Università degli studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

Supporto di strumenti di programmazione parallela strutturata per sistemi basati su CELL

Candidati:

Marco Fais
Michele Onnis

Relatori

Prof. Marco Vanneschi
Prof. Massimo Coppola

Controrelatore

Prof. Vincenzo Gervasi

Anno Accademico 2007/2008

On two occasions I have been asked, - "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" In one case a member of the Upper, and in the other a member of the Lower House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

- Charles Babbage

Ringraziamenti Michele

Mi sento fortunato, ho raggiunto un bell'obiettivo e ho avuto la possibilità di farlo. Sono stati anni che hanno comportato sacrifici non da poco per la mia famiglia, un padre, una madre e una sorella d'oro, che mi hanno sempre sostenuto e mi hanno dato l'opportunità di formarmi su ciò che mi affascina e che desidero sia il mio lavoro. Ma sono ancora più fortunato, perchè di famiglie ne ho due. Perchè ho dei suoceri, un cognato e una nonna "adottiva", che mi hanno accolto come se fossi un figlio, un fratello e un nipote. E la mia felicità è guardare negli occhi tutti voi, e sentire che siete orgogliosi di me. Vi voglio bene.

Sono stato fortunato anche quando ho deciso di fare la tesi con i Prof. Marco Vanneschi e Massimo Coppola, perchè nel loro impegno infondono una passione che ti contagia e ti coinvolge. Ho potuto lavorare con Massimiliano Meneghin, gentilissimo, sempre pronto a dare una mano, e con una capacità di risolvere problemi che mi stupisce sempre. Carlo Bertolli non si è mai risparmiato nel dare supporto umano e tecnico. E il laboratorio di Architetture Parallele è proprio un bel posto in cui lavorare, dove c'è sempre la possibilità di farsi due risate tra un caffè, una cecina e un problemino da risolvere. Grazie ad Alessio, Paolo, Daniele, Gabriele, e a tutti gli inquilini del laboratorio; con voi sono stato proprio bene.

Con Marco abbiamo condiviso tesi, studi e soprattutto casa, fianco a fianco giorno dopo giorno. Sono stato fortunato per l'ennesima volta, perchè in lui ho sempre trovato un amico e una persona buona e leale. Mi ha sopportato per tutto questo tempo, è davvero un grande. E come lui gli amici sardi

con cui abbiamo condiviso la trasferta pisana; Carlo, Simona, Federico, Maddalena, Barbara... con voi ci si sente a casa. E grazie agli amici che mi hanno sempre supportato quando si doveva prendere nuovamente l'aereo e andare via: Massimiliano, Federica e Alessandrino, Simone e Roberta, Massimo e Romina.

Ma la mia fortuna sfacciata è stata conoscere una ragazza come te, Alessandra. Sono stato lontano da te per tanto tempo, ma tu mi hai sempre sostenuto, e mi sei stata sempre vicina. Questo traguardo lo dedico a te che sei la mia ragazza, tra pochi mesi sarai mia moglie, e quando lo dico si realizza un sogno. Grazie amore!

Michele

Ringraziamenti Marco

Questo lavoro rappresenta la conclusione di un percorso che mi ha permesso di crescere tanto sia dal punto di vista professionale che da quello umano. Con queste poche righe voglio ringraziare tutte le persone che in questi anni hanno fatto parte della mia vita e, in un modo o nell'altro, mi hanno reso una persona migliore.

Voglio iniziare col ringraziare i miei genitori e mia sorella Alessandra, per essermi stati vicini e avermi sempre sostenuto appoggiando tutte le scelte che ho fatto. Vi voglio bene.

Un grazie ai Prof. Marco Vanneschi e Massimo Coppola, per i preziosi consigli che hanno saputo darmi: due ottimi docenti, due persone di straordinaria cultura che sono orgoglioso di aver avuto come relatori, sicuramente tra le persone che più di tutte hanno influito sulla mia crescita professionale.

Grazie a tutto il laboratorio di Architetture Parallele, nel quale ho avuto la fortuna di conoscere delle persone davvero fantastiche che mi hanno fatto sentire a mio agio fin dal primo giorno, a cominciare da Alessio Pascucci e Carlo Bertolli, sempre pronti a discutere sull'eventuale problema di turno mettendo a disposizione le loro conoscenze e la loro esperienza, continuando poi con Gabriele Mencagli, Daniele Buono, Paolo Lategano, Marina Bartolozzi, Silvia Lametti, Sabrina Aversano, Giacomo Fariello, Andrea Ballini e tutti gli altri ragazzi che hanno frequentato il laboratorio in questi mesi. Un ringraziamento particolare va a Massimiliano Meneghin, che ci ha seguito durante lo sviluppo della libreria e ha rappresentato un punto di riferimento fondamentale, dimostrando di essere un amico su cui poter sempre contare.

E poi non dimenticherò mai i momenti di svago in laboratorio, dalle pause caffè ai divanetti, al mitico *Sacher Cake Contest*.

Voglio ringraziare anche gli amici che mi sono stati accanto in questi anni e in compagnia dei quali ho passato delle bellissime serate: Carlo e Simona, Federico, Barbara, Maddalena, Laura, Silvia.

Un ultimo pensiero va a Michele col quale, dopo tirocinio e tesi della laurea triennale ho avuto la fortuna di condividere anche il percorso della laurea specialistica, tesi compresa. Ci conosciamo ormai da otto anni, da quattro siamo coinquilini, è una grande persona e un preparatissimo collega, e sono contento di poterlo definire uno dei miei migliori amici.

Questa tesi è dedicata a tutti voi. Grazie ancora.

Marco

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi anni lo sviluppo tecnologico delle macchine *uniprocessor* ha raggiunto un punto di stallo. Il tentativo di incrementare la velocità dei processori tramite metodi tradizionali come l'aumento della frequenza di clock o l'uso di pipeline sempre più profonde sta per raggiungere i limiti fisici. Per questo motivo le maggiori case costruttrici di processori (AMD, IBM, Intel, Sun) stanno cambiando strategia, rivolgendo la loro attenzione su un altro modello di calcolo, quello delle macchine *multiprocessor* su singolo chip, che hanno ormai raggiunto il mercato consumer, sia nei personal computer che nelle macchine da gioco. Le CPU *multicore* incapsulano in un unico chip più processori, connessi fra loro da una rete di interconnessione. Data la sempre crescente domanda di potenza di calcolo si prevede, in un futuro prossimo, un aumento del numero dei core, che con buone probabilità seguirà un andamento simile a quello dettato dalla legge di Moore per la frequenza delle macchine *uniprocessor*. I processori *multicore* vengono tuttora programmati come macchine sequenziali, lanciando più thread indipendenti sui vari core e sfruttando solamente parte delle potenzialità offerte. Questo approccio può risultare soddisfacente quando il numero di core è limitato (le macchine *multicore* tuttora in commercio hanno un numero di core che varia dai due alla decina). Essendo questo numero destinato ad aumentare, è necessario studiare delle tecniche standard per lavorare con un numero di processori ben superiore alla decina. La programmazione parallela è tuttavia ancora giovane e, a differenza della programmazione sequenziale, non esistono strumenti standard che permettano uno sviluppo semplice di applicazioni parallele por-

tabili. Gli strumenti di sviluppo messi a disposizione dalle case costruttrici operano spesso a un livello troppo dipendente dalla macchina, non astraendo abbastanza dall'architettura sottostante oppure, pur operando ad alto livello, non permettono di esplicitare il parallelismo. In queste condizioni, scrivere un'applicazione parallela diventa un lavoro complesso e a basso livello, dove il coordinamento tra processi e il codice sequenziale del programma sono mescolati in maniera non strutturata.

La programmazione parallela strutturata fornisce un modello che consente di esprimere un'applicazione parallela come composizione di strutture di computazione elementari chiamate *skeleton*, separando gli aspetti relativi al coordinamento da quelli legati al codice sequenziale. Per ogni *skeleton* è possibile ricavare un modello dei costi, dipendente dall'architettura e dalla particolare applicazione, che consente di stimare le prestazioni dei moduli coinvolti. La natura modulare del modello, associata alla predicibilità delle performance degli *skeleton*, offre uno strumento formale utile per la parallelizzazione di una computazione.

L'obiettivo che ci si prefigge con questo lavoro di tesi è quello di implementare il supporto ad uno strumento di programmazione parallela strutturata su un'architettura *multiprocessor* come il Cell.

La prima generazione di *Cell Broadband Engine* è anche la prima implementazione reale di una nuova famiglia di *multiprocessor* conformi alla *Cell Broadband Engine Architecture* (*CBEA*), una nuova architettura che estende quella a 64 bit del PowerPC. La CBEA e il *Cell Broadband Engine* sono il risultato della collaborazione, iniziata alla fine del 2001, tra Sony, Toshiba e IBM. E' un'architettura NUMA costituita da nove core, funzionanti ad una frequenza di 3.2 GHz, di cui uno è un processore pipeline tradizionale, conforme all'architettura PowerPC. I restanti otto sono core più semplici, privi di cache, dotati di un set di istruzioni vettoriali, simile a quello del coprocessore VMX, spesso affiancato ai processori PowerPC, e di una memoria locale di 256 KB. I core possono comunicare tramite una struttura di interconnessione composta da quattro anelli, capace di una banda complessiva di 204.8 GB/s,

e di una banda di 25.6 GB/s per un trasferimento punto a punto fra due memorie locali. Nelle versioni tuttora in commercio, il processore è affiancato da una memoria Rambus XDR, capace di una banda di 25.6 GB/s. Per una descrizione più dettagliata dell'architettura si rimanda al capitolo 1. In particolare, sviluppo e sperimentazione sono stati svolti sulla Playstation 3 commercializzata da Sony.

L'approccio seguito nello sviluppo di applicazioni parallele durante questo lavoro è basato su due livelli distinti. Il primo è costituito dal linguaggio concorrente di sistema (LC) presentato in [8], che a sua volta riprende il modello CSP, presentato in [1]. Il secondo è costituito dal linguaggio ASSIST [1], definito in [3], che fornisce un'ulteriore livello di astrazione a partire dal linguaggio LC.

La libreria MammuT [2] è una libreria di classi per il linguaggio C++, sviluppata presso il Laboratorio di Architetture Parallele del Dipartimento di Informatica da Massimiliano Meneghin durante il periodo del suo dottorato di ricerca. La libreria implementa i costrutti principali del linguaggio LC su Cell. Vengono offerte diverse versioni dei canali di comunicazione e un'implementazione del comando alternativo, utilizzabile con guardie in ingresso e in uscita. Durante questo lavoro di tesi la libreria è stata estesa per includere ulteriori costrutti di LC come array di processi e array di canali, fondamentali per la programmazione parametrica di applicazioni parallele.

ASSIST nasce dall'idea di estendere i modelli classici di programmazione parallela strutturata. Questo per almeno due ragioni:

1. oltre alla possibiltà di esprimere alcuni tipici schemi paralleli, è necessario un più ampio grado di flessibilità per descrivere la struttura di un programma parallelo; in generale dovrebbe essere possibile esprimere

[1] *A Software development System based upon Integrated Skeleton Technology*
[2] *cell Multicore Architecture coMMUnication supporT*

delle strutture a grafo generiche, i cui nodi sono parallelizzati in modo strutturato;

2. i modelli di programmazione a *skeleton* hanno una semantica funzionale e deterministica, che può essere una limitazione pesante in molti tipi di applicazioni. I concetti di stato interno di un modulo e di nondeterminismo nelle comunicazioni tra moduli sono spesso fondamentali, ma non sono previsti nei modelli a *skeleton*. Inoltre, la programmazione di algoritmi data parallel con pattern di comunicazione complessi è difficile con i modelli a *skeleton*.

Lo sviluppo di applicazioni parallele in ASSIST avviene per mezzo di un linguaggio di coordinamento, chiamato ASSIST-CL; il modello di programmazione ha le seguenti caratteristiche:

- i programmi paralleli possono essere espressi da grafi generici;
- i moduli possono essere paralleli o sequenziali, con un'alta flessibilità per quanto riguarda la sostituzione di tali moduli in modo da modificare il grado di parallelismo;
- il modulo parallelo è rappresentato dal costrutto parmod, che può essere considerato una sorta di *skeleton* generico: può essere specializzato per emulare gli *skeleton* più comuni e può anche esprimere in modo semplice nuove forme di parallelismo (ad esempio, data parallel con stencil);
- i moduli sequenziali e paralleli hanno un proprio stato interno;
- i moduli paralleli possono avere un comportamento nondeterministico;
- la composizione di moduli paralleli e sequenziali è espressa tramite il meccanismo degli stream, che consentono di rappresentare in modo semplice le interfacce dei moduli.

La struttura di un programma ASSIST è quindi un grafo in cui i nodi sono i componenti e gli archi sono interfacce astratte, costituite da stream

(sequenze ordinate di lunghezza potenzialmente infinita di valori dello stesso tipo). I componenti sono espressi da moduli ASSIST, che possono essere paralleli (costrutto *parmod*) o sequenziali. Un modulo sequenziale è il componente più semplice esprimibile in ASSIST: ha uno stato interno e si attiva al ricevimento di un valore sugli stream di input secondo un comportamento deterministico a data-flow. Il *parmod* rappresenta un modulo parallelo di un programma ASSIST ed è costituito da tre sezioni: *Input Section* (IS), *Virtual Processors Section* (VPS) e *Output Section* (OS). La *Input Section* gestisce gli stream di input e si occupa di distribuire i dati ricevuti ai virtual processors (VP). La *Virtual Processors Section* consiste in un insieme di entità indipendenti e cooperanti tra loro chiamate VP, il cui compito è quello di eseguire la computazione parallela; per questo motivo la *VP Section* viene anche definita come il motore di calcolo del *parmod*. La *Output Section* infine si occupa di raccogliere il risultato della computazione eseguita sui VP, trasmettendolo poi su uno stream di output. Anche il modulo parallelo ha uno stato interno, che può essere partizionato o replicato sui VP.

ASSIST è stato implementato e utilizzato su architetture distribuite basate su griglie computazionali, e su cluster di PC o workstations. Si ritiene che i vantaggi offerti da un approccio strutturato possano ripercuotersi anche sulla programmazione parallela di processori *multicore*. Per approfondimenti riguardanti le implementazioni di ASSIST, si vedano [6, 7]. La parte più consistente del lavoro ha riguardato proprio l'implementazione del supporto per il costrutto *parmod* di ASSIST su Cell.

La tesi si articola in cinque capitoli. Il primo presenta innanzitutto l'architettura Cell, per poi continuare con una breve descrizione degli strumenti di programmazione forniti dalla IBM e concludere con una panoramica su alcune librerie a più alto livello. Il secondo capitolo descrive il linguaggio LC e la libreria MammuT. Il terzo capitolo presenta l'implementazione del costrutto *parmod* realizzata, e il quarto mostra come utilizzare la libreria per scrivere un semplice esempio. Il quinto capitolo analizza le prestazioni della

libreria in riferimento ad una applicazione di test. Il lavoro si conclude con alcune considerazioni finali riguardanti il lavoro svolto e i possibili sviluppi futuri.

Capitolo 1

Architettura del processore Cell

Cell è il nome di una tipologia di processori sviluppati da IBM in collaborazione con Sony e Toshiba ed è la prima implementazione dell'architettura chiamata *Cell Broadband Engine Architecture* (CBEA); in [12] viene presentato il processore Cell sia dal punto di vista architetturale che da quello prestazionale.

L'architettura CBEA permette varie configurazioni. Nel settembre 2006 la IBM presentò il suo primo server basato sul processore Cell, il BladeCenter QS20; il sistema, tuttora in commercio, è dotato di due Cell con 512MB di memoria ciascuno connessi in una configurazione NUMA, e la connettività esterna è garantita dalle interfacce di rete Gigabit e Infiniband. Poco dopo, nel novembre 2006, la Sony rilasciò la sua nuova console da gioco, la PlayStation 3; nonostante la PS3 non sia pensata per il calcolo ad alte prestazioni [1] viene comunque equipaggiata con una versione ridotta del processore Cell e il suo prezzo (attualmente di circa 400 euro) la rende una soluzione particolarmente attraente per costruire dei cluster di Cell. In [15] sono valutate le performance e analizzati i limiti della piattaforma PS3 nell'ambito dell'*high performance cluster computing*. Di recente è stato proposto un altro possibile utilizzo per un processore Cell, cioè come acceleratore delle parti computazionalmente più pesanti di un'applicazione ad alte prestazioni. Tale approccio,

[1] HPC, *High Performance Computing*

presentato in [14], prende il nome di *hybrid programming model* perchè prevede di distribuire l'esecuzione dell'applicazione tra processori eterogenei: un processo eseguito su un processore host, ad esempio un processore con architettura x86_64, crea un processo acceleratore su un processore acceleratore dedicato, come il PowerXCell8i. Il PowerXCell8i è una nuova implementazione dell'architettura CBEA che include una unità per le operazioni in doppia precisione migliorata e il supporto alle memorie DDR2 SDRAM. L'IBM ha già sviluppato un'implementazione hardware e un'infrastruttura software per abilitare il modello di computazione ibrido nell'ambito del progetto Roadrunner presso i Los Alamos National Laboratory (LANL).

In questa sezione sarà presentata l'implementazione del processore Cell attualmente commercializzata, utilizzata nella Playstation 3 di Sony e nei server basati su Cell di IBM.

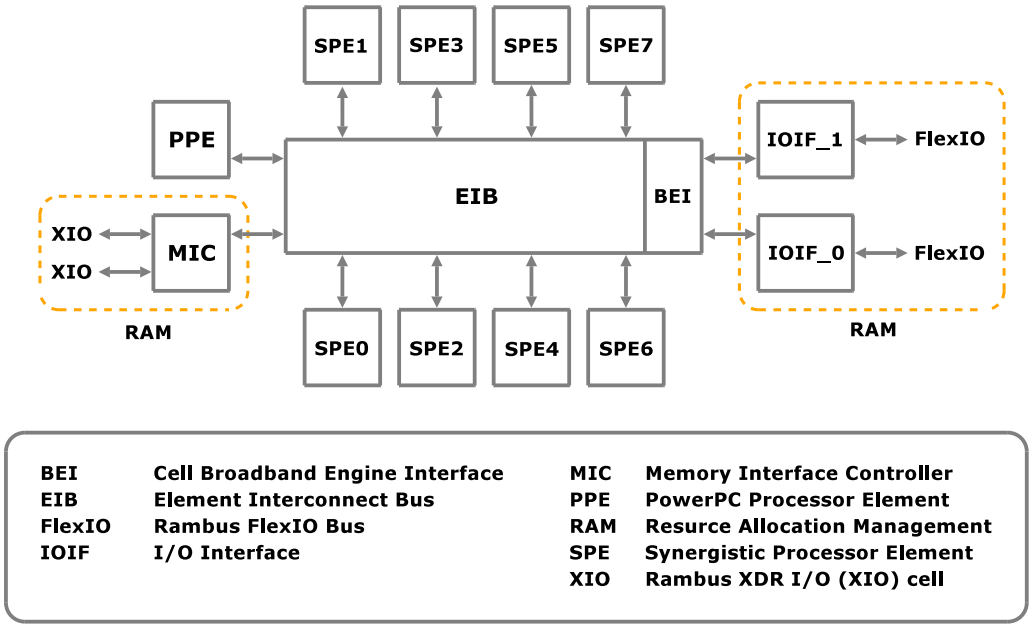

Figura 1.1: Architettura del processore Cell

Il sistema, schematizzato in figura 1.1, è composto da:

- 1 *PowerPC Processor Element* (PPE), operante alla frequenza di 3.2 GHz;

- 8 *Synergistic Processor Elements* (SPE), operanti alla frequenza di 3.2 GHz;

- un'interfaccia di memoria esterna chiamata *Memory Interface Controller* (MIC);

- 2 interfacce di Input/Output (IOIF_0 e IOIF_1);

- una rete di interconnessione chiamata *Element Interconnect Bus* (EIB) che connette il PPE, gli 8 SPE, la memoria e le interfacce di input/output (attraverso l'unità BEI, *Broadband Engine Interface*), la cui frequenza di funzionamento è pari alla metà della frequenza di funzionamento dei processori (1.6 GHz).

Il PPE rappresenta il processore principale del Cell e contiene un core tradizionale chiamato PPU (*PowerPC Processor Unit*) compatibile con la famiglia dei processori PowerPC di cui fa parte. Ogni SPE contiene un'unità chiamata SPU (*Synergistic Processor Unit*) e un'unità MFC (*Memory Flow Controller*).

Gli SPU sono processori più semplici rispetto al PPU, in quanto non offrono servizi come caching, memoria virtuale e supporto alla multiprogrammazione; sono caratterizzati da un set di istruzioni vettoriali.

L'interfaccia di memoria collega le unità di calcolo e le unità di Input/Output a una memoria Rambus XDR, che fornisce un'ampiezza di banda di picco di 25.6 Gbyte/s.

