims = LEVELS,
    chld = CHILDREN
  };

  void setup(const TopologyTreeGroup<dims, chld> vps, int vpms, int level)
  {
    mapper.setAllVp(vps);
    mapper.distribute(vpms, level);
  }

  template<typename sent_t, typename coll> int put(const sent_t& elems, coll& scatter)
  {
    typedef typename coll::tosend_t tosend_t;
    tosend_t buffer;
```

```
    int vpms = mapper.getVPMCard();

    buffer.reserve(vpms);

    for(int i = 0; i < vpms; i++)
    {
      Index<1> index;
      typedef deref<1, sent_t> vpm_t;
      typename tosend_t::value_type& data = buffer[i];
      TopologyGroup<1> vp = mapper.mapVpLevel(i, dims - 1);

      vp.firstIndex(index);

#ifndef UNSAFE_CAST
      data.share(const_cast<typename vpm_t::return_t *>(&vpm_t::apply_c(elems, index)),
                 vp.cardinality());
#else
      typedef typename tosend_t::value_type::value_type db_base_t;
      data.share((db_base_t*) const_cast<typename vpm_t::return_t *>(&vpm_t::apply_c(elems, index)),
                 vp.cardinality() * (sizeof(typename vpm_t::return_t) / sizeof (db_base_t)));
#endif
    }
    return(scatter.put(buffer));
  }
};
```

A.13 Politiche di collezione

Implementano le politiche di collezione legate alle strategie **from ROOT** e **from LEAVES**; aggiunte alla libreria ASSIST.

```
template <int LEVELS, int CHILDREN> struct CollRoot_Poly
{
  enum
  {
    dims = LEVELS,
    chld = CHILDREN,
    needbuf = 0
  };

  void setup(const TopologyTreeGroup<dims, chld>& vps, int vpms, int level)
  {
  }

  template <typename buffer_t, typename oper_t>
    void xform(const buffer_t& elems, oper_t op)
    {
      op.set_buffer(elems);
    }
};

template <int LEVELS, int CHILDREN> struct CollLeaves_Poly
{
  //Da implementare
};
```

APPENDICE B

Sorgenti dei programmi

B.1 Massimo

B.1.1 Topologia ad albero

```
#define LEVELS 10
#define CHILDREN 2
#define DIM_VECT 1024
#define ELEMS 2047
#define NUM_ITER 150000

proc create_vect(output_stream long vect[DIM_VECT])
$c++{
  int tmp[DIM_VECT];
  for(int i = 0; i < DIM_VECT ; i++)
    tmp[i] = i;
  assist_out(vect, tmp);
}c++$

send_data(output_stream long data[DIM_VECT])
{
  create_vect(output_stream data);
}

proc print_val(in long val)
inc<"iostream">
$c++{
  cerr << "Massimo:␣" << val << endl;
}c++$

print_result(input_stream long res)
{
  print_val(in res);
```

```
}

proc select(in long value1, long value2, long iter, long level out long max)
inc<"cmath">
$c++{
  if((LEVELS - iter) == level)
  {
    for(double i = 0; i < NUM_ITER; i++)
      double val = cos(i) * sin(i);

    max = value1;
    if(value1 < value2)
      max = value2;
  }
}c++$

proc copy_elem(in long from_value, long iter, long level out long to_value)
inc<"cmath">
$c++{
  if((LEVELS - iter) == level)
  {
    for(double i = 0; i < NUM_ITER; i++)
      double val = cos(i) * sin(i);

    to_value = from_value;
  }
}c++$

parmod max(input_stream long vect[DIM_VECT] output_stream long max)
{
  topology tree[l:LEVELS][c:CHILDREN] vps;
  attribute long values[ELEMS] scatter_tree values[*i] onto vps;
  attribute long recv[DIM_VECT] scatter_tree recv[*j] onto vps[LEVELS][j];

  input_section
  {
    guard: on , , vect
    {
      distribution vect[*h] scatter to recv[h];
    }
  }

  virtual_processors
  {
    select_elem(in guard out max)
    {
      VP l = LEVELS, c // Nodi foglia.
      {
        for(index0 = 0; index0 <= (LEVELS + 1); index0++)
        {
          copy_elem(in recv[c], index0, l out values[NODE_INDEX]);
        };
      }

      VP l = 1..(LEVELS - 1), c // Nodi intermedi.
      {
        for(index1 = 0; index1 <= (LEVELS + 1); index1++)
        {
          select(in values[CHILD_INDEX(1)], values[CHILD_INDEX(2)], index1, l
                 out values[NODE_INDEX]);
        };
      }
```

```
      VP l = 0, c = 0 // Radice.
      {
        for(index2 = 0; index2 <= (LEVELS + 1); index2++)
        {
          select(in values[CHILD_INDEX(1)], values[CHILD_INDEX(2)], index2, l
                 out values[NODE_INDEX]);
        };
        assist_out(max, values[NODE_INDEX]);
      }
    }
  }

  output_section
  {
    collects max from ROOT vps;
  }
}

generic main()
{
  stream long [DIM_VECT] send;
  stream long res;

  send_data(output_stream send);
  max(input_stream send output_stream res);
  print_result(input_stream res);
}
```

B.1.2 Topologia array