Le interfacce di I/O permettono di collegare unità di I/O alla CPU, utilizzando il protocollo IOIF. Inoltre, una delle 2 interfacce fornite può essere utilizzata per collegare fra loro due Cell, i cui nodi comunicheranno secondo lo stesso protocollo (chiamato BIF) utilizzato per la comunicazione tra nodi di un Cell singolo.

L'EIB è una rete di interconnessione formata da quattro anelli unidirezionali da 16 byte arbitrati da un arbitro centralizzato. Il flusso dei dati è in senso orario per due di questi anelli e in senso antiorario per i restanti due. Ogni anello è in grado di gestire fino a tre trasferimenti simultanei.

Il set di istruzioni del PPU è diverso da quello degli SPU: per questo motivo i programmi del PPE devono essere compilati con un compilatore distinto da quello utilizzato per compilare i programmi SPE. L'allocazione di un processo su un SPE deve essere effettuata da un processo in esecuzione sul PPE che, mediante opportuni comandi, trasferirà l'immagine eseguibile del processo da allocare nella memoria locale del SPE, e imposterà il valore iniziale del progam counter. La terminazione del processo PPE provoca l'arresto dell'applicazione concorrente: sarà quindi necessario che il processo PPE attenda la terminazione dei processi da esso generati, per evitare una terminazione prematura dell'applicazione concorrente.

1.1 PowerPC Processor Element

Il PPE comprende al suo interno un core RISC [2] PowerPC a 64 bit, dotato di coprocessore VMX per il calcolo vettoriale, conforme allo standard *PowerPC Architecture, version 2.02*. Il PPE esegue il sistema operativo e svolge il ruolo di coordinatore delle risorse del sistema. Come mostrato in figura 1.2 esso è composto da due parti fondamentali: *PowerPC Processor Unit* (PPU) e *PowerPC Processor Storage Subsystem* (PPSS).

[2] *Reduced Instruction Set Computer*, è un approccio secondo il quale l'utilizzo di un set di istruzioni ridotto, poche e semplici istruzioni, aiuta la progettazione di un calcolatore avente una struttura regolare quindi più facile da realizzare e meno costoso; un approccio opposto è il *Complex Instruction Set Computer* (CISC) in cui un set di istruzioni più complesso riduce il numero di istruzioni macchina necessarie a realizzare un programma ad alto livello.

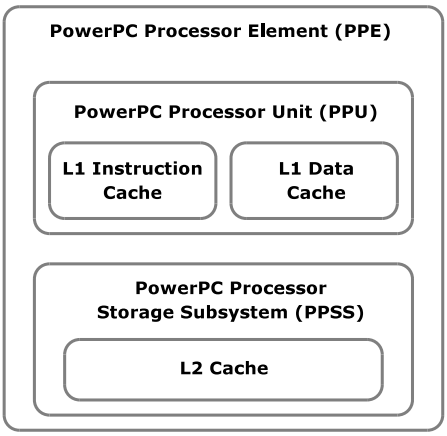

Figura 1.2: Componenti PPE

Il PPU è un core PowerPC tradizionale con modello di esecuzione *in-order* [3], è *dual threaded* e superscalare, ossia supporta l'esecuzione simultanea di due thread ed è in grado di eseguire due istruzioni indipendenti in parallelo. È dotato di due cache di primo livello, una per i dati e l'altra per le istruzioni, di 32 KB ciascuna. È capace di due modalità di funzionamento, a 32 e a 64 bit. IBM suggerisce l'utilizzo del core PPE per le mansioni di controllo, e degli SPE per il calcolo e il trattamento dei dati.

Il PPSS gestisce le richieste di accesso alla memoria dal PPU e richieste provenienti dall'esterno verso il PPE come per esempio richieste dalle unità di I/O e dagli SPE. Il PPSS contiene una memoria cache di secondo livello unificata per dati e istruzioni di 512 KB; la dimensione di un blocco per la gerarchia di memoria cache L2-L1 è di 128 byte. Sono supportati i modelli di gestione della memoria virtuale con paginazione e segmentazione.

[3]Modalità opposta all'esecuzione *out-of-order* in cui le istruzioni e i comandi non vengono necessariamente eseguiti nello stesso ordine in cui sono stati schedulati.

1.2 Synergistic Processor Element

Ognuno degli 8 SPE contiene al suo interno un core RISC a 128 bit detto *Synergistic Processor Unit* (SPU), una memoria privata detta *Local Storage* (LS) di 256 KB e l'unità di accesso alla memoria detta *Memory Flow Controller* (MFC). Ogni SPU è dotato di un nuovo set di istruzioni vettoriali simile a quello del coprocessore VMX, particolarmente adatto per il calcolo intensivo su dati vettoriali. L'SPU è un processore indipendente dotato di proprio contatore istruzioni e ottimizzato per eseguire threads creati e gestiti dal PPE; il core preleva le istruzioni dalla propria memoria privata, LS, in cui sono memorizzati anche i dati, cui accede con l'utilizzo di istruzioni di load e store mediante indirizzi fisici. L'accesso alla memoria principale avviene esclusivamente tramite trasferimenti DMA, gestiti dall'unità MFC. Il core è *in-order* ed è dotato di due pipeline per l'esecuzione delle istruzioni, chiamate *even pipeline* e *odd pipeline*. Le istruzioni sono assegnate a una delle due pipeline a seconda del loro tipo: le istruzioni che effettuano calcolo in virgola fissa o mobile sono eseguite sulla *even pipeline*, le restanti sulla *odd pipeline*. Il processore può eseguire due istruzioni simultaneamente, una per pipeline. Perchè questo avvenga, il compilatore deve garantire l'alternarsi di istruzioni da eseguire sulla *odd pipeline* e sulla *even pipeline*. Un'eccezione a questa regola sono le istruzioni di calcolo in precisione doppia, che sono eseguite in pipeline solo parzialmente e sono soggette ad una latenza superiore rispetto alle altre istruzioni. Il processore non fornisce funzionalità di *branch prediction* [4] e memoria virtuale, è presente comunque un'istruzione di *branch hinting* che, se eseguita con un anticipo sufficiente, permette di annullare l'effetto del salto.

Il *Memory Flow Controller* interfaccia l'SPU sulla struttura di interconnessione EIB, gestisce i trasferimenti di memoria tra la memoria locale dello SPE cui appartiene e ogni altra risorsa connessa attraverso la rete di inter-

[4] Funzionalità che cerca di prevedere l'esito di un'operazione su cui si basa un'istruzione di salto condizionato.

connessione del chip (memoria principale, memoria locale di un altro SPE, dispositivo di I/O), nonchè le comunicazioni interprocessor, di cui si parlerà nel seguito.

L'unità MFC esegue comandi inviati dal processore, che possono essere rappresentati logicamente da una tupla

```
(operazione, indirizzo fisico, indirizzo logico, tag)
```

dove `indirizzo fisico` riferisce una locazione della memoria locale, e `indirizzo logico` riferisce una locazione di una delle memorie esterne. I trasferimenti di memoria sono effettuati da un controllore DMA, che possiede due code di comandi, una in cui sono memorizzati i comandi DMA generati all'interno dello stesso SPE, chiamata *SPU command queue* e un'altra in cui sono memorizzati i comandi inviati da unità esterne, quali il PPE, gli altri SPE o una unità di I/O, chiamata *proxy queue*. L'unità MFC è in grado di gestire trasferimenti di DMA cui gli indirizzi di base delle aree di memoria sorgente e destinazione siano allineati ai 16 byte. Le dimensioni dei blocchi di memoria supportate per i trasferimenti DMA sono 1, 2, 4, 8 e 16 byte, e tutti i multipli di 16 byte fino a 16 KB. Trasferimenti di dimensioni maggiori possono essere schedulati utilizzando il meccanismo delle liste di comandi DMA. Questi comandi possono essere inseriti solo nella *SPU command queue*, e non nella *proxy queue*. Una lista di comandi può contenere fino a 2048 comandi DMA, e consente quindi di trasferire fino a 32 MB di memoria. Nonostante sia sufficiente un allineamento ai 16 byte degli indirizzi delle aree di memoria sorgente e destinazione, il produttore suggerisce, per incrementare la performance del trasferimento, di utilizzare aree di memoria con indirizzi di base allineati ai 128 byte, in quanto gestiti più efficientemente dal protocollo della rete di interconnessione.

Nella *SPU command queue* possono essere memorizzati fino a 16 comandi di DMA, mentre nella *proxy queue* solamente 8. È quindi preferibile, quando possibile, schedulare i comandi internamente al SPE, piuttosto che utilizzare il meccanismo delle proxy queue e, in caso di trasferimenti di dimensioni mag-

giori ai 16 KB, ricorrere al meccanismo delle liste dei comandi piuttosto che a una serie di comandi semplici, in modo tale da occupare una sola entry della coda dei comandi. L'unità MFC gestisce in modo autonomo e *out-of-order* le richieste di trasferimenti dati; attraverso il campo `tag`, che rappresenta un'etichetta data al trasferimento dal programmatore, è possibile effettuare vari tipi di sincronizzazione sul completamento dei comandi, come attendere la terminazione di tutti i comandi etichettati con un certo `tag` o schedulare un comando in *fence* o in *barrier* con altri comandi DMA aventi lo stesso `tag`. Un comando in *fence* non può essere eseguito fino a che tutti i comandi precedentemente schedulati con lo stesso `tag` non sono terminati; tuttavia è possibile che comandi con lo stesso `tag` schedulati successivamente vengano eseguiti prima. Un comando in *barrier* non può essere eseguito fino a quando tutti i comandi precedenti con lo stesso `tag` non sono terminati e i successivi comandi con lo stesso `tag` non verranno eseguiti fino a quando il comando in *barrier* non sarà terminato.

1.3 Element Interconnect Bus

La rete di interconnessione EIB è costituita da 4 anelli circolari unidirezionali ampi 16 byte ciascuno che funzionano alla metà della frequenza del processore; in due degli anelli i dati fluiscono in senso orario e nei restanti due in senso antiorario. I messaggi sono divisi in pacchetti da 128 byte. La rete è gestita secondo una modalità connection oriented ed è presente un arbitro che regola l'accesso al bus e processa le richieste scegliendo l'anello che dovrà soddisfare il trasferimento dati utilizzando un algoritmo di routing che garantisce che un pacchetto non effettui un numero di hop maggiore della metà del diametro della rete.

Ogni anello permette un massimo di tre trasferimenti dati contemporanei, a condizione che i loro percorsi non siano sovrapposti; questo permette un massimo di 12 trasferimenti simultanei sui quattro anelli. La banda massima

teorica complessiva della rete è di 204.8 GB/s. La rete EIB è analizzata più dettagliatamente in [11, 13] dove, tra le altre cose, viene mostrato come sia possibile ottenere una banda di 25.6 GB/s sui trasferimenti di memoria fra memorie locali e fra una memoria locale e la memoria principale.

1.4 Comunicazioni Interprocessor

L'architettura prevede due tipi di comunicazioni interprocessor:

- canali dei segnali;
- mailbox.

Ogni SPE ha associati due canali di segnali su cui è possibile ricevere messaggi di 32 bit. Ogni SPU può leggere i canali `Sig_Notify_1` e `Sig_Notify_2` utilizzando le funzioni bloccanti `SPU_RdSigNotify1` e `SPU_RdSigNotify2`. Il PPE o gli altri SPE possono scrivere i canali dei segnali utilizzando gli indirizzi mappati in memoria. I messaggi vengono inviati dai vari SPE, attraverso l'MFC, utilizzando appositi comandi DMA; è possibile associare un `tag` al comando per inviare le segnalazioni in *fence* o in *barrier* proprio come avviene per gli altri comandi DMA. I canali dei segnali possono essere utilizzati in scrittura secondo due modalità: *Overwrite* o *OR*. Nel primo caso il valore del messaggio viene sovrascritto, con l'eventualità di perdere il valore di un messaggio precedentemente inviato ma non ancora letto. In modalità *OR* invece l'invio di un messaggio su un canale dei segnali provoca la modifica del valore contenuto nel canale: se `old_v` era il valore contenuto e `msg` il valore del messaggio inviato, il nuovo valore contenuto nel canale sarà `new_v = old_v | msg`, dove `|` rappresenta l'operatore di *OR bit a bit*.

Ad ogni SPE sono inoltre associate una mailbox in entrata e due in uscita su cui è possibile inviare e ricevere messaggi di una parola di 32 bit. La mailbox in entrata può ospitare quattro entry, ognuna di 32 bit, mentre le

mailbox in uscita possono ospitare un solo messaggio. Le operazioni di lettura/scrittura da parte degli SPE sono bloccanti. Una delle due mailbox in uscita, chiamata *SPU write outbound interrupt mailbox*, invia un'interruzione al PPE ad ogni operazione di scrittura. Utilizzando questo canale è possibile per il PPE effettuare una lettura bloccante del messaggio mentre la lettura dall'altro canale non è bloccante per il PPE. Il PPE utilizza gli indirizzi dei canali delle mailbox mappati in memoria per scrivere e leggere queste; allo stesso modo un SPE può scrivere una mail ad un altro SPE utilizzando l'indirizzo della mailbox corrispondente mappato in memoria nello spazio di indirizzamento globale. L'invio di mail da SPE a SPE consiste in un trasferimento DMA.

1.5 Considerazioni sull'architettura

Il Cell può di fatto essere visto e utilizzato come una macchina NUMA (*Non Uniform Memory Access*). Una macchina NUMA è così definita in [8]:
L'architettura multiprocessore a processori dedicati consiste essenzialmente di un certo numero di nodi di elaborazione completi interconnessi tramite uno spazio di memorizzazione comune e da strutture di comunicazione diretta.

Come descritto nella sezione 1.2, ogni SPE contiene al suo interno un processore ausiliario, il *Memory Flow Controller* (MFC), il quale contiene al suo interno una MMU, che effettua la traduzione da indirizzi logici ad indirizzi fisici. Un SPE, attraverso il proprio MFC, può quindi accedere con indirizzi logici sia alle memorie locali degli altri SPE che alla memoria principale; tuttavia un SPE accede alla propria memoria locale solo con indirizzi fisici.

Il processo `P` allocato sul PPE può accedere alle memorie locali degli SPE, in quanto queste sono mappate nello spazio di indirizzamento di `P`; le memorie locali possono inoltre essere accedute mediante indirizzi logici dalle unità MFC degli SPE, i quali condividono la tabella di rilocazione con `P`. L'insie-

me delle memorie locali può quindi essere visto come un'area di memoria condivisa fra le MFC dei vari SPE. Gli SPU possono essere considerati dei semplici esecutori di istruzioni, alla stregua di coprocessori delle MFC, che costituiscono i nodi dell'architettura NUMA. Infine la rete EIB costituisce la struttura di interconnessione necessaria per poter ricondurre la macchina ad un'architettura NUMA, in quanto offre il supporto per i trasferimenti in memoria condivisa.

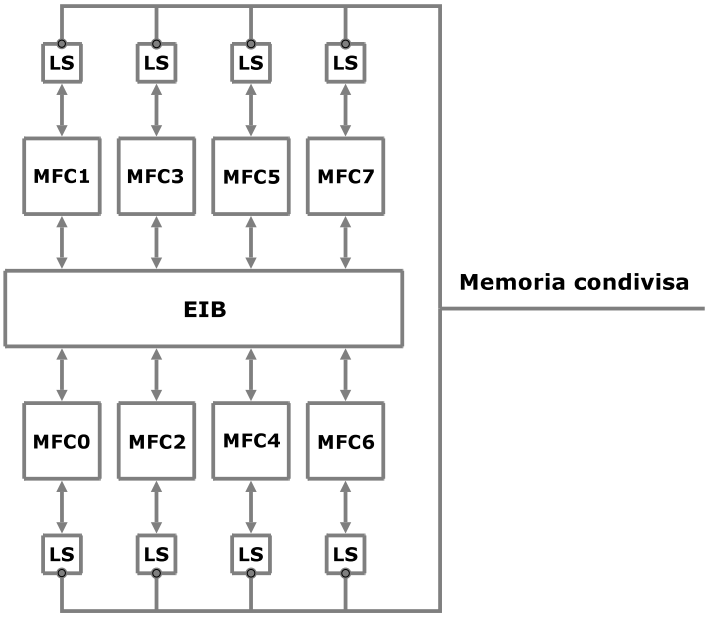

Figura 1.3: Il Cell come macchina NUMA

Secondo questa visione dell'architettura, la memoria principale può essere considerata come una unità di I/O condivisa dai vari nodi, o come un nodo la cui unica funzione è la gestione della memoria stessa, cui è possibile fare richieste di lettura/scrittura. Lo schema in figura 1.3 evidenzia le componenti della macchina NUMA.

1.6 IBM Cell SDK

In questa sezione viene mostrato il modello di programmazione fornito dal IBM SDK per scrivere un'applicazione per il Cell BE.

1.6.1 Contesti SPE

Il processore Cell/BE ha un singolo PPE previsto principalmente per l'esecuzione del sistema operativo, il controllo dell'applicazione, la gestione delle risorse e la gestione dei thread SPE. L'esecuzione di qualsiasi applicazione parte su questo processore. Le applicazioni non possono interagire direttamente con le risorse fisiche degli SPE, in quanto queste sono gestite dal sistema operativo. L'esecuzione di codice sugli SPE viene gestita tramite la libreria *libspe* (*SPE Runtime management library*), che mette a disposizione l'astrazione dei *contesti SPE*, rappresentazioni logiche degli SPE. Il sistema operativo schedulerà i contesti sugli SPE fisici.

Esistono due modalità per il caricamento di programmi su SPE.

SPU embed library
: Il codice SPE è incluso direttamente nell'eseguibile PPE. A tempo di esecuzione, il programma è acceduto tramite un puntatore esterno che può essere utilizzato dal programmatore per caricare il programma nel *local storage*.

Caricamento da file
: Il programma SPE viene compilato in un file separato, e il file viene caricato a tempo di esecuzione e mappato in memoria.

In entrambi i casi si ha a disposizione un puntatore alla locazione di memoria in cui è stato mappato il codice SPE. Il caricamento del programma sul SPE è implementato nella libreria *libspe* tramite un trasferimento in DMA.

Le principali funzioni per la gestione dei contesti SPE sono le seguenti:

```
#include <libspe2.h>

spe_gang_context_ptr_t spe_gang_context_create(unsigned int flags);

int spe_gang_context_destroy(spe_gang_context_ptr_t gang);

spe_context_ptr_t spe_context_create(unsigned int flags,
                                     spe_gang_context_ptr_t gang);

int spe_context_destroy(spe_context_ptr_t spe);

spe_program_handle_t *spe_image_open (const char *filename);

int spe_image_close(spe_program_handle_t *program);

int spe_program_load(spe_context_ptr_t spe,
                     spe_program_handle_t *program)

int spe_context_run(spe_context_ptr_t spe,
                    unsigned int *entry, unsigned int runflags,
                    void *argp, void *envp,
                    spe_stop_info_t *stopinfo)
```

La funzione `spe_gang_context_create` consente di creare una *gang*, cioè un costrutto che permette raggruppare un insieme di contesti SPE. Il puntatore alla *gang* creata viene utilizzato per la creazione dei contesti SPE tramite la funzione `spe_context_create`, in modo di associare il contesto alla gang. Le funzioni `spe_gang_context_destroy` e `spe_context_destroy` servono per liberare le risorse allocate rispettivamente per una gang e per un contesto SPE. I programmi SPE possono essere caricati da file tramite la funzione `spe_image_open`. Tramite la funzione `spe_program_load` è possibile caricare un programma SPE su un determinato contesto SPE; il puntatore `spe_program_handle_t *program` può essere un puntatore esterno (se si sta utilizzando il metodo delle *SPU embed library*) o un puntatore ottenuto

tramite una chiamata precedente alla funzione `spe_image_open`. Nel caso di programma SPE caricato dinamicamente da file, le risorse allocate per tenere in memoria l'eseguibile SPE possono essere liberate tramite la funzione `spe_image_close`. Infine, la funzione `spe_context_run` viene utilizzata per eseguire un contesto SPE. Questa funzione è bloccante, perciò per ogni contesto SPE l'applicazione deve prevedere un thread sul PPE, dedicato al lancio del contesto.

1.6.2 Gerarchia di memoria e trasferimenti

Nella tabella 1.1 sono indicati i comandi per i trasferimenti DMA. Questi comandi appartengono ad una serie di estensioni per il C chiamate *intrinsics*, che rendono disponibili al programmatore C le funzionalità delle istruzioni assembler appartenenti all'*istruction set* degli SPE.

Comando	**SPE**	**PPE**
Comandi put		
`put`	`mfc_put`	`spe_mfc_put`
`putf`	`mfc_putf`	`spe_mfc_putf`
`putb`	`mfc_putb`	`spe_mfc_putb`
`putl`	`mfc_putl`	
`putlf`	`mfc_putlf`	
`putlb`	`mfc_putlb`	
Comandi get		
`get`	`mfc_get`	`spe_mfc_get`
`getf`	`mfc_getf`	`spe_mfc_getf`
`getb`	`mfc_getb`	`spe_mfc_getb`
`getl`	`mfc_getl`	
`getlf`	`mfc_getlf`	
`getlb`	`mfc_getlb`	

Tabella 1.1: Comandi DMA supportati dalla MFC

La direzione del trasferimento prende l'SPE come punto di riferimento, perciò ad esempio un comando `get` indica un trasferimento del blocco puntato dall'*effective address* nel *local storage*, mentre un comando `put` indica il trasferimento del blocco puntato dal *local storage address* nella posizione dell'*effective address*. I trasferimenti DMA sono asincroni, e l'unità MFC può eseguire i comandi in un ordine diverso rispetto alla loro posizione nella coda. Per introdurre un parziale ordinamento, è possibile utilizzare i comandi con i suffissi `f` e `b`, ad indicare trasferimenti in *fence* o *barrier* (vedi sezione 1.2).

Con un utilizzo sapiente dei trasferimenti in DMA tra memoria principale e memoria locale degli SPE, il programmatore è in grado di sovrapporre completamente la comunicazione al calcolo. Tuttavia le problematiche legate a un utilizzo efficiente della memoria locale sono molteplici; in [19] questi aspetti vengono affrontati nel dettaglio.

1.6.3 Vettorizzazione

Gli SPE sono processori RISC con un set di istruzioni SIMD (*Synergistic Processor Unit Instruction Set Architecture*) che lavorano su operandi da 128 bit. Sono sprovvisti della logica relativa alla *branch prediction* e all'esecuzione fuori ordine delle istruzioni. Anche la logica di traduzione degli indirizzi è assente, perchè gli SPE accedono con operazioni di *load* e *store* esclusivamente tramite indirizzi fisici all'interno del proprio *local storage*, mentre gli accessi in memoria principale o su altri LS vengono effettuati tramite l'unità MFC. Operazioni su operandi scalari sono possibili, ma vengono tradotte dal compilatore in una sequenza di operazioni vettoriali, e per questo motivo sono molto meno efficienti.

I programmi SPE possono essere scritti in C o C++ senza particolari modifiche, ma per ottenere buone performance occorre considerare le peculiarità del processore e scrivere il programma sfruttando opportuni *intrinsics*. Il programmatore gradualmente può modificare il programma originale tra-

mite la *vettorizzazione* del codice. I tipi SIMD disponibili sono tutti da 128 bit e contengono da 1 a 16 elementi.

Tipo	**Contenuto**
`vector unsigned char`	16 unsigned char (8 bit ciascuno)
`vector signed char`	16 signed char (8 bit ciascuno)
`vector unsigned short`	8 unsigned short (16 bit ciascuno)
`vector signed short`	8 signed short (16 bit ciascuno)
`vector unsigned int`	4 unsigned int (32 bit ciascuno)
`vector signed int`	4 signed int (32 bit ciascuno)
`vector unsigned long long`	4 unsigned (64 bit ciascuno)
`vector signed long long`	4 signed (64 bit ciascuno)
`vector float`	4 float (32 bit ciascuno)
`vector double`	2 double (64 bit ciascuno)
`qword`	16 byte

Tabella 1.2: Tipi vettoriali

Le operazioni possibili tramite gli *intrinsics* comprendono istruzioni di cast, matematiche (`spu_add`, `spu_mul` ...), operazioni su byte (`spu_shuffle` ...) e operazioni logiche (`spu_and`, `spu_or` ...).

L'assenza di supporto per la *branch prediction* sugli SPE viene risolta mediante direttive `__builtin_expect` che indicano al compilatore il ramo del branch più probabile. Ad esempio:

```
if(__builtin_expect((a > b),0)) {
    ...
} else {
    ...
}
```

In questo caso il programmatore predice che l'espressione all'interno dell'`if` è falsa, garantendo che il ramo corrispondente verrà eseguito senza salti, e

quindi senza introdurre overhead.

Ulteriori approfondimenti riguardo le tecniche di ottimizzazione applicabili al codice dei programmi SPE a livello di compilatore sono discusse in [20].

1.6.4 Comunicazioni inter-processor

Il Cell/BE contiene diversi meccanismi per la comunicazione tra PPE ed SPE, e tra SPE ed SPE, implementati nell'unità MFC. I due meccanismi principali sono le *mailbox* e i canali dei segnali, esposti nella sezione 1.4.

Mailbox

Un programma SPE può accedere alla propria mailbox tramite funzioni `spu_*_mbox` definite nell'header `spu_mfcio.h`. Un programma PPE può accedere alla mailbox di un SPE tramite le funzioni `spe_*_mbox*`, dichiarate nell'header `libspe.h`.

Mailbox	**Funzione**	**Bloccante**
SPE		
write outbound mailbox	`spu_write_out_mbox`	Si
write outbound int. mailbox	`spu_write_out_intr_mbox`	Si
read inbound mailbox	`spu_read_in_mbox`	Si
PPE		
write outbound mailbox	`spe_write_out_mbox`	No
write outbound int. mailbox	`spe_write_out_intr_mbox`	Si/No
read inbound mailbox	`spe_read_in_mbox`	Si/No

Tabella 1.3: Funzioni di accesso alle mailbox

Mailbox	**Funzione**
SPE	
write outbound mailbox	`spu_stat_out_mbox`
write outbound int. mailbox	`spu_stat_out_intr_mbox`
read inbound mailbox	`spu_stat_in_mbox`
PPE	
write outbound mailbox	`spe_out_mbox_status`
write outbound int. mailbox	`spe_out_intr_mbox_status`
read inbound mailbox	`spe_in_mbox_status`

Tabella 1.4: Funzioni per la lettura dello stato delle mailbox

Canali dei segnali

Un programma SPE legge i registri dei segnali locali attraverso la propria unità MFC, e può inviare un segnale ad un altro SPE tramite i comandi `sndsig`, `sndsigf` (invio di un segnale in *fence*) e `sndsigb` (invio di un segnale in *barrier*). I comandi sono implementati attraverso `put` DMA.

Un programma SPE può leggere i propri registri dei segnali tramite le funzioni `spu_read_signal*` e `spu_stat_signal*` dichiarate nell'header `spu_mfcio.h`. Il programma SPE può scrivere i registri dei segnali di altri SPE tramite le funzioni `mfc_sndsig*` dichiarate nell'header `spu_mfcio.h`. Il programma PPE può accedere ai registri dei segnali degli SPE utilizzando la funzione `spe_signal_write`. La modalità di scrittura dei registri dei segnali può essere impostata passando i flag `SPE_CFG_SIGNOTIFY1_OR` e `SPE_CFG_SIGNOTIFY2_OR` alla funzione `spe_context_create` per abilitare la modalità OR [5] all'atto della creazione del contesto.

[5] La modalità Overwrite è la modalità di default.

1.7 Framework esistenti

L'architettura Cell ha destato fin da subito grande interesse in ambito scientifico. Questo ha portato a un numero sempre crescente di studi e pubblicazioni su come sfruttare al massimo le potenzialità offerte dal processore, oltre allo sviluppo di una serie di tool che consentisse di ridurre la complessità astraendo dai livelli più bassi dell'architettura. Questa sezione offrirà una breve panoramica su alcuni di essi.

1.7.1 RapidMind

La piattaforma di sviluppo RapidMind, presentata in [16], è un framework che consente di esprimere computazioni data-parallel all'interno di codice C++ ed eseguirle efficientemente su processori multicore. In realtà le applicazioni realizzabili con la libreria corrispondono solo a semplici *Map* mentre non sono previste forme di data-parallel con stencil. Lo scopo che si prefigge la libreria è di fornire agli sviluppatori un modello di programmazione parallela che sia semplice da usare e portabile. Semplice nel senso che si vuole svincolare gli sviluppatori dai dettagli relativi alla gestione a basso livello dell'architettura, permettendogli di concentrarsi esclusivamente sullo sviluppo degli algoritmi paralleli. Portabile perché, proprio grazie a questo meccanismo di astrazione dei dettagli dell'architettura, è possibile scrivere un unico programma parallelo e farlo poi girare su architetture diverse come Cell BE, GPU o CPU multicore. In figura 1.4 è possibile vedere uno schema dell'architettura di RapidMind.

Il processore host è quello che esegue il sistema operativo mentre i dispositivi target rappresentano l'hardware che sarà utilizzato dalla piattaforma per accelerare l'applicazione. Nel caso del Cell BE, l'host sarà il PPE e i dispositivi target saranno gli SPE. RapidMind ha un'interfaccia C++ per descrivere la computazione, anziché un linguaggio separato. Tale interfaccia

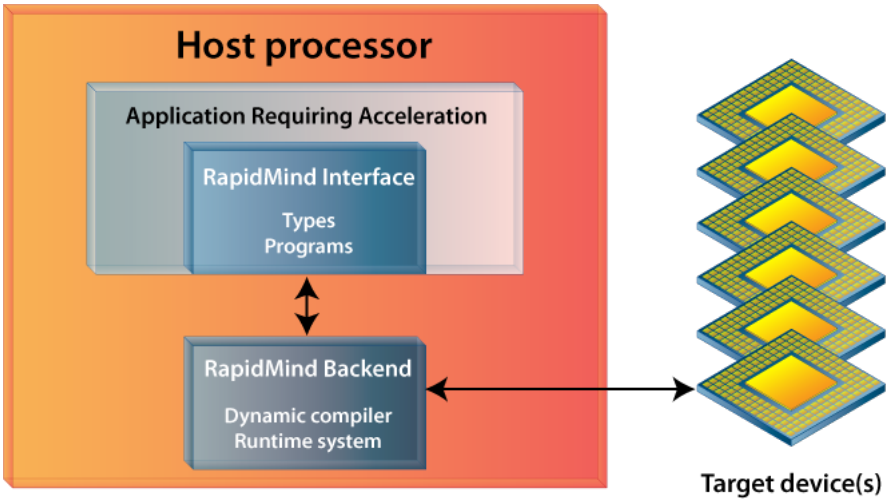

Figura 1.4: Architettura RapidMind

è implementata usando solo costrutti del C++ standard; per utilizzarla in un programma è sufficiente includere un file header e linkare una libreria: in questo modo il programmatore ha a disposizione tre nuovi tipi fondamentali chiamati *Value*, *Array* e *Program*. I primi due sono contenitori per dati mentre il terzo è un contenitore per operazioni (rappresenta una computazione da eseguire in parallelo). Ogni architettura supportata è gestita dalla piattaforma tramite un diverso backend, che si occupa della gestione delle comunicazioni tra processore host e dispositivi target, dei trasferimenti di dati, e di problemi come sincronizzazione e bilanciamento del carico. Il backend comprende un runtime che si occupa di tutti questi compiti, oltre ad un componente per la compilazione dinamica del codice che consente di ottenere le massime prestazioni su qualsiasi hardware. Attualmente vengono forniti tre diversi backend per architetture Cell BE, GPU e CPU multicore.

Saranno ora analizzati brevemente i tipi cui si è accennato precedentemente. Questi tipi sono in realtà delle classi, implementate come template.

`Value<N, T>`

Rappresenta una `N`-upla di valori di tipo `T`, dove `T` può essere un tipo

numerico di base: floating point a singola o doppia precisione, intero con o senza segno. E' possibile utilizzare praticamente tutti gli operatori aritmetici, logici, di confronto e bitwise, che sono stati opportunamente overloadati. Sono disponibili inoltre funzioni trigonometriche, logaritmiche ed esponenziali.

`Array<D, T>`

Rappresenta un contenitore multidimensionale per dati. Il parametro `D` rappresenta il numero di dimensioni dell'array e può essere 1, 2, o 3; il parametro `T` indica il tipo degli elementi. Attualmente, i tipi validi sono ristretti alle varie istanze della classe `Value<N, T>`. Le istanze di questa classe supportano gli operatori `[ ]` e `( )` per l'accesso random. L'operatore `[ ]` usa coordinate intere, partendo da 0. L'operatore `( )` invece, usa coordinate reali nel range `[0, 1]` in ciascuna delle sue dimensioni. Sia gli array che i value seguono una semantica di passaggio per valore. Ci sono altri due tipi simili agli array e sono detti array *references* (`ArrayRef<D, T>`) e *accessors* (`ArrayAccessor<D, T>`). Gli accessors consentono di riferire solo alcune porzioni di un array. La classe `ArrayAccessor` funziona in modo molto simile a un iteratore del C++. L'assegnamento agli elementi dell'accessor va a modificare gli elementi corrispondenti dell'array che l'accessor "vede". Esistono delle funzioni che consentono di accedere ad una porzione dell'array che rendono proprio un oggetto di tipo `ArrayAccessor`. Alcune di queste sono: `take`, `slice`, `offset`, `stride`. A volte è utile poter riferire indifferentemente un array o un accessor: la classe `ArrayRef` esiste proprio per questo. Assegnare un array o un accessor a un reference fa in modo che questo punti all'oggetto assegnato, più o meno come farebbe un puntatore del C++. Esiste ancora un altro tipo di array chiamato virtual array. Questo tipo di array ha la caratteristica di non occupare spazio in memoria perché generato al volo quando viene usato e risulta spesso utile come stream di input di un `Program`. Un virtual array non contiene dati ma pattern regolari. Una funzione che genera

un array virtuale è la grid che, come forse si può intuire dal nome, rende una griglia di interi contigui.

`Program`

Un oggetto di questo tipo memorizza una sequenza di operazioni. Queste operazioni sono specificate passando alla cosiddetta *retained mode*, indicata con la macro `BEGIN`. Prima di continuare a vedere i `Program` però, è necessario aprire una parentesi e parlare delle modalità di esecuzione del codice. Esistono due modalità in cui le istruzioni di un programma possono essere eseguite: *immediate mode* e *retained mode*. Normalmente il sistema opera in *immediate mode*: le operazioni su una tupla di valori hanno luogo quando specificato (come in una normale libreria vettoriale), la computazione viene eseguita sullo stesso processore che esegue il programma host e il risultato viene memorizzato nella tupla di output. Entrando in *retained mode* viene creato un nuovo oggetto della classe `Program`, che viene restituito dalla macro `BEGIN`. In questa modalità le operazioni sulle tuple non vengono eseguite immediatamente ma vengono invece valutate simbolicamente e memorizzate nell'oggetto della classe `Program` in uno speciale linguaggio di rappresentazione interna di RapidMind chiamato IR (*RapidMind-specific Internal Representation*). Si esce dalla modalità *retained* usando la macro `END`, che chiude l'oggetto `Program` e lo segna come pronto per la compilazione. Al momento della prima esecuzione, esso dovrà essere compilato nel linguaggio macchina specifico dell'hardware che si sta utilizzando. A tale scopo sarà usato il compilatore dinamico presente nel corrispondente backend. A questo punto l'oggetto è pronto e può essere usato per la computazione. E' bene precisare che solo le operazioni che coinvolgono i tipi value e array di RapidMind vengono memorizzate in un oggetto `Program` durante l'esecuzione in *retained mode*; le operazioni sui tipi del C++ vengono invece eseguite immediatamente (ad esempio, una variabile di un tipo del C++ sarà valutata e il suo valore sarà memorizzato nel `Program` come costante). Un oggetto

di tipo `Program` deve necessariamente avere almeno un input e un output (ovviamente può averne anche di più). Per dichiararli si usano dei template: `In<T>` e `Out<T>` per indicare rispettivamente un input e un output di tipo `T`. All'interno di un `Program` è possibile utilizzare variabili non locali, dichiarate cioè all'esterno della definizione del `Program` stesso. Queste variabili prendono il nome di uniform. La piattaforma si preoccuperà di memorizzare per ogni `Program` da quali dati dipende e di distribuire i valori aggiornati di tali uniform quando necessario. Gli oggetti di tipo `Program` hanno una semantica di esecuzione ristretta. In particolare, possono leggere qualsiasi variabile dichiarata in *immediate mode* ma non possono modificare il valore di una variabile che non sia locale (o meglio, possono farlo ma le modifiche non saranno visibili all'esterno del `Program`). Gli unici output permessi sono quelli esplicitamente indicati col template `Out<T>`. Questo significa che i `Program` possono essere visti dall'esterno come funzioni pure. Inoltre i `Program` possono contenere istruzioni per il controllo del flusso e possono richiamare altri programs o funzioni. Una cosa importante da notare è che la definizione degli oggetti di tipo `Program` avviene al momento dell'esecuzione del programma C++. Le istruzioni di controllo del flusso del C++ possono manipolare la generazione del codice, nel senso che sono in grado di stabilire quali operazioni saranno registrate in *retained mode* e quali no ma non risulteranno in un vero controllo del flusso al momento dell'esecuzione del program. Per avere un controllo del flusso dinamico, che dipenda dai valori calcolati all'interno del program, è necessario utilizzare delle apposite macro:

- `IF` / `ELSEIF` / `ELSE` / `ENDIF`
- `FOR` / `ENDFOR`
- `WHILE` / `ENDWHILE`
- `DO` / `UNTIL`

È interessante vedere come si possa paragonare l'uso del costrutto `if` / `else` del C++ all'interno di un `Program` con l'uso delle direttive del preprocessore `#ifdef` / `#else` / `#endif` all'interno di un normale programma C / C++.

1.7.2 BlockLib

BlockLib è una libreria, presentata in [17], che tenta di astrarre dalla complessità della programmazione su Cell mediante un approccio a skeleton. Gli skeleton forniti consentono di esplicitare i pattern di computazione data-parallel più elementari come *Map*, *Reduce* e un altro pattern che gli autori definiscono *map-with-overlap*, che corrisponde a un data-parallel con stencil fisso in cui per calcolare il valore di un determinato elemento di un array si ha bisogno anche degli elementi vicini. L'obiettivo della libreria è quello di semplificare il lavoro del programmatore fornendo meccanismi che automatizzano la gestione della memoria e la parallelizzazione della computazione. Il programmatore deve fornire la funzione da applicare sui dati e il supporto si occupa di gestire la computazione scatterizzando automaticamente l'array di elementi tra gli SPE.

1.7.3 MPI microtask

Il modello di programmazione MPI microtask, presentato in [18], è basato sullo standard di programmazione MPI (*Message Passing Interface*) per macchine parallele a memoria distribuita. Partendo dal presupposto che la memoria locale di un SPE non è abbastanza grande per contenere il codice e i dati di un'applicazione parallela, gli autori propongono un modello che consente di decomporre la computazione in una serie di microtask che possano essere lanciati in maniera indipendente sugli SPE. Il compito del programmatore consiste nel suddividere l'applicazione in microtask; il supporto

del sistema individuerà le dipendenze tra i microtask e schedulerà nel modo appropriato la loro esecuzione sugli SPE. Le comunicazioni tra microtask avvengono tramite i comunicatori di MPI.

Capitolo 2

Strumenti di programmazione: la libreria MammuT

Nel capitolo precedente è stato introdotto il Software Development Kit per l'architettura Cell fornito dalla IBM. Il problema principale legato all'uso di tale strumento sta nel fatto che, nonostante il Cell SDK costituisca un primo sottile strato di astrazione sopra il livello assembler, esso lascia ancora al programmatore l'onere della gestione di tutta una serie di problematiche legate a trasferimenti di dati e sincronizzazione tra processi che, oltre ad essere operazioni prone ad errori, complicano parecchio la fase di programmazione e fanno che si che i tempi di sviluppo di un'applicazione si allunghino. É necessario quindi un livello di astrazione che consenta di nascondere i dettagli dell'architettura, semplificando il lavoro del programmatore che potrà quindi concentrarsi maggiormente sull'applicazione da sviluppare. L'abilità di nascondere i dettagli di basso livello dell'implementazione, unita alla disponibilità di un modello dei costi statico e a una maggiore espressività sono i punti di forza di un approccio parallelo strutturato.

In questo capitolo verrà innanzitutto presentato il linguaggio concorrente a scambio di messaggi LC, un formalismo per la descrizione di programmi paralleli che introduce i concetti di canale di comunicazione, guardia e comando alternativo. Nel seguito sarà presentata la libreria MammuT, che

implementa i costrutti del linguaggio LC.

2.1 LC: linguaggio concorrente a scambio di messaggi

LC è un linguaggio concorrente a scambio di messaggi con modello di cooperazione ad ambiente locale. Esso nasce in ambito accademico come formalismo per la descrizione di programmi paralleli e si ispira al linguaggio ECSP, estensione del linguaggio CSP.