```
#define LEVELS 10
#define DIM_VECT 1024
#define NUM_ITER 500000

proc create_vect(output_stream long vect[DIM_VECT])
$c++{
  int tmp[DIM_VECT];
  for(int i = 0; i < DIM_VECT ; i++)
    tmp[i] = i;
  assist_out(vect, tmp);
}c++$

send_data(output_stream long data[DIM_VECT])
{
  create_vect(output_stream data);
}

proc print_val(in long val)
inc<"iostream">
$c++{
  cerr << "Massimo:␣" << val << endl;
}c++$

print_result(input_stream long res)
{
  print_val(in res);
}
```

```
proc set_index(out long value)
$c++{
  value = 2;
}c++$

proc select(in long myvp, long elems[DIM_VECT], long index, long iter
            out long value, long new_index)
inc<"cmath", "iostream">
$c++{
  int copy = index;
  int nodes = (int) pow((double) 2, (double) iter);

  for(int i = 0; i < nodes; i++)
  {
    if((i * copy) == myvp)
    {
      for(double i = 0; i < NUM_ITER; i++)
        double val = cos(i) * sin(i);

      if(elems[myvp + (copy / 2)] > elems[myvp])
        value = elems[myvp + (copy / 2)];
    }
  }
  new_index = new_index * 2;
}c++$

proc send_max(in long myvp, long value output_stream long result)
$c++{
  if(myvp == 0)
    assist_out(result, value);
}c++$

parmod max(input_stream long vect[DIM_VECT] output_stream long max)
{
  topology array[i:DIM_VECT] vps;
  attribute long idx replicated;
  attribute long recv[DIM_VECT] scatter recv[*j] onto vps[j];

  init
  {
    VP i
    {
      set_index(out idx);
    }
  }

  input_section
  {
    guard: on , , vect
    {
      distribution vect[*h] scatter to recv[h];
    }
  }

  virtual_processors
  {
    select_elem(in guard out max)
    {
      VP i
      {
        for(index0 = (LEVELS - 1); index0 >= -1; index0--)
        {
```

```
          select(in i, recv[], idx, index0 out recv[i], idx);
        };
        send_max(in i, recv[i] output_stream max);
      }
    }
  }

  output_section
  {
    collects max from ANY vps;
  }
}

generic main()
{
  stream long [DIM_VECT] send;
  stream long res;

  send_data(output_stream send);
  max(input_stream send output_stream res);
  print_result(input_stream res);
}
```

B.2 Data parallel

B.2.1 Topologia ad albero

```
#define LEVELS 10
#define CHILDREN 2
#define DIM_VECT 2047
#define NUM_ITER 500000

proc create_vect(output_stream long vect[DIM_VECT])
$c++{
  int tmp[DIM_VECT];
  for(int i = 0; i < DIM_VECT ; i++)
    tmp[i] = i;
  assist_out(vect, tmp);
}c++$

send_tree(output_stream long data[DIM_VECT])
{
  create_vect(output_stream data);
}

proc nop(in long data)
$c++{
}c++$

get_tree(input_stream long elem)
{
  nop(in elem);
}

proc prod(in long value out long prod_value)
$c++{
  int tmp;
```

```
  for(double i = 0; i < NUM_ITER; i++)
    double val = cos(i) * sin(i);

  tmp = value * value;
}c++$

parmod top_tree(input_stream long vect[DIM_VECT] output_stream long send_elem)
{
  topology tree[l:LEVELS][c:CHILDREN] vps;
  attribute long recv[DIM_VECT] scatter_tree vect[*i] onto vps;

  input_section
  {
    guard: on , , vect
    {
      distribution vect[*h] scatter to recv[h];
    }
  }

  virtual_processors
  {
    prod_tree(in guard out send_elem)
    {
      VP l = 1..(LEVELS), c
      {
        prod(in recv[NODE_INDEX] out recv[NODE_INDEX]);
      }

      VP l = 0, c = 0
      {
        prod(in recv[NODE_INDEX] out recv[NODE_INDEX]);
        assist_out(send_elem, recv[NODE_INDEX]);
      }
    }
  }

  output_section
  {
    collects send_elem from ROOT vps;
  }
}

generic main()
{
  stream long [DIM_VECT] send;
  stream long result;

  send(output_stream send);
  top_tree(input_stream send output_stream result);
  get_tree(input_stream result);
}
```

B.2.2 Topologia array

```
#define DIM_VECT 2047
#define NUM_ITER 500000

proc create_vect(output_stream long vect[DIM_VECT])
```

```
$c++{
  int tmp[DIM_VECT];
  for(int i = 0; i < DIM_VECT ; i++)
    tmp[i] = i;
  assist_out(vect, tmp);
}c++$

send_array(output_stream long data[DIM_VECT])
{
  create_vect(output_stream data);
}

proc nop(in long data)
$c++{
}c++$

get_array(input_stream long elem)
{
  nop(in elem);
}

proc prod(in long value, long index output_stream long prod_value)
$c++{
  int tmp;

  for(double i = 0; i < NUM_ITER; i++)
    double val = cos(i) * sin(i);

  tmp = value * value;
  if(index == 0)
    assist_out(prod_value, tmp);
}c++$

parmod top_array(input_stream long vect[DIM_VECT] output_stream long send_elem)
{
  topology array[i:DIM_VECT] vps;
  attribute long recv[DIM_VECT] scatter vect[*j] onto vps[j];

  input_section
  {
    guard: on , , vect
    {
      distribution vect[*h] scatter to recv[h];
    }
  }

  virtual_processors
  {
    sqrt_array(in guard out send_elem)
    {
      VP i
      {
        prod(in recv[i], i output_stream send_elem);
      }
    }
  }

  output_section
  {
    collects send_elem from ANY vps;
  }
}
```

```
generic main()
{
  stream long [DIM_VECT] send;
  stream long result;

  send_array(output_stream send);
  top_array(input_stream send output_stream result);
  get_array(input_stream result);
}
```