Un programma concorrente è una collezione di processi che comunicano tramite opportuni canali e può essere dichiarato come:

```
parallel <lista nomi unici di processi>;
         <eventuale dichiarazione di nomi parametrici>;

<definizione di processo>;
...
<definizione di processo>;
```

I processi vengono riferiti tramite i loro nomi, che sono unici in tutto il programma. La definizione di un processo ha una struttura del tipo:

```
<nome unico di processo>::
    <dichiarazioni>
    <codice>
```

I processi possono anche essere definiti parametricamente in funzione di una variabile libera. Ad esempio un array unidimensionale di processi è espresso come:

```
parallel P[N]; int A[N];
Proc[j]::
    ...
    int A[j];
    ...
    A[j] = F(..., A[j], ...);
    ...
```

Le componenti dell'array `A` sono partizionate tra i processi `P[0]`, `...`, `P[N-1]`, ed ognuno di essi dichiara ed usa, come variabile locale, l'elemento di `A` la cui posizione all'interno dell'array corrisponde al proprio indice. La dichiarazione di `A` nella parte generale ha il significato di dichiarare il nome della variabile che può essere utilizzata parametricamente nei processi del programma parallelo.

La parte sequenziale del linguaggio concorrente è quella di un linguaggio imperativo standard.

2.1.1 Canali di comunicazione

I processi che costituiscono il programma parallelo cooperano tramite lo scambio di messaggi inviati attraverso i canali di comunicazione. Nel linguaggio LC i canali sono tipati: il tipo dei canali corrisponde a quello dei messaggi che possono esservi inviati e delle variabili targa cui tali messaggi sono assegnati. A tempo di compilazione può quindi essere controllata la coincidenza dei tipi dei messaggi e delle variabili targa.

Come già detto, il modello di cooperazione di questo linguaggio è ad ambiente locale quindi i canali non sono oggetti condivisi (solo la conoscenza dei nomi è a comune tra processi partner). Il nome di un canale è espresso da una qualunque stringa di caratteri, che il supporto codificherà come un intero, ed è unico in tutto il programma espresso da parallel.

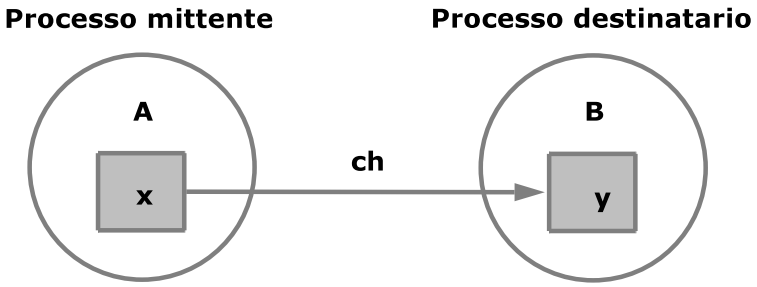

Figura 2.1: I due processi A e B cooperano a scambio di messaggi

La sintassi dei comandi per inviare e ricevere messaggi è:

```
send(nome canale, valore messaggio)

receive(nome canale, variabile targa)
```

Il modello di comunicazione è con porte unidirezionali con identificatore unico: il nome di un canale è unico in tutto il programma parallelo ed è dichiarato nei processi che lo usano, distinguendo se si tratta di canali in ingresso o in uscita. Facendo riferimento alla figura 2.1 il codice LC del programma parallelo è del tipo:

```
parallel A, B;
A::
    ...
    channel out ch;
    ...
    send(ch, x);
    ...
B::
    ...
    channel in ch;
    ...
    receive(ch, y);
    ...
```

Il nome di un canale, oltre ad essere una costante, può anche essere una variabile: il linguaggio LC prevede infatti un tipo apposito chiamato *channelname*. Per dichiarare una variabile di tipo *channelname*, si utilizza la parola chiave `var` nella dichiarazione del canale. I possibili nomi di canale che possono essere assunti da una variabile *channelname* sono determinabili a tempo di compilazione da un'analisi statica del programma parallelo. Sul tipo *channelname* è definita l'operazione di assegnamento in modo da poter gestire parametricamente i canali, usando nomi di canali in messaggi e variabili targa.

Il seguente esempio mostra un programma parallelo composto da quattro processi. I processi `A`, `B`, `C` operano da "client" nei confronti del processo "server" `D`, inviando sul canale `chD` un messaggio espresso come tupla contenente la richiesta di servizio e il nome del canale su cui desiderano ricevere la risposta al servizio. La variabile `chOut` del processo `D` assumerà, ad ogni `receive`, il valore dell'identificatore del canale su cui dovrà essere inviata la risposta.

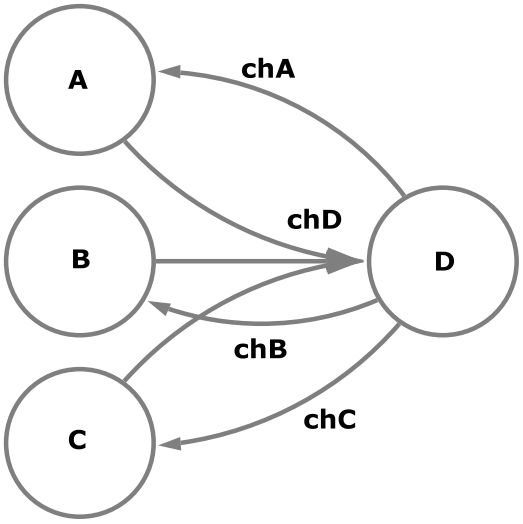

Figura 2.2: Esempio di utilizzo delle variabili channelname

```
parallel A, B, C, D;
A::
    channel in chA, ...;
    channel out chD, ...;
    ...
    send(chD, (chA, msgA));
    ...
    receive(chA, resultA);
    ...
B::
    channel in chB, ...;
    channel out chD, ...;
    ...
    send(chD, (chB, msgB));
    ...
    receive(chB, resultB);
    ...
C::
    channel in chC, ...;
    channel out chD, ...;
    ...
    send(chD, (chC, msgC));
    ...
    receive(chC, resultC);
    ...
D::
    channel in chD, ...;
    channel out var chOut, ...;
    ...
    receive(chD, (chOut, msg);
    ...
    result = F(msg);
    send(chOut, result);
    ...
```

Il tipo elementare *channelname* può essere usato in strutture di canali; ad esempio, è possibile dichiarare un array di m canali *ch[m]*. In tal caso, la

dichiarazione della struttura avviene anche nella parte generale del costrutto `parallel`, oltre che nei processi che utilizzano i canali della struttura. Il seguente esempio mostra come, usando le variabili di tipo *channelname* e/o i canali in tipi strutturati, un processo possa scegliere parametricamente a programma sia i canali di ingresso (in questo caso, si tratta di una semplice scansione di `N` canali simmetrici) che quelli di uscita:

```
parallel CLIENT[N], SERVER; channel chRequest[N];
CLIENT[j]::
    channel in chResult, ...;
    channel out chRequest[j], ...;
    ...
    send(chRequest[j], (chResult, msg));
    ...
    receive(chResult, resultMsg);
    ...
SERVER::
    channel in chRequest[N], ...;
    channel out var chOut;
    ...
    for j=1 to N do {
        receive(chRequest[j], (chOut, msg));
        result = F(msg);
        send(chOut, result);
    }
    ...
```

2.1.2 Forme di comunicazione

Le forme di comunicazione possibili nel linguaggio LC sono:

- *simmetrica* o *asimmetrica in ingresso*, che equivale a parlare di comunicazioni *1:1* o *N:1*; a programma i due casi non devono essere distinti da specifiche dichiarazioni in quanto i possibili mittenti di uno stesso canale asimmetrico saranno identificati in fase di compilazione;

- *sincrona* o *asincrona*: sia k il grado di asincronia di un canale, con $k \geq 0$, dove $k=0$ rappresenta il caso di comunicazione sincrona, k deve essere una costante e rappresenta il numero di messaggi che il mittente può inviare senza attendere che il destinatario abbia effettuato almeno una `receive`; la specifica del grado di asincronia di un canale viene associata alla dichiarazione del nome del canale ed è sufficiente che essa avvenga nel solo processo destinatario; nel caso $k=0$ la specifica può essere omessa.

Nel seguente esempio:

```
channel in ch1(4), ch2(1), ch3, ch4(0); channel out ...
```

i canali di nome `ch1`, `ch2`, `ch3`, `ch4` hanno grado di asincronia uguale rispettivamente a 4, 1, 0, 0. Nel caso di canale asimmetrico asincrono con grado di asincronia k, ogni mittente ha grado di asincronia k nei confronti del destinatario.

2.1.3 Controllo del nondeterminismo

Il controllo del nondeterminismo nelle comunicazioni viene espresso mediante comandi alternativi con guardia. Un comando alternativo consente di esprimere la possibilità di ricevere un messaggio da un qualsiasi canale all'interno di un determinato insieme variabile e controllabile a programma, eventualmente pilotando la scelta con delle priorità. La sintassi di un comando alternativo è:

```
alternative {
    priority(p_1), pred_1(...), receive(...) do {
        ... lista comandi LC1 ...
    } or priority(p_2), pred_2(...), receive(...) do {
        ... lista comandi LC2 ...
    } or priority(p_3), pred_3(...), receive(...) do {
        ... lista comandi LC3 ...
```

```
    }
}
```

in cui l'i-esima guardia

```
priority(p_i), pred_i(...), receive(...)
```

è formata da:

- una eventuale *priorità* indicata dal valore intero `p_i`, che se omessa indica che la guardia è a priorità minima;
- una eventuale *guardia locale* rappresentata dalla valutazione del predicato `pred_i` sullo stato interno presente del processo;
- una eventuale *guardia globale* espressa dalla primitiva di comunicazione `receive`.

Ogni guardia deve contenere almeno uno di questi tre elementi.

Una guardia può trovarsi in uno dei seguenti stati:

- *verificata*, se `pred_i` è vero e la corrispondente `receive` può essere eseguita;
- *sospesa*, se `pred_i` è vero ma la corrispondente `receive` non può essere eseguita;
- *fallita*, se `pred_i` è falso.

Se una o più guardie sono verificate, viene scelta quella a priorità maggiore e vengono eseguite la primitiva di comunicazione e la lista di comandi associata; a questo punto il comando alternativo termina. Quindi, la priorità è uno strumento che permette al programmatore di pilotare la scelta nel caso in cui sia possibile la comunicazione su più di uno dei canali specificati. La scelta nel caso di priorità uguali è dettata da una politica non nota al programmatore.

Se tutte le guardie sono sospese, il processo si sospende sul comando alternativo e verrà risvegliato non appena una o più guardie diverranno verificate.

Se tutte le guardie sono fallite, il comando alternativo termina senza eseguire alcuna `receive` nè liste di comandi LC.

Tramite un comando alternativo si è in grado di esprimere la semantica di una `receive` da canale asimmetrico. Facendo riferimento all'esempio in figura 2.2, si potrebbe sostituire il canale asimmetrico `chD` con 3 canali simmetrici (`ch1`, `ch2`, `ch3`) gestiti sul processo `D` con un comando alternativo, nel seguente modo:

```
alternative {
    receive(ch1, (chOut, msg)) do {...}
    or receive(ch2, (chOut, msg)) do {...}
    or receive(ch3, (chOut, msg)) do {...}
}
```

2.2 MammuT

MammuT (cell Multi-core Architecture coMMUnication supporT) è una libreria di classi per il linguaggio C++ realizzata presso il Laboratorio di Architetture Parallele del Dipartimento di Informatica che implementa il supporto del formalismo presentato nelle sezioni precedenti. La libreria non supporta tutte le astrazioni offerte dal linguaggio LC. Durante il lavoro di tesi ci si è accorti che alcune di queste, come gli array di canali e di processi, sarebbero tornate molto utili per l'implementazione del livello ASSIST, e si è quindi deciso di introdurle. Inoltre è stato implementato un meccanismo per descrivere in forma più dichiarativa il grafo dell'applicazione; questo ha consentito di automatizzare alcune parti della fase di setup dell'applicazione, semplificando il lavoro del programmatore, utente della libreria. Le variabili

channelname, invece, continuano a non essere presenti nella libreria.

A livello di codice la libreria è divisa in due parti distinte: PPE e SPE. Per ogni costrutto esistono due implementazioni, ottimizzate per i due diversi tipi di core, pur esponendo un'interfaccia uguale.

L'esecuzione di un programma parallelo sull'architettura Cell parte con l'esecuzione di un processo allocato sul PPE, che si occupa della creazione e del lancio di tutti gli altri processi che compongono il programma parallelo. Il sistema operativo non offre supporto alla multiprogrammazione degli SPE che, a causa della limitatezza della memoria indirizzabile (solo 256 KB) si dimostrano comunque poco adatti. I processi dell'applicazione parallela sono quindi assegnati staticamente agli SPE, su cui non avverrà alcuna commutazione di contesto.

In linea di massima il processo sul PPE seguirà uno schema come il seguente:

- inizializzazione parziale delle strutture dati del supporto;
- lancio dell'applicazione;
- eventuale codice del processo.

Allo stesso modo un generico processo lanciato su uno degli SPE avrà uno schema come il seguente:

- inizializzazione delle strutture dati del supporto;
- registrazione dei canali di comunicazione;
- codice del processo.

In questa sezione saranno analizzati i costrutti principali della libreria MammuT: si inizierà con la descrizione delle operazioni di setup dell'applicazione, per poi proseguire con i canali di comunicazione e concludere con la presentazione di guardie e comando alternativo. Nella sezione successiva

saranno introdotti i costrutti implementati nell'ambito di questo lavoro di tesi come estensione di MammuT: array di processi, array di canali e grafo dell'applicazione.

2.2.1 Stcom e inizializzazione della libreria

Il cuore della libreria è costituito da un oggetto comunicatore che nasconde al programmatore tutte le strutture dati interne del supporto, quali ad esempio la tabella dei canali e quella dei processi. Il comunicatore è implementato tramite la classe `Stcom`, che gestisce tutte le operazioni di inizializzazione della libreria. Come detto precedentemente, per ogni costrutto della libreria esistono due implementazioni ottimizzate per i due diversi tipi di core, PPE e SPE. La classe `Stcom` non fa eccezione e ogni processo del programma concorrente deve costruire un oggetto di tipo `Stcom`, utilizzando la versione corretta.

Stcom su PPE

Un processo allocato sul PPE utilizza il comunicatore per gestire tre tipi di operazioni:

- registrazione dei processi;
- registrazione dei canali di comunicazione;
- lancio dei processi concorrenti.

La procedura di registrazione dei processi permette di associare ad ogni processo del programma concorrente il corrispondente codice da eseguire.

Per ogni processo che si vuole registrare si dovrà chiamare il metodo `proc_register` della classe `Stcom`. Esistono due versioni di questo metodo ed entrambe hanno due parametri. Come primo parametro, la prima versione prende il path del file eseguibile associato al processo mentre la seconda versione prende l'handle del binario SPE associato al processo. Come secondo

parametro entrambe le versioni prendono l'id da assegnare al processo. Le prima versione va utilizzata quando il codice per il processo risiede su un file separato, mentre la seconda versione deve essere utilizzata quando il binario SPE è incluso nel binario PPE (vedi sezione 1.6.1).

Le signature dei due metodi sono le seguenti:

```
int Stcom::proc_register(char *path, proc_id_t procid);

int Stcom::proc_register(spe_program_handle_t *phandle,
                         proc_id_t procid);
```

dove il tipo `proc_id_it` è definito come segue:

```
typedef int proc_id_t;
```

La procedura di registrazione dei canali viene utilizzata per definire l'insieme dei canali di comunicazione che sono utilizzati nel programma concorrente. Oltre ad inizializzare le strutture dati interne alla libreria, come la tabella dei canali, questa procedura si occupa di una prima parziale inizializzazione dei canali di comunicazione, che verrà poi completata dai processi interessati. Analogamente a quanto avviene nella registrazione dei processi, ad ogni canale corrisponderà una chiamata al metodo `ch_register` della classe `Stcom`, che prende come parametri il puntatore al canale da registrare e un identificatore univoco da assegnare al canale. Il metodo ha quindi la seguente signature:

```
void Stcom::ch_register(Channel_ppe *ch, ch_id_t chid);
```

dove il tipo `ch_id_it` è definito come segue:

```
typedef int ch_id_t;
```

Nel caso in cui la registrazione non sia effettuata correttamente, ad esempio se lo stesso identificatore viene utilizzato per due registrazioni distinte, la

libreria solleverà degli errori. Più avanti si vedrà che l'introduzione del grafo dell'applicazione consentirà di gestire automaticamente tutti gli id e gran parte delle registrazioni.

Il lancio dei processi concorrenti avviene attraverso la chiamata del metodo `start`, sempre della classe `Stcom`, che si occupa anche di avviare la procedura di inizializzazione della libreria. La signature del metodo è la seguente:

```
void Stcom::start();
```

Stcom su SPE

Come detto precedentemente, anche un processo allocato su uno degli SPE deve creare un oggetto di tipo `Stcom` ma, rispetto al processo eseguito sul PPE, non ci sono processi da registrare o lanciare: l'unica operazione per cui un processo SPE ha bisogno di un oggetto `Stcom` è la registrazione dei canali di comunicazione. Ogni processo SPE deve creare e registrare tutti e solo i canali di cui ha bisogno. Per ogni canale da registrare lato SPE, un oggetto canale del tipo corrispondente deve essere stato creato e registrato nel processo PPE, e il processo SPE deve essere stato indicato come uno dei partner della comunicazione. La registrazione di un canale si effettua per mezzo del metodo `ch_register` della classe `Stcom`, che lato SPE ha una signature differente rispetto a quella vista sul PPE:

```
template <class T_channel>
void Stcom::ch_register(T_channel *ch, ch_id_t chid,
                        int sender_or_receiver)
```

Il parametro `ch` è il puntatore al canale di comunicazione da registrare, il cui tipo deve corrispondere con quello utilizzato sul processo PPE. Il secondo parametro è l'identificatore unico del canale, che deve corrispondere con quello assegnato nel processo PPE. Il terzo e ultimo parametro è un flag che

può assumere valori `SENDER` o `RECEIVER` a seconda che il processo SPE sia, rispettivamente, mittente o destinatario del canale.

Inizializzazione libreria

É sicuramente interessante vedere, anche a grandi linee, i passi che compongono la procedura di inizializzazione della libreria. Essa viene lanciata dal metodo `start` della classe `Stcom`, nella sua implementazione per il processo PPE, ed è divisa in più fasi:

1. Il processo sul PPE inizializza la tabella dei processi, impostando i valori iniziali dei campi di tutti i descrittori di processo e crea i contesti SPE necessari per lanciare i processi che compongono l'applicazione concorrente, caricando il codice eseguibile dei processi nei relativi contesti.

2. Il processo sul PPE alloca la tabella dei canali e, per ogni canale di comunicazione utilizzato nell'applicazione, inizializza la entry corrispondente. L'inizializzazione dei canali di comunicazione, in questa fase, consiste nell'inizializzazione parziale dei *descrittori di endpoint* relativi al canale stesso.

3. Il processo sul PPE lancia i contesti SPE e comunica via *mailbox* l'indirizzo effettivo [1] in memoria principale del *control block*, una struttura dati contenente informazioni necessarie all'inizializzazione del supporto sui processi allocati sugli SPE, quali ad esempio indirizzo effettivo e dimensione della tabella dei canali e il proprio identificatore unico di processo.

4. Il processo sul PPE effettua una sincronizzazione a barriera con i processi SPE, implementata con le *mailbox*, attendendo una loro segnalazione.

[1] EA, Effective Address

5. I processi SPE leggono da memoria principale il *control block* e creano una copia della tabella dei canali nel proprio *local storage*. Dalla tabella dei canali sono ricavate varie informazioni, tra cui il numero di canali di comunicazione di cui il processo è un estremo.

6. I processi SPE leggono i *descrittori di endpoint* relativi ai canali cui partecipano; ogni processo legge e inizializza i propri descrittori, ma non quelli relativi ai partner della comunicazione. L'inserimento delle proprie informazioni nei *descrittori di endpoint* avviene durante la procedura di registrazione dei canali che, come visto in precedenza, corrisponde a una serie di chiamate al metodo `ch_register` della classe `Stcom` nella versione per i processi SPE.

7. L'ultima chiamata al metodo `ch_register` invia una segnalazione al processo sul PPE e attende da questo una segnalazione di risposta ad indicare la fine della sincronizzazione a barriera.

8. Quando tutti i processi allocati sugli SPE hanno inviato la segnalazione, il processo sul PPE invia loro la segnalazione di risposta.

9. Una volta ricevuta la segnalazione di "via libera" dal processo sul PPE, i processi SPE leggono i *descrittori di endpoint* dei loro partner di comunicazione, che a quel punto saranno completamente inizializzati.

2.2.2 Canali di comunicazione

I processi di un programma parallelo comunicano tramite opportuni canali. La libreria MammuT fornisce i canali per coprire tutte le possibili tipologie di comunicazioni tra le diverse unità all'interno del Cell: PPE - SPE, SPE - PPE, SPE - SPE.

Sono presenti tre diverse implementazioni per i canali di comunicazione SPE - SPE mentre per comunicazioni tra PPE e SPE l'implementazione fornita è una sola.

Lo schema di base è comune a tutte le implementazioni. Per evitare copie superflue del messaggio, viene preallocato un buffer di ricezione sul destinatario; dato che LC supporta solo messaggi tipati la dimensione di ciascun buffer è calcolabile a tempo di compilazione. Lato mittente vengono memorizzati un insieme di riferimenti a variabile targa, dove con riferimento si intende un indirizzo utilizzato per scrivere nella posizione corretta del buffer del ricevente. Ad ogni elemento del buffer sul ricevente e ad ogni riferimento a variabile targa sul mittente è associato un bit di presenza; tali bit sono manipolati tramite le due primitive `set_bit` e `wait_bit`. La prima consente di settare un bit di presenza remoto e può essere usata sia dal mittente per impostare il bit associato a un elemento del buffer, sia dal ricevente per impostare il bit associato a un riferimento. La seconda consente di testare il valore di un bit locale in attesa che venga settato dal partner della comunicazione. Il significato dei bit di presenza varia tra mittente e destinatario: lato mittente un certo bit associato a un riferimento indica se quel particolare riferimento può essere utilizzato per scrivere un messaggio al ricevente; lato ricevente un certo bit associato a un elemento del buffer indica se il mittente ha scritto un messaggio in quella particolare posizione del buffer. Per effettuare una `send` quindi, il mittente deve innanzitutto aspettare che venga settato il bit di presenza associato a un riferimento, quindi scrive il messaggio sul buffer del ricevente, setta il relativo bit di presenza remoto e azzera il bit locale corrispondente al riferimento della variabile targa appena scritta. La `receive` è molto più semplice: basta attendere che il bit di presenza associato all'elemento del buffer corrente venga settato dal mittente; quando questo avviene si è sicuri che il mittente ha scritto il messaggio nel buffer.

Gli elementi del buffer vengono usati in modo circolare, e la dimensione del buffer è data dal grado di asincronia del canale: un grado di asincronia

k fa sì che venga dichiarato un buffer con $k + 1$ posizioni. Questo consente di introdurre un'ottimizzazione per cui il mittente per prima cosa scrive il messaggio nel buffer e poi controlla il bit relativo a quella entry.

La `receive` restituisce al programma il puntatore all'elemento del buffer che contiene il messaggio ricevuto. Tale puntatore è valido fino alla `receive` successiva; se l'applicazione dovesse ancora avere bisogno di quel particolare valore dovrebbe copiarlo in un'altra locazione di memoria.

Per evitare di dover copiare il messaggio ricevuto prima di effettuare un'altra `receive`, è stato previsto un meccanismo che consente di sovradimensionare il buffer del ricevente, tramite la dichiarazione di una finestra costituita da un certo numero di posizioni aggiuntive nel buffer. La dimensione della finestra rappresenta la soglia temporale di validità di un messaggio ricevuto: una finestra lunga w significa che un determinato messaggio ricevuto non sarà più valido dopo aver effettuato w `receive`. Un canale con grado di asincronia k e dimensione della finestra aggiuntiva pari a w, considerando anche l'ottimizzazione che consente di testare il bit dopo aver scritto il messaggio in variabile targa, avrà quindi un buffer con $k + w + 1$ posizioni.

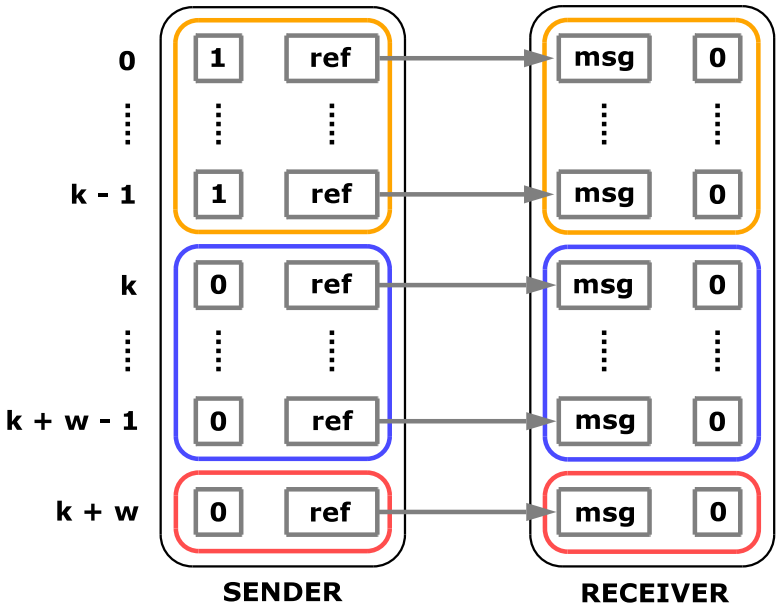

Figura 2.3: Strutture dati per l'implementazione dei canali

Implementazione canali SPE - SPE

La libreria fornisce tre diverse implementazioni per canali di comunicazione tra SPE, che fanno uso di meccanismi di basso livello differenti.

La prima versione implementa i bit di presenza associati a variabili targa e riferimenti utilizzando i registri hardware dei segnali presenti sul Cell. L'invio dei dati avviene tramite un'operazione di DMA mentre la primitiva `set_bit` è implementata con l'invio di un segnale al SPE remoto. La latenza di una send è composta quindi dalla latenza del trasferimento DMA e dalla latenza dell'invio del segnale. Questa versione del canale richiede un numero di bit di segnale, sia lato mittente che lato ricevente, pari al numero di entry presenti nel buffer.

Un problema dell'implementazione basata su segnali riguarda l'overhead dovuto al trasferimento aggiuntivo sulla struttura di interconnessione per l'invio della segnalazione al partner. Per evitare questo ritardo si è pensato di unire dati e sincronizzazione in un unico messaggio: si considera ogni messaggio come una struttura contenente il messaggio originale più un byte che ha la funzione di bit di stato. Per attendere l'arrivo del messaggio, il processo destinatario non deve più fare polling sul registro dei segnali ma nel suo local storage, sul byte di sincronizzazione all'interno della prossima posizione del buffer. Le due implementazioni di canali non ancora presentate si basano su queste osservazioni. Una di queste, chiamata DMA1, funziona solo per messaggi [2] con dimensioni fino a 128 byte. La latenza della send equivale alla latenza del trasferimento DMA. Il numero di bit di segnale richiesti è zero sul ricevente mentre sul mittente è pari al numero di entry presenti nel buffer.

L'ultima versione di canale implementata si chiama DMA2: come la DMA1 è basata unicamente su operazioni di DMA ma può trasferire anche messaggi più lunghi di 128 byte. L'invio del messaggio viene diviso in due blocchi spediti separatamente: la prima operazione in DMA trasferisce il più grande blocco del messaggio multiplo di 128 byte; la seconda opera-

[2] In questo contesto, il messaggio è inteso come dati più byte di sincronizzazione.

zione in DMA invece trasferisce 128 byte, che comprendono l'ultima parte del messaggio e il byte di sincronizzazione. Nel caso in cui la dimensione del messaggio sia un multiplo di 128 byte, la prima operazione in DMA trasferirà tutto il messaggio e la seconda trasferirà 128 byte con il solo byte di sincronizzazione come dato utile. La latenza della send equivale alla latenza di due trasferimenti DMA e, analogamente alla versione DMA1, i bit di segnale vengono usati solo sul mittente in numero pari alle entry del buffer di ricezione.

Implementazione canali PPE - SPE

I canali per le comunicazioni tra PPE e SPE sono molto simili alla versione dei canali SPE - SPE basata sull'uso dei segnali. L'invio dei dati avviene sempre con un'operazione di trasferimento in DMA mentre l'implementazione della primitiva `set_bit` varia in base alla direzione della comunicazione. Le segnalazioni dal PPE a uno degli SPE utilizzano i segnali mentre quelle nella direzione opposta utilizzano le mailbox perchè il PPE, al contrario degli SPE, non ha un registro per i segnali. Ad ogni canale di comunicazione allocato sul processo PPE è associato un semaforo il cui valore indica il numero di messaggi presenti in coda, nel caso in cui il processo PPE sia il destinatario del canale, e il numero di variabili targa disponibili per la scrittura, nel caso in cui il processo PPE sia il mittente del canale. Il test sulla presenza di messaggi, nel caso del destinatario, o sulla disponibilità di variabili targa nel caso del mittente, è implementato con una primitiva `P` sul semaforo associato al canale. Il partner della comunicazione, per segnalare l'avvenuta ricezione o l'avvenuto invio di un messaggio, utilizza il meccanismo delle mailbox per inviare il corrispondente identificatore di canale. Ad ogni processo allocato sugli SPE è associato un thread sul PPE, chiamato *listener*, che attende messaggi sulla mailbox di uscita del SPE corrispondente, effettuando una `V` sul semaforo associato all'identificatore di canale ricevuto.

Interfacce per l'utilizzo dei canali

L'interfaccia per la dichiarazione di un canale è costituita da due classi template, una da utilizzare sul codice dei processi PPE e l'altra sul codice dei processi SPE.

La classe da utilizzare sul PPE è la seguente:

```
template <endPointRole t_role, class T_item,
          int t_degree = 1, int t_nKeep = 0>
class Ppe_channel { ... };
```

I parametri di template presenti sono quattro. Gli ultimi tre sono abbastanza intuitivi: `T_item` rappresenta il tipo degli oggetti trasferiti sul canale, `t_degree` è il grado di asincronia e `t_nKeep` è la dimensione della finestra di entry aggiuntive sul buffer del ricevente di cui si è discusso in precedenza. Il parametro `t_role` rappresenta invece il ruolo del processo PPE nel canale; i valori che questo parametro può assumere sono tre e ad ognuno di essi corrisponde una diversa specializzazione del template:

- `NONE` indica che il canale è di tipo SPE - SPE quindi l'unico compito del processo PPE consiste nell'inizializzazione delle strutture dati fondamentali, come la entry nella tabella dei canali e i *descrittori di endpoint*;

  ```
  template <class T_item, int t_degree, int t_nKeep>
  class Ppe_channel<NONE, T_item, t_degree, t_nKeep> :
      public SPEChannelSkel<T_item, t_degree, t_nKeep>
  { ... };
  ```

- `SENDER` indica che il canale è di tipo PPE - SPE e il processo PPE è il mittente;

```
template <class T_item, int t_degree, int t_nKeep>
class Ppe_channel<SENDER, T_item, t_degree, t_nKeep> :
    public SR_ppech<T_item, t_degree, t_nKeep>
{ ... };
```

- `RECEIVER` indica che il canale è di tipo SPE - PPE e il processo PPE è il ricevente.

```
template <class T_item, int t_degree, int t_nKeep>
class Ppe_channel<RECEIVER, T_item, t_degree, t_nKeep> :
    public RR_ppech<T_item, t_degree, t_nKeep>
{ ... };
```

Nel codice dei processi SPE la classe da utilizzare per dichiarare un canale è la seguente:

```
template <endPointRole t_endPoint, class T_item,
          int t_degree = 1, int t_nKeep = 0,
          imp_type_t t_impl = IMP_AUTO>
class Spe_channel { };
```

Il primo parametro di template, `t_endPoint`, indica il ruolo del processo nel canale e i valori che può assumere sono due: `SENDER` o `RECEIVER`. I successivi tre parametri sono anologhi a quelli visti nella classe lato PPE: `T_item` rappresenta il tipo degli oggetti trasferiti sul canale, `t_degree` è il grado di asincronia e `t_nKeep` è la dimensione della finestra sul buffer del ricevente. L'ultimo parametro, `t_impl`, indica il tipo di implementazione desiderata per il canale; i valori che questo parametro può assumere sono cinque e ad ognuno di essi corrispondono due diverse specializzazioni del template, una per il caso `t_endpoint = SENDER` e l'altra per il caso del `t_endpoint = RECEIVER`:

- `IMP_WITH_SIGNAL` consente di scegliere l'implementazione basata sull'uso dei segnali per le sincronizzazioni;

```
template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<SENDER, T_item, t_degree, t_nKeep,
                  IMP_WITH_SIGNAL> :
    public SRChannelSignal<T_item, t_degree, t_nKeep>
{ ... }

template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<RECEIVER, T_item, t_degree, t_nKeep,
                  IMP_WITH_SIGNAL> :
    public RRChannelSignal<T_item, t_degree, t_nKeep>
{ ... }
```

- `IMP_WITH_DMA` consente di scegliere l'implementazione chiamata DMA1, che fa uso di una singola operazione di DMA;

```
template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<SENDER, T_item, t_degree, t_nKeep,
                  IMP_WITH_DMA> :
    public SRChannelDMA<T_item, t_degree, t_nKeep>
{ ... }

template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<RECEIVER, T_item, t_degree, t_nKeep,
                  IMP_WITH_DMA> :
    public RRChannelDMA<T_item, t_degree, t_nKeep>
{ ... }
```

- `IMP_WITH_TWO_DMA` consente di scegliere l'implementazione chiamata DMA2, che utilizza due operazioni di DMA;

```
template <class T_item, int t_degree, int t_nKeep>
```

```
class Spe_channel<SENDER, T_item, t_degree, t_nKeep,
                  IMP_WITH_TWO_DMA> :
    public SRChannelDMABuffer <T_item, t_degree, t_nKeep>
{ ... }

template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<RECEIVER, T_item, t_degree, t_nKeep,
                  IMP_WITH_TWO_DMA> :
    public RRChannelDMABuffer <T_item, t_degree, t_nKeep>
{ ... }
```

- IMP_AUTO lascia al supporto la scelta dell'implementazione migliore in base alla dimensione del messaggio da spedire; per messaggi con dimensione inferiore a 128 byte la soluzione DMA1 è quella che ottiene la latenza più bassa, grazie alla singola connessione necessaria sull'*Element Interconnection Bus*, mentre per messaggi più lunghi la latenza migliore si raggiunge con la versione DMA2 dei canali;

```
template <endPointRole t_endPoint, class T_item,
          int t_degree, int t_nKeep>
class Spe_channel<t_endPoint, T_item, t_degree, t_nKeep,
                  IMP_AUTO> :
    public Spe_channel<t_endPoint, T_item, t_degree, t_nKeep,
                  (imp_type_t) (sizeof(T_item) > (128 - 4) ?
                                IMP_WITH_TWO_DMA :
                                IMP_WITH_DMA) >
{ ... }
```

- IMP_PPECH consente di scegliere un canale per comunicazioni col PPE.

```
template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<SENDER, T_item, t_degree, t_nKeep,
                  IMP_PPECH> :
    public SRChannelPPE<T_item, t_degree, t_nKeep>
```

```
{ ... }

template <class T_item, int t_degree, int t_nKeep>
class Spe_channel<RECEIVER, T_item, t_degree, t_nKeep,
                  IMP_PPECH> :
    public RRChannelPPE<T_item, t_degree, t_nKeep>
{ ... }
```

I costruttori delle varie specializzazioni del template hanno un parametro `n_item` che indica il numero di elementi di tipo `T_item` che compongono i messaggi da trasferire sul canale. Il valore `n_item` è una caratteristica costante del canale perchè viene impostato all'atto della dichiarazione del canale e non può più essere modificato. Questo parametro pone dei limiti nel caso in cui si decida di far scegliere al supporto l'implementazione di un canale SPE - SPE in quanto la dimensione del messaggio non dipende solo dal tipo degli *item* ma anche dal loro numero. Per questo motivo il canale con implementazione `IMP_AUTO` non consente al programmatore di scegliere il numero di *item* che compongono il messaggio.

Creazione dei canali

La creazione di un canale di comunicazione SPE - SPE consiste nella creazione di tre oggetti canali:

- nel processo PPE;
- nel processo mittente;
- nel processo destinatario.

e gli oggetti creati devono avere dei tipi coerenti [3]. Il processo sul PPE deve

[3] La coerenza riguarda sia i parametri di template usati nella definizione degli oggetti canali che il parametro `n_item` del costruttore.

- creare il canale indicando che il suo ruolo è `NONE`;
- registrare gli estremi della comunicazione utilizzando i metodi seguenti

```
virtual int Channel_ppe::add_sr(proc_id_t id);

virtual int Channel_ppe::add_rr(proc_id_t id);
```

  indicando come parametro l'identificatore del processo interessato;

- registrare il canale utilizzando il metodo `ch_register` della classe `Stcom`.

I processi mittente e destinatario devono creare il proprio oggetto canale utilizzando l'interfaccia per i processi SPE precedentemente descritta; il canale deve poi essere registrato utilizzando il metodo `ch_register` della classe `Stcom`.

La creazione di un canale di comunicazione PPE - SPE consiste nella creazione di due oggetti canali:

- nel processo PPE;
- nel processo partner.

Le operazioni da svolgere nei processi PPE e SPE sono identiche al caso precedente. a parte il fatto che il PPE dovrà dichiarare un ruolo di `SENDER` o `RECEIVER`. L'identificatore unico del processo PPE è definito dalla costante `Pids::PPE_ID`.

Primitive fornite dai canali

Le principali primitive fornite dai canali di comunicazione sono la `send` per il processo mittente e la `receive` per il processo destinatario. I due metodi hanno la seguente signature:

```
int send(T_item *msg);

T_item *receive();
```

Il metodo `send` prende come parametro il puntatore al messaggio da spedire e restituisce l'identificatore del trasferimento. Per allocare le aree di memoria che conterranno i messaggi da inviare è presente un metodo apposito, con la seguente signature:

```
T_item *allocMsg(alignment_type_t alignment = ALIGN_128);
```

Il tipo `alignment_type_t` è definito come segue:

```
enum alignment_type_t { ALIGN_16, ALIGN_128 };
```

e indica l'allineamento desiderato per l'area di memoria da allocare; le alternative possibili sono due: 16 byte o 128 byte. L'allineamento a 128 byte è più dispendioso in termini di memoria, ma offre prestazioni migliori.

Il metodo `receive` restituisce il puntatore alla entry del buffer di ricezione che contiene il messaggio. Il puntatore ricevuto è valido fino all'esecuzione della receive successiva, a meno che il buffer non comprenda una finestra con dimensione maggiore di 1.

La primitiva `send` non attende il trasferimento del messaggio; l'area di memoria contenente il messaggio inviato non può essere scritta immediatamente dopo la send. Per essere sicuri di poter scrivere su quella determinata area di memoria è necessario chiamare la primitiva `wait`. La presenza di questa primitiva permette di differire l'attesa della fine del trasferimento del messaggio, rendendo possibile la sovrapposizione della comunicazione al calcolo. La signature del metodo `wait` è la seguente:

```
void wait(T_item *msg, int tag);
```

dove `msg` indica l'indirizzo del'area di memoria contenente il messaggio di cui si vuole attendere la fine del trasferimento, mentre il parametro `tag` è l'intero restituito dalla `send` relativa al messaggio in questione.

2.2.3 Guardie e comando alternativo

Il nondeterminismo viene gestito tramite comandi alternativi con guardie. L'utilizzo del comando alternativo fornito dalla libreria prevede le stesse fasi sia per un processo PPE che per un processo SPE:

- definizione delle guardie;
- creazione del comando alternativo, creazione e registrazione delle guardie;
- esecuzione del comando alternativo.

Le implementazioni di guardie e comando alternativo sono diverse per processi PPE e SPE, ma l'interfaccia esposta è la stessa. L'interfaccia delle guardie è definita dalla classe astratta `Guard`, presentata nella sezione 2.4.1.

I metodi più importanti di una guardia sono senza dubbio tre:

- `evaluate`, che rappresenta la guardia locale, cioè il predicato sullo stato interno presente del processo; se questo metodo rende `false` la guardia risulta *fallita*;
- `slotAvailable`, che indica se è possibile effettuare una primitiva di comunicazione (`send` o `receive`) sul canale; se questo metodo rende `true` la guardia risulta *verificata* altrimenti risulta *sospesa*;
- `compute`, che contiene il codice da eseguire quando una guardia risulta *verificata* (comprende anche la primitiva di comunicazione sul canale).

La libreria mette a disposizione due diverse classi per rappresentare le guardie. La prima modella un tipo di guardia che gestisce un unico canale di comunicazione; è stata chiamata `GuardOne` ed è una classe template il cui unico parametro indica il tipo del canale di comunicazione gestito.

```
template <class T_ch>
class GuardOne : public Guard
{ ... };
```

Un'obiezione che potrebbe essere sollevata riguarda la reale utilità di avere una classe template col tipo del canale come parametro: si sarebbe potuto evitare sfruttando i meccanismi di ereditarietà e polimorfismo del C++, utilizzando un puntatore alla classe base dei canali (non template) per memorizzare l'oggetto canale. Questo però non è stato possibile perchè i metodi `send` e `receive` dei canali, per motivi di efficienza, non sono `virtual` e sono invocabili **unicamente** su un oggetto del tipo corretto del canale.

La seconda classe è stata chiamata `GuardMulti` e modella un tipo di guardia che gestisce più canali di comunicazione in AND tramite una composizione di oggetti `GuardOne`, ognuno dei quali si occupa di uno dei canali.

```
class GuardMulti : public Guard
{ ... };
```

Per personalizzare una guardia il programmatore deve creare una propria classe che erediti da `GuardOne` o `GuardMulti` ridefinendone il metodo `compute`.

Il comando alternativo è modellato con la classe `AltCmd`, che contiene una lista di guardie, aggiunte dal programmatore a tempo di esecuzione tramite il metodo `addGuard`. All'interno del comando alternativo le guardie vengono poi ordinate per priorità, e ad ognuna di esse viene assegnato un id univoco.

Il metodo che consente di eseguire il comando alternativo è chiamato `eval`: il suo compito è quello di scorrere le guardie ed eseguire il metodo `compute` della prima che risulta *verificata*. Il valore di ritorno del metodo è l'identificatore della guardia valutata; se tutte le guardie risultano *fallite* il metodo restituisce il valore -1.

2.3 Estensione di MammuT

Nella sezione precedente sono stati introdotti i costrutti principali della libreria MammuT. Questa sezione avrà il compito di presentare i costrutti introdotti come estensione di MammuT: gli array di processi e canali e il grafo dell'applicazione.

2.3.1 Array di processi e canali

Gli array di processi e di canali aumentano notevolmente l'espressività del linguaggio LC, consentendo la scrittura di programmi parametrici. Ad esempio, se di deve scrivere un farm o una map, è comodo specificare i worker come array di processi. Il processo che distribuisce gli input può utilizzare un array di canali della stessa cardinalità, in modo che il canale i-esimo abbia come destinatario il processo i-esimo dell'array. Anche l'implementazione di ASSIST su Cell si basa pesantemente sui costrutti parametrici.

Il punto di partenza per l'implementazione di questi costrutti è costituito dai metodi per la registrazione di processi (`Stcom::proc_register`) e canali (`Stcom::ch_register`). Entrambi i metodi richiedono come parametro l'id del processo o del canale che si vuole registrare, ma l'obiettivo è quello di risparmiare al programmatore l'onere di una gestione manuale degli id, prona ad errori.

Utilizzare un range di id consecutivi per un array di processi o un array di canali è molto vantaggioso, perchè la corrispondenza tra id e indice nell'array

è diretta. In un range di id `[startid, startid + N - 1]`, l'i-esimo id è `startid + i` (con `i < N`). Viceversa, se si vuole testare l'appartenenza di un id ad un certo range, basta verificare che `id - startid < N`. Il termine `id - startid` denota la posizione dell'id all'interno del range. Per definire un range bastano quindi due interi: il primo id (`startid`) e la dimensione (`N`).

Tramite i range di id, è possibile implementare sia gli array di processi, sia gli array di canali. L'unico dato necessario per far riferimento ad un processo è il suo id, perciò un array di processi coincide con un range di id di processi. La registrazione di un array di processi è possibile tramite la funzione

```
void registerProcArray(MammuT::Stcom *com,
                       const ProcIDRange &range,
                       spe_program_handle_t *handle)
```

Un array di canali è implementato come un array di puntatori a canali dello stesso tipo. In generale, un processo utilizza solo una parte dei canali di un array, dunque è necessario offrire un'interfaccia che consenta la registrazione selettiva dei canali. Il listato nella sezione 2.4.2 mostra la classe `ChannelArraySPE`, utilizzata per dichiarare un array di canali su un SPE.

`ChannelArraySPE` è una classe template il cui parametro permette di specificare il tipo dei canali appartenenti all'array. É presente un metodo `registerChannel` che permette di registrare il canale in posizione `index`, mentre un altro metodo, `registerAll`, registra tutti i canali. Quest'ultimo metodo è comodo quando il processo corrente partecipa a tutti i canali dell'array. E' importante sottolineare che nel caso di registrazione singola il programmatore non deve specificare l'id del canale da registrare, ma solo l'indice del canale rispetto all'array. La classe emula un vero array tramite l'operatore `[ ]`, e l'accesso tramite indice può essere effettuato anche con il metodo `at`. Infine il metodo `size` restituisce la dimensione dell'array.

Ai canali non utilizzati corrisponde un puntatore a `NULL`, ed è responsabilità del programmatore assicurarsi di accedere coerentemente ai soli canali

registrati dell'array.

2.3.2 Grafo dell'applicazione

L'implementazione degli array di processi e gli array di canali non è completa se non viene fornito un sistema semplice per rendere noto a tutti i processi la struttura dell'applicazione. Nel linguaggio LC, la visibilità dei nomi parametrici dichiarati nel costrutto `parallel` è globale, quindi occorre un metodo per riprodurre questa funzionalità nella libreria. Per ottenere questo risultato si è introdotta una classe chiamata `Graph`, presentata nella sezione 2.4.3.

Il costruttore ha un parametro di tipo intero previsto per un'estensione futura che possa allocare i processi dinamicamente a partire dal numero di processi disponibili; per il momento questo parametro non viene utilizzato. Tramite i due metodi `setProcIDRange` e `setChannelIDRange` è possibile inizializzare i range di id a una determinata dimensione. La semantica del metodo `setProcIDRange` è la seguente:

```
r = [m_nextProcID, m_nextProcID + num - 1];
m_nextProcID += num;
```

La variabile `m_nextProcID` è un membro di classe che contiene sempre il valore del primo id di processo libero. Lo stesso ragionamento vale per i range di id di canali con il metodo `setChannelIDRange` memorizzando il valore del primo id di canale libero nella variabile membro `m_nextChannelID`. A questo punto, per rendere noti i range a tutti i processi dell'applicazione è sufficiente dichiarare una nuova classe che eredita dalla classe `Graph`. La nuova classe funge da "contenitore" per i range di id corrispondenti ai nomi parametrici in LC, e il costruttore della classe può inizializzare i range secondo lo schema desiderato. Istanziando un oggetto di questa classe, ogni processo ha a

disposizione il "grafo" dell'applicazione. Ad esempio, consideriamo i seguenti nomi parametrici dichiarati in LC:

```
parallel p1[N], p2[K], channel ch1[N], channel ch2[K];
...
```

Il programmatore può facilmente riprodurre lo stesso schema scrivendo questa classe:

```
#pragma once
#include <common/assistgraph.h>

/* classe custom, specifica per l'applicazione */
class MyGraph : public MammuT::Graph
{
public:
    MammuT::ProcIDRange p1;
    MammuT::ProcIDRange p2;
    MammuT::ChannelIDRange ch1;
    MammuT::ChannelIDRange ch2;

    MyGraph(unsigned spes) : MammuT::Graph(spes)
    {
        setProcIDRange(p1, N);
        setProcIDRange(p2, K);
        setChannelIDRange(ch1, N);
        setChannelIDRange(ch2, K);
    }
};
```

A questo punto i processi dell'applicazione possono accedere ai range attraverso un'istanza della classe `MyGraph`.

2.4 Interfacce delle classi principali

2.4.1 Classe Guard

```
/**
 * @brief Guardia: interfaccia base
 */
class Guard
{
protected:
    /* ID della guardia (l'indice nell'array di guardie del
     * comando alternativo) */
    int m_id;
    /// Priorita' associata alla guardia
    int m_priority;
    /// Consente di abilitare/disabilitare la guardia
    bool m_enabled;

    /* La classe del comando alternativo e' dichiarata friend
     * per fare in modo che i metodi seguenti (protected)
     * siano invocabili ESCLUSIVAMENTE dall'interno del
     * comando alternativo */
    friend class AltCmd;

    /**
     * @brief Aggiorna l'ID della guardia.
     *
     * Tipicamente e' comodo far corrispondere l'id con
     * l'indice della guardia nell'array di guardie non
     * ordinato presente nel comando alternativo.
     */
     void setId(int id);

     /**
     * @brief Aggiorna la priorita' della guardia.
     */
    void setPriority(int priority);
```

```
public:
    /**
     * @brief Costruttore.
     *
     * @param priority priorita' associata alla guardia
     */
    Guard(int priority);

    /**
     * @brief Distruttore.
     */
    virtual ~Guard();

    /**
     * @brief Metodo da eseguire quando la guardia risulta
     *        verificata.
     *
     * @return 0 in caso di successo, -1 altrimenti
     */
    virtual int compute() = 0;

    /**
     * @brief Predicato associato alla guardia.
     *
     * @return true se il predicato e' verificato,
     *         false altrimenti
     */
    virtual bool evaluate();

    /**
     * @brief Abilita la guardia.
     */
    void enable();

    /**
     * @brief Disabilita la guardia.
     */
```

```
    void disable();

    /**
     * @brief Indica la disponibilita' dello slot di
     * comunicazione della guardia. Si dice che uno slot e'
     * disponibile se si ha la possibilita' di eseguire una
     * primitiva di comunicazione su un canale senza necessita'
     * di effettuare una sincronizzazione con il partner.
     */
    virtual bool slotAvailable() = 0;

    /**
     * @brief Restituisce l'ID della guardia.
     *
     * L'ID corrisponde all'indice della guardia nell'array di
     * guardie non ordinato presente nel comando alternativo.
     * Se la guardia non e' stata ancora aggiunta a un comando
     * alternativo, l'ID sara' uguale a -1.
     */
     int id() const;

    /**
     * @brief Rende il valore della priorita' della guardia.
     */
    int priority() const;

};
```

2.4.2 Classe ChannelArraySPE

```
#pragma once
#include <common/bit_vector.h>
#include <common/mammut_constants.h>
#include <spe/stcom.h>
#include <spe/channels/spe_channels.h>
```

```
#include <common/range.h>

template <class T_ch>
class ChannelArraySPE
{
protected:
    /// array di puntatori a canale
    ChType** m_chptr;
    /// range di ID dei canali dell'array
    ChannelIDRange m_range;
    /// numero di item da spedire per ogni send
    unsigned m_numItem;

public:
    typedef typename T_ch::ItemType ItemType;
    typedef T_ch ChType;

    /**
     * @brief Costruttore.
     *
     * @param range range di ID dei canali appartenenti all'array
     * @param com puntatore al supporto
     * @param nitem numero di item per messaggio
     *              spedito da un canale
     */
    ChannelArraySPE(const ChannelIDRange& range, MammuT::Stcom *com,
                    unsigned nitem = 1);

    /**
     * @brief Metodo per la registrazione di un canale.
     *
     * @param index indice del canale da registrare
     * @param com supporto su cui registrare il canale
     */
    void registerChannel(unsigned index, MammuT::Stcom *com);

    /**
     * @brief Metodo per la registrazione di tutti i canali.
```

```
     *
     * @param com supporto su cui registrare il canale
     */
    void registerAll(MammuT::Stcom *com);

    /**
     * @brief Restituisce un puntatore a canale
     * @param index indice del puntatore nell'array di canali
     */
    ChType* operator[](unsigned index) const;

    /**
     * @brief Restituisce un puntatore a canale
     * @param index indice del puntatore nell'array di canali
     */
    ChType* at(unsigned index) const;

    /**
     * @brief Restituisce il numero di canali dell'array
     */
    unsigned size() const;
};
```

2.4.3 Classe Graph

```
#pragma once
#include <common/mammut_constants.h>
#include <vector>
#include "range.h"
#include <stdio.h>
#include <assert.h>

/**
 * @brief Classe base per poter esplicitare
 *        il grafo LC dell'applicazione.
```

```
 */
class Graph
{
protected:
    /// numero di SPE disponibili
    unsigned m_spes;
    /** @brief contatore per l'assegnamento di range di
     * id consecutivi per gli array di processi */
    unsigned m_nextProcID;
    /** @brief contatore per l'assegnamento di range di
     * id consecutivi per gli array di canali */
    unsigned m_nextChannelID;

public:
    Graph(unsigned availableSPEs);

    /**
     * @brief Assegna un range di ID di processo consecutivi.
     *
     * @param r range di ID di processi da assegnare
     * @param num dimensione del range da assegnare
     */
    void setProcIDRange(ProcIDRange &r, unsigned num);

    /**
     * @brief Assegna un range di ID di canale consecutivi.
     *
     * @param r range di ID di canali da assegnare
     * @param num dimensione del range da assegnare
     */
    void setChannelIDRange(ChannelIDRange &r, unsigned num);
};
```

Capitolo 3

Strumenti di programmazione: il livello ASSIST

La libreria MammuT fornisce i costrutti tipici del linguaggio LC: canali, e array di canali, processi e array di processi, comando alternativo con guardie. Su questa base è possibile implementare un'ulteriore astrazione che corrisponde al costrutto *parmod* di ASSIST.

Tramite il *parmod* è possibile esprimere un modulo parallelo, costituito da tre sezioni: *Input Section* (*IS*), *Virtual Processors Section* (*VPS*) e *Output Section* (*OS*). La *Input Section* gestisce gli stream di input e si occupa di distribuire i dati ricevuti ai *Virtual Processors* (VP). La *Virtual Processors Section* consiste in un insieme di entità indipendenti e cooperanti tra loro (chiamate VP) il cui compito è quello di eseguire la computazione parallela; per questo motivo la *VP Section* viene anche definita come il motore di calcolo del *parmod*. La *Output Section* infine, si occupa di raccogliere il risultato della computazione eseguita sui VP trasmettendolo poi su un canale di output. Le tre sezioni cooperano secondo uno schema a pipeline; la sovrapposizione delle attività svolte nelle tre sezioni è spesso importante ai fini dell'ottimizzazione di performance e scalabilità, in modo particolare per mascherare le comunicazioni esterne in riferimento al tempo di calcolo dei

VP. Inoltre uno schema a pipeline consente di mascherare i ritardi dovuti a eventuali operazioni di pre/post processing che potrebbero non essere parallelizzabili in modo efficiente sulla *VP Section*.

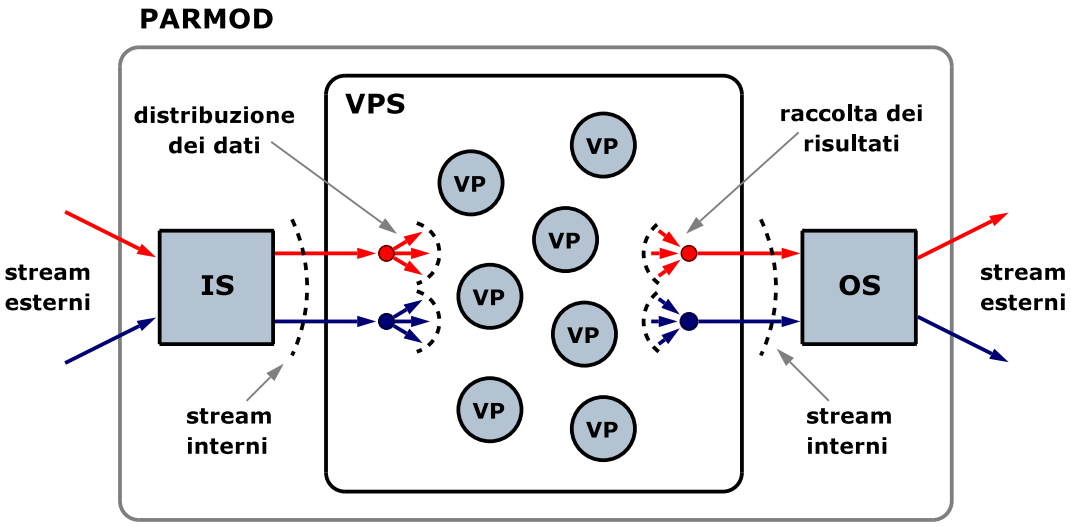

Figura 3.1: Struttura del *parmod* in ASSIST

3.1 Un esempio: data parallel con stencil fisso in ASSIST-CL

In questa sezione verranno illustrate le caratteristiche del linguaggio ASSIST-CL, analizzandone i costrutti e le potenzialità espressive tramite un semplice esempio. Si ipotizzi di voler scrivere un versione parallela di un semplice algoritmo di convoluzione su una matrice quadrata. Ogni elemento della matrice viene modificato da una funzione applicata a se stesso e agli elementi vicini. Per gli elementi sui bordi viene preso l'elemento sul bordo opposto, secondo uno schema "toroidale" su entrambe le direzioni. L'algoritmo è espresso dal seguente pseudocodice:

```
for (int k = 0; k < ITERATIONS; i++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            new_a[i][j] = 4 * a[i][j] -
                              (a[(j - 1 + N) % N][j] +
                               a[(i + 1) % N][j] +
                               a[i][(j - 1 + N) % N] +
                               a[i][(j + 1) % N]);
        }

        /* scambio puntatori */
        swap(a, new_a);
    }
}
```

La versione parallela lavora su uno stream di matrici, prodotto da un processo sequenziale `generator`. Un *parmod* `convolution` eseguirà il calcolo e genererà uno stream di risultati, ricevuti da un processo sequenziale `save`. Il codice in ASSIST-CL è il seguente:

```
#define ITERATIONS ...
#define N ...

generic main() {
    stream float A[N][N];
    stream float res[N][N];
    generator (output_stream A);
    convolution (input_stream A output_stream res);
    save (input_stream res);
}


generator(output_stream float A[N][N]) {
    f_produce(output_stream A);
}
```

```
proc f_produce(output_stream float A[N][N])
$c++{
    float matrix[N][N];

    while (...) {
        ... inizializzazione matrix ...

        assist_out(A, matrix);
    }
}c++$


parmod convolution (input_stream float A[N][N]
                    output_stream float res[N][N])
{
    topology array [i:N][j:N] Pv;
    attribute float S[N][N];
    stream float vp_res;

    do input_section {
        guard1: on , , A {
            distribution A[*i0][*j0] scatter to S[i0][j0];
        }
    } while (true)


    virtual_processors {
        elab1 (in guard1 out ris) {
            VP i,j {
                for (int k = 0; k < ITERATIONS; k++) {
                    conv(in S[i][j], S[(i - 1 + N) % N][j],
                                     S[(i + 1) % N][j],
                                     S[i][(j - 1 + N) % N],
                                     S[i][(j + 1) % N]
                         out S[i][j]);
                }
```

```
                assist_out(vp_res, S[i][j]);
            }
        }
    }

    output_section {
        collects vp_res from ALL Pv[i][j] {
            int el;
            int B_[N][N];
            AST_FOR_EACH(el) {
                B_[i][j] = el;
            }
            assist_out (B, B_);
            }<>;
        }
    }
}


proc conv(in float center, float left, float right,
             float down, float up
          out result)
$c++{
    result = 4.0f - (left + right + down + up);
}c++$


save (input_stream float res[N][N])
$c++{
    ...
}c++$
```

Il blocco `generic` definisce la struttura dell'applicazione come combinazione di stream, processi sequenziali e *parmod*, riproducendo la struttura descritta in precedenza.

Il *parmod* `convolution` è definito attraverso i tre blocchi `input_section`, `virtual_processors` e `output_section`. L'espressione `topology array` offre un meccanismo di *naming* dei VP appartenenti al *parmod*. In questo caso viene dichiarata una topologia di N x N VP. Tramite il costrutto `attribute` viene dichiarata una matrice S. L'elemento `(i, j)` della matrice S viene scritto esclusivamente dal VP `(i, j)`, secondo la *owner computes rule* (vedi [8], capitolo VII).

Nel blocco `input_section` è presente una guardia `guard1` che distribuisce ai VP gli elementi ricevuti sullo stream `A` tramite una distribuzione `scatter`. La regola utilizzata nel costrutto assegna al VP `(i, j)` l'elemento `A[i][j]`, accedibile tramite l'attributo S.

Nella sezione `virtual_processors` viene espressa la computazione effettuata da ogni VP del *parmod*. La regola di attivazione `elab1`, connessa alla guardia `guard1`, indica che il VP eseguirà la computazione specificata dalla procedura *conv* in seguito alla ricezione di un dato sullo stream connesso alla guardia `guard1`. L'elaborazione effettuata dal VP `(i, j)` itera il calcolo del valore `S[i][j]` sulla base dei quattro valori "vicini". Il calcolo di un elemento dell'array è contenuto nella procedura `conv`. E' importante sottolineare che il calcolo sui VP viene espresso con la grana più fine possibile. Un VP non corrisponde ad un processo o a un thread fisico, ed è compito del compilatore stabilire il mapping tra processi virtuali e reali. Quando tutte le iterazioni su un elemento sono state effettuate, il risultato viene spedito sullo stream interno `vp_res`.

Nella blocco `output_section` viene implementata la *gather* della matrice risultato, tramite una politica `collects from ALL` sullo stream interno, associata al costrutto `AST_FOR_EACH`, che copia l'elemento ricevuto nella locazione `(i, j)` di una matrice da inviare in output quando tutti gli elementi della matrice sono stati ricevuti.

Nel codice esposto non c'è traccia delle comunicazioni necessarie tra VP per scambiare gli elementi vicini. Questa è una astrazione offerta dal linguaggio: sarà compito del compilatore analizzare le dipendenze tra i dati,

e in base a queste stabilire un mapping ottimale dei precessori virtuali sui processi fisici, predisponendo i canali e le comunicazioni necessarie per la realizzazione dell'algoritmo.

3.2 Input Section

La *Input Section* gestisce un certo numero di stream di input. L'attivazione del modulo parallelo avviene in maniera nondeterministica al ricevimento di un dato su un sottoinsieme di tali input stream. La semantica del nondeterminismo è quella del modello CSP basata su un comando alternativo con guardie. Ogni guardia contiene al più tre elementi, ciascuno dei quali può anche non essere presente: una guardia di input (costituita da uno o più input stream, in AND tra loro), una guardia locale (un predicato booleano, che abilita o disabilita la guardia) e una priorità che può essere modificata a programma. Ad ogni stream di input è associata una politica di distribuzione dei valori ricevuti ai VP. Sono possibili diverse strategie di distribuzione:

- *on demand*: il valore ricevuto è trasmesso ad un VP "pronto", scelto in modo nondeterministico;
- *scatter*: il valore strutturato ricevuto è partizionato tra i VP secondo una regola espressa a programma;
- *multicast*: il valore ricevuto è spedito a tutti i VP;
- *scheduled*: il valore ricevuto è spedito a un determinato VP scelto a programma.

La *Input Section* è programmabile e consente di definire una fase di preprocessing sui dati prima della loro distribuzione ai VP. In alcune applicazioni è presente un processo manager che comunica con la *Input Section* tramite un canale di controllo. Generalmente i valori ricevuti su questo canale non devono essere distribuiti ai VP dunque è stata prevista la possibilità di specificare un input stream a cui non sia associata nessuna politica di distribuzione.

3.3 Virtual Processors Section

L'attivazione dei singoli VP della *VP Section* avviene in seguito alla ricezione di uno o più valori distribuiti dalla *Input Section*.
È doveroso rimarcare che il nondeterminismo si trova sulla *Input Section* e non sui VP, che sono invece dei semplici esecutori. Se ogni VP gestisse i propri input in modo nondeterministico potrebbero sorgere diversi problemi. In primo luogo potrebbe venire alterato l'ordine in cui vengono processati gli elementi ricevuti sugli stream di input della *Input Section*; questo per alcuni tipi di computazioni potrebbe non essere accettabile in quanto i risultati ottenuti potrebbero essere influenzati dal riordinamento. Inoltre potrebbero verificarsi situazioni di stallo, soprattutto nei casi di computazioni con stencil. Ciò comporta la necessità di avere un meccanismo che consenta di indicare ai VP gli input che porteranno alla prossima attivazione. Questo aspetto viene gestito da parte della *Input Section* spedendo un codice operativo che indica ai VP quali input dovranno utilizzare.

3.4 Output Section

La *Output Section* gestisce un certo numero di stream di output, generati a partire dalla raccolta dei risultati provenienti dai VP. Analogamente alla *Input Section*, anche la *Output Section* è programmabile e consente di definire una fase di post-processing sui dati ricevuti. Sono possibili tre diverse politiche di collezionamento:

- *from any*: il codice di post-processing viene eseguito in seguito alla ricezione di dati provenienti da uno dei VP, scelto in modo nondeterministico;

- *from one*: il codice di post-processing viene eseguito alla ricezione di dati da un preciso VP, selezionabile a programma;

- *from all*: il codice di post-processing viene eseguito alla ricezione di dati da tutti i VP;

Per una panoramica più completa su ASSIST, si vedano [4, 5].

3.5 Semantica

È possibile esprimere i costrutti di ASSIST tramite i costrutti del linguaggio LC. Da questa relazione si può ricavare un'implementazione basata sulla libreria MammuT. In questo paragrafo verrà analizzata nel dettaglio la semantica del *parmod* di ASSIST.

Ad ogni input stream corrisponde un canale di comunicazione che ha come destinatario il processo della *Input Section*. Per distribuire i dati ricevuti è necessario predisporre un canale di comunicazione verso ogni VP, ed è comodo raggruppare su un array i canali utilizzati per la distribuzione dei dati di un input stream. I canali di comunicazione sono tipati e questo comporta la necessità di avere array di canali diversi per distribuire i dati ricevuti da input stream differenti.

La politica di distribuzione dei dati in arrivo su un input stream è modificabile a tempo di esecuzione. La distribuzione *scatter* presenta una peculiarità rispetto alle altre, perchè il numero di elementi del messaggio spedito ad ogni VP è una frazione del numero di elementi del messaggio ricevuto sull'input stream. Il tipo ed il numero di elementi dei messaggi di un canale di comunicazione sono parametri costanti, perciò è necessario prevedere un array di canali apposito per gestire la distribuzione *scatter*.

Ogni VP ha in ingresso un certo numero di canali per ricevere i dati distribuiti dalla *Input Section*. La specifica del linguaggio di coordinamento di ASSIST prevede la dichiarazione di una o più regole di attivazione per i VP, ognuna identificata univocamente dagli input stream associati. L'ordine delle attivazioni sui VP è dettato dall'ordine in cui vengono gestiti gli input stream sulla *Input Section*, e un modo semplice per propagare tali informa-

zioni consiste nell'introduzione di un array di canali tra la *Input Section* e i VP. Questo array di canali è creato e gestito esclusivamente dal supporto, e le informazioni trasportate sono "codici operativi" che indicano l'insieme di canali su cui il VP riceverà i dati necessari alla regola di attivazione corrispondente. Tali codici operativi possono quindi essere considerati dei token di attivazione.

Dal punto di vista del VP, questo schema si snoda in due fasi.
Nella prima fase, il VP effettua una receive sul canale di controllo, ottiene il codice operativo relativo all'attivazione corrente, ed effettua la receive dei dati sull'insieme dei canali individuato dal codice operativo. Nella seconda fase, il VP esegue la funzione di elaborazione relativa alla regola di attivazione individuata. È bene precisare che questa funzione deve essere scritta dal programmatore, e può prevedere comunicazioni con altri VP o con processi esterni al *parmod*. L'unico aspetto della *VP Section* gestito automaticamente dalla libreria è l'attivazione; non sono previste astrazioni aggiuntive per supportare algoritmi data-parallel con stencil (fisso o variabile), quindi è compito del programmatore esplicitare i pattern di comunicazione tra VP. I processi della VP Section sono a tutti gli effetti processi LC: nella libreria non esiste il concetto di "processo virtuale".

Oltre alle comunicazioni tra VP, il programmatore deve preoccuparsi di gestire l'invio dei risultati della computazione alla *Output Section*.

Ad ogni output stream corrisponde un canale di comunicazione che ha come mittente il processo della *Output Section*. Per ricevere i risultati è necessario predisporre un canale di comunicazione tra ogni VP e il processo dell'*Output Section*, analogamente allo schema usato sulla *Input Section*. La *Output Section* gestisce in maniera nondeterministica l'arrivo dei risultati spediti dai processi della *VP Section*. La politica di collezionamento dei dati per un output stream è modificabile a tempo di esecuzione.

3.6 Implementazione

Uno degli obiettivi della libreria è l'esposizione di un'interfaccia semplice per la dichiarazione e l'utilizzo dei costrutti del *parmod*. La creazione di *Input Section*, *VP Section* e *Output Section* dovrà essere dichiarativa evitando che il programmatore debba utilizzare esplicitamente i costrutti del livello MammuT come canali, guardie o comandi alternativi. Per poter automatizzare la fase di dichiarazione del *parmod* è necessario però avere a disposizione un certo numero di informazioni. Durante la presentazione del livello MammuT si è parlato dell'introduzione di un grafo dell'applicazione per memorizzare gli id di processi e canali. È sufficiente quindi estendere il grafo MammuT per aggiungere le informazioni relative ai costrutti introdotti nel livello ASSIST. L'idea è quella di avere un descrittore per ogni costrutto del *parmod*; ogni descrittore contiene i dati necessari all'inizializzazione del costrutto cui si riferisce. Nel seguito di questo paragrafo verrà analizzata l'implementazione delle sezioni del *parmod* e contestualmente saranno introdotti i relativi descrittori utilizzati nel grafo. Una parte dell'esposizione sarà sempre dedicata all'analisi delle interfacce dei costrutti cercando di enfatizzare il punto di vista del programmatore che dovrà utilizzare la libreria.

3.6.1 Codice operativo

La semantica introdotta relega il nondeterminismo sulla *Input Section* e richiede un certo grado di coordinamento tra *Input Section* e *VP Section*. Nel paragrafo precedente si è accennato all'uso di codici operativi come token di attivazione dei VP. Il compito principale dei codici operativi è indicare ai VP su quali canali ricevere i dati necessari per l'attivazione, e nell'implementazione si è scelto di esprimere tale lista tramite una maschera di bit, ognuno dei quali corrisponde ad un canale in ingresso sui VP. Una maschera di 32 bit consente quindi di avere fino a 32 canali tra *Input Section* e ognuno dei VP. Questo non è mai un problema in quanto il limite al numero di canali

utilizzabili è dettato dall'architettura: sugli SPE i bit utilizzabili per i segnali sono 64, che nel caso peggiore si traducono in 32 canali.

Il programmatore potrebbe aver bisogno di comunicare qualche altra informazione dalla *Input Section* alla *VP Section*. Ad esempio si potrebbe voler segnalare il verificarsi di una certa condizione o il valore di un certo contatore. Per soddisfare questa eventuale esigenza si è previsto di affiancare alla maschera di bit un ulteriore campo intero gestito esclusivamente dal programmatore sia lato *Input Section* sia lato VP.

Un codice operativo è quindi una struttura composta da due interi: il primo è la maschera di bit mentre il secondo è messo a disposizione del programmatore per altre comunicazioni tra *Input Section* e VP. È opportuno sottolineare che dal punto di vista delle prestazioni la spedizione di due interi anzichè uno non comporta nessun overhead in quanto la dimensione dei messaggi spediti è sempre allineata ai 128 Byte nei canali SPE - SPE e a 16 Byte in quelli SPE - PPE.

```
/* Struttura per il codice operativo */
struct Opc
{
    Opc();
    /* Maschera di bit */
    int mask;
    /* Codice custom */
    int customID;
};
```

3.6.2 Input Section

L'implementazione attuale della *Input Section* è basata sul lavoro di tirocinio svolto da Silvia Lametti presso il Laboratorio di Architetture Parallele del Dipartimento di Informatica, descritto in [10].

La *Input Section* rappresenta l'interfaccia di ingresso al *parmod*. Gli input sono gestiti in modo nondeterministico: questo significa che per ogni input stream è presente una guardia nel comando alternativo della *Input Section* (ovviamente nel caso di input stream in AND, la guardia sarà unica). Ad ogni input stream corrisponde un canale di comunicazione che ha come destinatario il processo della *Input Section*. I dati ricevuti vengono poi distribuiti ai VP tramite un apposito array di canali di comunicazione. La politica di distribuzione associata ai valori in arrivo su un determinato canale in ingresso alla *Input Section* può essere modificata a tempo di esecuzione. Il fatto che la distribuzione *scatter* richieda un array di canali apposito, rende necessario specificare staticamente la lista delle distribuzioni attivabili in modo da poter dichiarare solamente gli array di canali necessari. Queste informazioni possono essere incapsulate in un descrittore apposito per l'input stream. La classe `InputStreamData` si occupa di modellare tale descrittore di input stream: essa gestisce l'assegnamento di una maschera di bit a cisacun array di canali verso i VP rendendo possibile il funzionamento del meccanismo dei codici operativi precedentemente descritto; inoltre memorizza informazioni come id del processo mittente, elenco delle distribuzioni abilitate e distribuzione di default, id del canale in ingresso e range di id relativi ai canali verso i VP, id dei processi VP. È possibile vedere l'interfaccia della classe `InputStreamData` nel paragrafo 3.7.1.

Le varie politiche di distribuzione dei dati ricevuti sull'input stream sono modellate con diverse classi (una per ogni politica), che implementano un'unica interfaccia comune. Ogni classe che implementa l'interfaccia `Distribution` fornisce la propria implementazione dei metodi `distribute` e `waitDistr`, utili rispettivamente per distribuire un valore ai VP e per attendere che la precedente operazione di distribuzione sia terminata. È bene sottolineare che la fase di distribuzione dei dati ai VP, implementata dal metodo `distribute`, comprende sempre sia l'invio dei dati che l'invio del codice operativo ai soli VP destinatari.

```
class Distribution
{
public:
    /**
     * @brief Costruttore.
     */
    Distribution();

    /**
     * @brief Distribuisce i dati ricevuti in input ai VP.
     *
     * @param msg puntatore al prossimo messaggio da "distribuire"
     * @param opcMask valore del campo mask del codice operativo
     *                da spedire al/ai VP
     */
    virtual void distribute(void *msg, int opcMask);

    /**
     * @brief Consente di attendere che l'ultima operazione
     *        di distribuzione effettuata sia completata, in
     *        modo da poter aggiornare il valore del messaggio.
     */
    virtual void waitDistr() {}
};
```

La classe `OnDemand` modella la politica di distribuzione *on demand* tramite un comando alternativo in uscita. Il comando alternativo consente di scegliere il VP libero a cui mandare i dati in base allo stato del canale che connette quel particolare VP alla *Input Section*: se è possibile si spediscono i dati sul canale, altrimenti si analizza lo stato del canale associato alla guardia successiva nel comando alternativo. L'implementazione classica della distribuzione *on demand* prevede l'utilizzo di canali di ritorno dai VP alla *Input Section* utilizzati da ognuno dei VP per comunicare la propria disponibilità. Questo anche perchè nel linguaggio LC il comando alternativo con guardie è previsto solo in ingresso; il livello MammuT della libreria estende il concet-

to introducendo la versione in uscita del costrutto comando alternativo. In un'architettura CELL, tale implementazione risulta più conveniente rispetto a quella classica in quanto consente di risparmiare i bit di segnale associati ai canali di ritorno, visto che il numero di bit di cui sono composti i registri dei segnali è limitato (2 registri da 32 bit su ogni SPE). Inoltre non è necessarrio aspettare la comunicazione di disponibilità da parte del VP in quanto la *Input Section* può dedurla dallo stato del canale.

La classe `Scatter` modella l'omonima politica di distribuzione. Attualmente si suppone che il valore composto da scatterizzare sia un array unidimensionale di elementi; l'array viene diviso in blocchi di uguali dimensioni e ciascuno di questi blocchi viene spedito a un VP diverso. Una possibile estensione consisterebbe nel rendere la politica di suddivisione del valore composto personalizzabile: ad esempio si potrebbe ricevere una matrice bidimensionale di elementi e la si potrebbe voler dividere a righe, a colonne, oppure ancora a blocchi.

La classe `Multicast` modella la politica di distribuzione *multicast*; l'implementazione consiste in una serie di send ai VP e da questo punto di vista rappresenta solo un'emulazione di una multicast.

La classe `Scheduled` infine, modella la politica di distribuzione *scheduled*. Tramite un apposito metodo `setChannelOut` il programmatore può impostare il VP destinatario e l'operazione di distribuzione consiste in una send al VP destinatario scelto.

I canali di comunicazione, sia in ingresso che in uscita, e le distribuzioni vengono gestite nella classe `InputStream`. Si rimanda al paragrafo 3.7.2 per l'interfaccia di tale classe. Uno dei parametri del costruttore della classe `InputStream` è il descrittore dell'input stream; in base alle impostazioni lette dal descrittore vengono create le distribuzioni necessarie e i relativi array di canali e viene impostata la distribuzione di default. Se almeno una tra le distribuzioni *on demand*, *multicast*, *scheduled* è abilitata viene creato un array di canali verso i VP. Se la distribuzione *scatter* è abilitata viene creato

un array di canali apposito. Nel caso di un input stream con nessuna distribuzione abilitata non saranno creati array di canali. La classe `InputStream` si occupa inoltre di gestire il canale di comunicazione in ingresso alla *Input Section*. Il metodo `setCurrentDistr` consente di impostare la distribuzione corrente scegliendo tra quelle abilitate.

Le guardie del comando alternativo della *Input Section* sono modellate dalle classi `GuardDistrOne` e `GuardDistrMulti`. Entrambe implementano l'interfaccia `GuardDistr`, mostrata nel paragrafo 3.7.3; la prima viene usata per rappresentare input stream singoli (con un solo canale in ingresso) mentre la seconda viene usata per rappresentare input stream multipli (con più canali in ingresso in AND). Il metodo `compute` della classe `GuardDistrOne` effettua la ricezione dei dati, chiama il metodo per la fase di pre-elaborazione e infine li distribuisce. La ricezione e la distribuzione dei dati viene gestita tramite un oggetto `InputStream`, creato nel costruttore. La classe `GuardDistrMulti` è implementata come composizione di oggetti `GuardDistrOne`. Nel metodo `compute` viene effetuata la ricezione dei dati sui canali associati alle guardie singole, viene chiamato il metodo per la fase di pre-elaborazione e infine viene effettuata la distribuzione dei dati sugli array di canali associati alle guardie singole. In entrambi i casi la guardia si occupa anche di impostare il campo maschera del codice operativo da inviare: nella guardia singola la maschera identifica l'array di canali corrispondente alla distribuzione corrente mentre nella guardia multipla la maschera sarà ottenuta mettendo in OR le maschere corrispondenti alle guardie interne.

La classe `InputSection` gestisce la ricezione dei dati tramite un comando alternativo e l'invio dei codici operativi ai VP tramite un array di canali. Anche questo costrutto ha associato un descrittore, modellato dalla classe `ISData` (interfaccia nel paragrafo 3.7.4). Le principali informazioni contenute nel descrittore sono l'id del processo che esegue la *Input Section*, gli id dei canali dedicati all'invio dei codici operativi e gli id dei processi che eseguono

i VP della *VP Section*.

Il costruttore della classe `InputSection` utilizza le informazioni presenti nel descrittore per inizializzare il comando alternativo e l'array di canali per i codici operativi. Vale la pena di spendere qualche parola per presentare i principali metodi della classe; per l'interfaccia completa della classe si rimanda al paragrafo 3.7.5. Il metodo `sendCop` si occupa dell'effettiva spedizione di un codice operativo a un determinato VP e viene invocato dalle classi che implementano le varie distribuzioni. Il metodo `go` effettua un passo del comando alternativo. Il metodo `preComputation` è pensato per contenere il codice delle operazioni da effettuare nella fase di pre-elaborazione. Quella fornita è un'implementazione vuota per i casi in cui non sia prevista una fase di pre-processing. Negli altri casi il programmatore dovrà estendere la classe `InputSection` e fornire una propria implementazione del metodo `preComputation`. Questo consente di personalizzare in modo piuttosto pesante il funzionamento della *Input Section*. Ad esempio si può voler cambiare il tipo di distribuzione associato a un certo input stream, al verificarsi di una certa condizione; oppure si può decidere che la receive di un messaggio proveniente da un certo processo manager provoca la ricezione di un dato da un altro input stream (in modo deterministico, quindi) e la sua distribuzione ai VP.

Il costruttore della classe `InputSection`, tra le altre cose, inizializza il comando alternativo; le guardie però dovrebbero essere create dal programmatore e aggiunte alla *Input Section* tramite il metodo `addInputGuard`. Per nascondere i dettagli relativi alle guardie si è deciso di creare un piccolo wrapper che consentisse di costruire una *Input Section* in modo più dichiarativo. Il wrapper è stato implementato tramite due classi chiamate `Emit` ed `EmitAND`, che consentono di esprimere rispettivamente un input stream singolo e più input stream in AND. Per vedere l'interfaccia delle due classi si faccia riferimento ai paragrafi 3.7.6 e 3.7.7.

La *Input Section* viene costruita come composizione di più costrutti `Emit` (sia singoli che multipli); l'`Emit` multiplo viene costruito come composizione

di `Emit` singoli. Per rendere possibile ciò la classe `Emit` è stata dotata di due costruttori che differiscono per il tipo del primo parametro, che rappresenta una sorta di "contenitore" dell'oggetto. Per creare un input stream singolo è sufficiente creare un'istanza della classe `Emit` utilizzando il costruttore che prende come primo parametro il puntatore all'oggetto `InputSection`. Per creare un input stream multiplo bisogna innanzitutto creare un'istanza della classe `EmitAND`. A questo punto la composizione degli `Emit` singoli nell'`EmitAND` avviene semplicementente creando le istanze della classe `Emit` utilizzando il costruttore che prende come primo parametro il puntatore all'oggetto `EmitAND`.

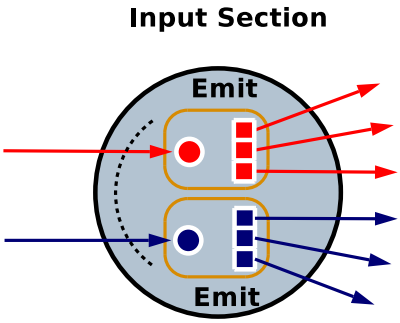

Figura 3.2: Input Section come composizione di costrutti `Emit`

3.6.3 VP Section

L'attivazione dei VP, come introdotto nella parte relativa alla semantica, si snoda in due fasi. La prima fase consiste nella ricezione del codice operativo sul canale dedicato, e la ricezione dei dati indicati dal codice operativo. La seconda fase consiste nell'esecuzione della funzione di elaborazione relativa alla regola di attivazione individuata. Nell'implementazione è stata automatizzata la prima fase. Questo perchè automatizzando le receive sui VP, il numero di send e receive tra *Input Section* e *VP Section* sarà sempre bilan-

ciato, sgravando il programmatore da un lavoro prono a errori.

Quando nel grafo ASSIST vengono dichiarati i descrittori degli input stream, viene assegnato un bit per ogni array di canali necessario per la distribuzione di dati tra *Input Section* e VP. Ipotizzando che gli array di canali siano N, ogni VP avrà N canali in ingresso dedicati per la ricezione dei dati distribuiti dalla Input Section (un canale per ogni array). A questo punto è possibile automatizzare la ricezione dei dati in maniera diretta, ponendo gli N puntatori a canali in ingresso in un array, ed effettuando una receive sul canale i-esimo solo se il bit i-esimo della maschera è attivato.

In realtà l'implementazione deve tenere conto di un ulteriore fattore. I canali sono tipati, le classi dei canali sono tutte template, e il metodo `receive` dei canali non è `virtual` per motivi di efficienza. Tuttavia, i canali in ingresso alla *VP Section* possono essere di tipo diverso, perchè veicolano dati provenienti da diversi *Input Stream*. Per questo motivo, la classe `VPSection` non può gestire un semplice array di puntatori di canale su cui invocare la `receive`. Il problema è stato risolto predisponendo una serie di classi che forniscono il polimorfismo assente nelle classi dei canali. La classe `ChWrapperBase` fornisce un'interfaccia comune per questa situazione, riproducendo la stessa interfaccia dei canali lato destinatario, ma con metodi `virtual`. Una classe template `ChWrapper` eredita dalla classe `ChWrapperBase` e implementa tutti i metodi `virtual`. I canali gestiti dalla classe `VPSection` sono tutti istanze della classe `ChWrapper`, e tramite un array di puntatori a `ChWrapperBase` la classe `VPSection` può gestire automaticamente canali di tipo diverso.

L'interfaccia completa della classe `VPSection` si trova nel paragrafo 3.7.8; non è necessario un descrittore associato alla *VP Section* in quanto viene utilizzato quello associato alla *Input Section* corrispondente. Il costruttore registra il canale per il codice operativo corrispondente al VP corrente e istanzia l'array di puntatori per i canali in ingresso. Il descrittore `ISData` è

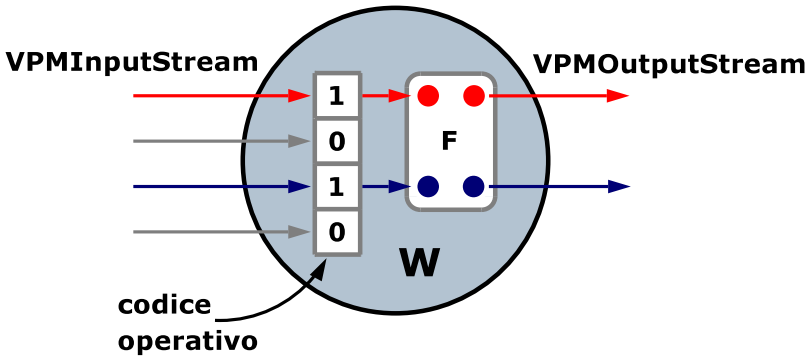

Figura 3.3: VP Section, codice operativo e gestione di input ed output

inizializzato nel grafo in maniera tale da rendere disponibile il numero di bit allocati per gli input stream in ingresso alla *Input Section*. In questo modo, il costruttore della classe `VPSection` può conoscere il numero di puntatori a canale necessari per l'automazione della ricezione dei dati.

La dichiarazione e la registrazione automatica dei canali in ingresso viene gestita per mezzo della classe `VPMInputStream` (l'interfaccia è nel paragrafo 3.7.9). Dal descrittore dell'input stream associato, passato come secondo parametro, il costruttore deriva l'id dei canali in ingresso al VP corrente, i canali in ingresso vengono registrati, e i puntatori ai canali vengono registrati nell'array gestito dalla classe `VPSection`. Se il canale appartiene all'array di canali identificato dal bit i-esimo, il puntatore al canale viene salvato nella i-esima locazione dell'array di puntatori. In questo modo la classe `VPSection` può fornire l'associazione bit - canale necessaria all'automazione della ricezione dei dati.

La classe `VPSection` fornisce un metodo `activate` che effettua la ricezione automatica dei dati provenient dalla *Input Section*. È compito del programmatore specificare il codice da eseguire in seguito alla ricezione di una particolare combinazione di input. Il metodo `activate` restituisce il codice operativo ricevuto, e il programmatore può esaminare il valore della

maschera di bit in modo da capire quali dati sono stati ricevuti in seguito all'attivazione. È possibile accedere ai dati ricevuti tramite opportuni metodi d'accesso della classe `VPMInputStream` (`msg` e `msgScatter`).

La classe `VPMOutputStream` consente di dichiarare sui VP i canali su cui inviare i risultati della computazione; la sua interfaccia si trova nel paragrafo 3.7.10. Così come la classe `VMPInputStream` serve per dichiarare e registrare automaticamente sui VP i canali in ingresso corrispondenti ad un dato input stream, la classe `VMPOutputStream` permette di dichiarare e registrare automaticamente sui VP i canali in uscita corrispondenti ad un dato output stream. Questo aspetto verrà approfondito nella sezione relativa all'implementazione della *Output Section*.

3.6.4 Output Section

La *Output Section* rappresenta l'interfaccia di uscita dal *parmod*. Ad ogni output stream corrisponde un canale di comunicazione che ha come mittente il processo della *Output Section*. È presente un comando alternativo per gestire in maniera nondeterministica l'arrivo dei risultati spediti dai processi della *VP Section*. La comunicazione di tali risultati dai VP alla *Output Section* avviene tramite un apposito array di canali. La politica di collezionamento dei dati associata ad un output stream è modificabile a tempo di esecuzione. Ogni politica di collezionamento viene implementata con una guardia o con un insieme di guardie apposito ed è quindi necessario specificare staticamente la lista delle politiche attivabili in modo da creare solo le guardie necessarie. Analogamente a quanto fatto per gli input stream, queste informazioni vengono incapsulate in un descrittore apposito per l'output stream, modellato tramite la classe `OutputStreamData`. Tra i dati memorizzati da un oggetto di tipo `OutputStreamData` ci sono gli id dei processi VP e l'id del processo destinatario, il range di id relativo ai canali per ricevere i dati dai VP e l'id

del canale in uscita, oltre naturalmente all'elenco delle politiche di collection abilitate e a quella di default.

I canali di comunicazione, sia in ingresso che in uscita vengono gestiti nella classe `OutputStream`. Uno dei parametri del costruttore della classe `OutputStream` è il descrittore dell'output stream; in base alle impostazioni lette dal descrittore viene creato un array di canali in ingresso per ricevere i dati dai VP e un canale in uscita verso il processo destinatario.

È possibile trovare le interfacce delle classi `OutputStreamData` e `OutputStream` nei paragrafi 3.7.11 e 3.7.12.

Nella parte introduttiva di questo capitolo si è detto che la libreria fornisce tre diverse politiche di collezionamento dei dati. Andando ad analizzare più nel dettaglio l'implementazione però, si può vedere che le politiche *from any* e *from all* hanno due versioni ciascuna, che differiscono tra loro per un piccolissimo dettaglio. Si può quindi affermare che le politiche di collection a disposizione sono in realtà cinque, e non tre. Le due versioni della politica *from any* prendono il nome di *from any* e *forward* mentre le due versioni della politica *from all* prendono il nome di *from all* e *gather*. La logica dietro ciascuna politica di collection è contenuta in un diverso tipo di guardia:

- la classe `GuardCollectFromAny` viene utilizzata per modellare sia la politica *from any* che quella *from one*;
- la classe `GuardCollectForward` modella la politica *forward*;
- la classe `GuardCollectFromAll` modella la politica *from all*;
- la classe `GuardCollectGather` modella la politica *gather*.

Le classi `GuardCollectFromAny` e `GuardCollectForward` ereditano dalla classe `GuardCollectOne`, un tipo di guardia che prevede un solo canale in ingresso. Le classi `GuardCollectFromAll` e `GuardCollectGather` ereditano

dalla classe `GuardCollectMulti`, un tipo di guardia che prevede più canali gestiti in AND.

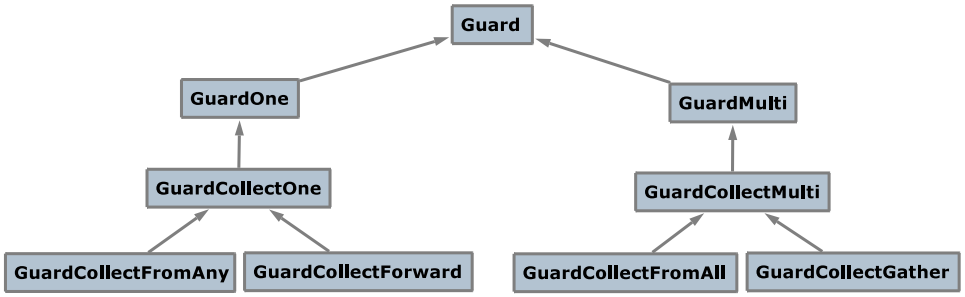

Figura 3.4: Gerarchia delle classi `GuardCollect*`

Le guardie che implementano le politiche di collezionamento associate a uno stesso output stream condividono i canali contenuti in quello stream. Per rendere possibile ciò ogni guardia ha un puntatore all'oggetto output stream cui è associata.

Per implementare una politica di collezione *from any*, *from one* o *forward* è necessario creare una guardia per ogni VP; nei primi due casi la guardia sarà un oggetto istanza della classe `GuardCollectFromAny` mentre nell'ultimo caso sarà un'istanza della classe `GuardCollectForward`. Le guardie vengono inserite tutte nel comando alternativo ma, se un output stream ha più politiche di collection associate, solo le guardie relative alla politica corrente saranno abilitate. Nel caso della politica *from one*, che prevede di ricevere da un determinato VP scelto a programma, solo la guardia corrispondente al VP da cui si vuole ricevere sarà abilitata in un certo istante.

Si analizzeranno ora le implementazioni dei metodi `compute` nelle diverse classi `GuardCollect*` per vedere le differenze tra le varie politiche di collezionamento dei dati. Tutte le implementazioni seguono uno schema generale che prevede di effettuare la receive dal canale o dai canali associati alla guardia, chiamare una funzione di post-computation ed effettuare l'invio del messaggio di output al processo destinatario. La classe `OutputStream` fornisce un metodo `setMsg` per impostare il puntatore al messaggio da inviare sul canale

in uscita, un metodo `send` per inviare tale messaggio e un metodo `wait` per attendere che l'invio dell'ultimo messaggio sia concluso, prima di modificare il messaggio.

La differenza tra la classe `GuardCollectForward` e `GuardCollectFromAny` sta nel fatto che nel primo caso il messaggio ricevuto dal VP viene automaticamente settato come prossimo messaggio da inviare sul canale in uscita mentre nel secondo caso questo non avviene e dovrà essere il programmatore a farlo nel metodo contenente il codice di post-elaborazione. Questo significa che utilizzando una politica *from any* il programmatore **deve** fornire un metodo per la fase di post-computation mentre utilizzando una politica *forward* tale metodo è opzionale.

Un discorso analogo può essere fatto confrontando la classe `GuardCollectGather` con la classe `GuardCollectFromAll`. Nel primo caso viene effettuata una receive da tutti i canali associati alla guardia, il messaggio da spedire sul canale in uscita viene ricomposto tramite un'operazione di gather e il suo puntatore viene passato al metodo `setMsg` della classe `OutputStream` per impostarlo come prossimo messaggio da inviare sul canale di output. Nel secondo caso invece, sarà il programmatore a doversi occupare di ricomporre il messaggio e impostare il prossimo messaggio da spedire sul canale di output. Quindi utilizzando una politica *from all* il programmatore **deve** fornire un metodo per la fase di post-computation mentre utilizzando una politica *gather* tale metodo è opzionale.

La classe `OutputSection` contiene solamente un comando alternativo e qualche altro metodo. Anche alla *Output Section* è associato un descrittore, modellato dalla classe `OSData`. Le principali informazioni contenute in tale descrittore sono l'id del processo che esegue l'*Output Section*, gli id dei processi che eseguono i VP della *VP Section* e un contatore per il numero di guardie da inserire nel comando alternativo.

Il costruttore della classe `OutputSection` utilizza il contatore delle guardie presente nel descrittore per inizializzare il comando alternativo.

È possibile trovare le interfacce delle classi `OSData` e `OutputSection` nei

paragrafi 3.7.13 e 3.7.14.

Analogamente alla classe `InputSection`, anche la classe `OutputSection` ha un metodo `go` per effettuare un passo del comando alternativo. Il codice delle operazioni da effettuare nella fase di post-elaborazione è contenuto nel metodo `postComputation`. L'implementazione del metodo fornita è quella vuota per i casi in cui non sia prevista una fase di post-processing. Negli altri casi il programmatore dovrà estendere la classe `OutputSection` e fornire una propria implementazione del metodo `postComputation`. In precedenza si sono discussi i casi in cui questa operazione è obbligatoria e quelli in cui è opzionale, esponendo le relative motivazioni.

Il costruttore della classe `OutputSection` si occupa dell'inizializzazione del comando alternativo; le guardie però dovrebbero essere create dal programmatore e aggiunte alla *Output Section* tramite il metodo `addOutputGuard`. Anche l'output stream dovrebbe essere creato dal programmatore. Per nascondere tutti questi dettagli si è deciso di creare un piccolo wrapper che consentisse di costruire una *Output Section* in modo più dichiarativo. Il wrapper è stato implementato tramite la classe `Collect` e consente di esprimere un output stream e tutte le politiche ad esso associate. La sua interfaccia è contenuta nel paragrafo 3.7.15.

Grazie a questo wrapper la *Output Section* viene costruita in modo molto semplice come composizione di più costrutti `Collect`, analogamente a quanto avviene per la *Input Section* con i costrutti `Emit` e `EmitAND`.

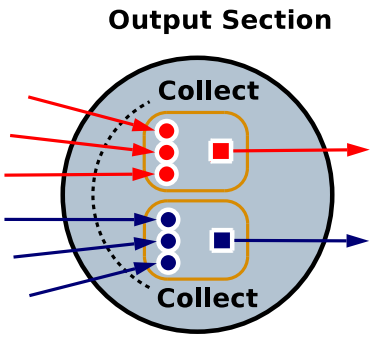

Figura 3.5: Output Section come composizione di costrutti `Collect`

3.7 Interfacce delle classi principali

3.7.1 Classe InputStreamData

```
class InputStreamData
{
protected:
    /// ID del processo mittente
    int m_senderID;
    /// ID del processo ricevente
    int m_receiverID;
    /// Maschera di bit per indicare le distribuzioni abilitate
    unsigned m_maskDistrs;
    /// Maschera di bit che indica la distribuzione di default
    Assist::Distr::distrMask m_defaultDistr;
    /// ID del canale di input
    MammuT::ChannelIDRange m_ch;
    /* Range di id per i canali di output per
     * la distribuzione scatter */
    MammuT::ChannelIDRange m_chRangeScatter;
    /* Range di id per i canali di output
     * delle altre distribuzioni */
    MammuT::ChannelIDRange m_chRangeDistr;
    /// Range di id dei processi del parmod
    MammuT::ProcIDRange m_vpmRange;
    /* Posizione del bit che indica
     * l'array di canali per la scatter */
    int m_streamIdScatter;
    /* Posizione del bit che indica l'array
     * di canali per le altre distribution */
    int m_streamIdOtherD;
    /* Maschera con un bit a 1 che indica
     * l'array di canali per la scatter */
    int m_chScatterMask;
    /* Maschera con un bit a 1 che indica l'array
     * di canali per le altre distribution
    int m_chOtherDMask;
```

```
public:
    /**
     * @brief Costruttore.
     */
    InputStreamData() {}

    /**
     * @brief Metodo per configurare i dati dell'input stream.
     *
     * Si abilitano le distribuzioni richieste e si allocano
     * gli id per i canali necessari per le comunicazioni
     * con i VPM.
     *
     * @param g grafo assist sul quale aggiungere il range
     *          di ID di canali
     * @param isData descrittore della input section
     * @param sender id del processo mittente
     * @param maskDistrs maschera di bit contenente le
     *                   distribuzioni da abilitare
     * @param defaultDistr maschera di bit contenente la
     *                     distribuzione da abilitare di default
     */
    void setInputStream(AssistGraph *g, ISData &isData, int sender,
                        unsigned maskDistrs,
                        Assist::Distr::distrMask defaultDistr =
                            (Assist::Distr::distrMask) -1);

    /**
     * @brief Restituisce il range di id di canali IS-VPM
     *        utilizzato per un determinato tipo di distribuzione.
     *
     * @param distrMask tipo di distribuzione cui si e' interessati
     *
     * @return range di id di canale
     */
    MammuT::ChannelIDRange getChDistr(int distrMask) const;
```

```
/**
 * @brief Restituisce il range di ID dei VPM
 *        connessi alla Input Section.
 */
MammuT::ProcIDRange vpmRange() const;

/**
 * @brief Restituisce il range di ID
 *        relativo al canale di input.
 */
MammuT::ChannelIDRange inputChRange() const;

/**
 * @brief Verifica se un determinato tipo
 *        di distribuzione e' abilitato.
 *
 * @param distrMask tipo di distribuzione
 *                  cui si e' interessati
 *
 * @return true se la distribuzione indicata
 *         e' abilitata, false altrimenti
 */
bool enabled(int distrMask) const;

/**
 * @brief Restituisce la maschera relativa
 *        al tipo di distribuzione di default.
 */
Assist::Distr::distrMask defaultDistr() const;

/**
 * @brief Restituisce la maschera di bit che
 *        indica i canali utilizzati per una certa
 *        distribuzione.
 *
 * @param distrMask tipo di distribuzione
 *                  cui si e' interessati
 *
```

```
     * @return maschera di bit che indica i canali utilizzati
     */
    int getMask(int distrMask) const;

    /**
     * @brief Restituisce l'id di una determinata distribuzione
     *        (posizione del bit nella relativa maschera di bit).
     *
     * @param distrMask tipo di distribuzione
     *                  cui si e' interessati
     *
     * @return posizione del bit relativo al canale usato
     * dalla distribuzione distrMask, nella maschera di bit
     */
    int getStreamId(int distrMask) const;

    /**
     * @brief Restituisce l'id del processo mittente.
     */
    int getSender() const;
};
```

3.7.2 Classe InputStream

```
template <class T_chIn, int t_outDegree = T_chIn::degree,
          imp_type_t t_impl = T_chIn::impl>
class InputStream
{
public:
    typedef T_chIn ChType;
    typedef MammuT::ChannelArraySPE<T_chIn> RRSPEchArray;
    typedef MammuT::Channels::Spe_channel<MammuT::ChRole::SENDER,
                                          typename T_chIn::ItemType,
                                          t_outDegree, 0, t_impl> SRSPEch;
    typedef MammuT::ChannelArraySPE<SRSPEch> SRSPEchArray;
```

```
    /* distribuzioni */
    typedef OnDemand<SRSPEchArray> OnDemandDistr;
    typedef Multicast<SRSPEchArray> MulticastDistr;
    typedef Scheduled<SRSPEchArray> ScheduledDistr;
    typedef Scatter<SRSPEchArray> ScatterDistr;

protected:
    /// Descrittore dell'input stream
    InputStreamData m_isd;
    /// Canale in ingresso che rappresenta lo stream di input
    RRSPEchArray *m_chin;
    /// Array di canali verso i VP per la distribuzione scatter
    SRSPEchArray *m_choutScatt;
    /// Array di canali verso i VP per le altre distribuzioni
    SRSPEchArray *m_choutOther;
    /// Puntatore all'eventuale distribuzione on demand
    OnDemandDistr *m_ondemD;
    /// Puntatore all'eventuale distribuzione multicast
    MulticastDistr *m_mcastD;
    /// Puntatore all'eventuale distribuzione scheduled
    ScheduledDistr *m_schedD;
    /// Puntatore all'eventuale distribuzione scatter
    ScatterDistr *m_scattD;
    /// Distribuzione corrente
    Distribution *m_currDistr;
    /* Maschera di bit che identifica il canale
     * utilizzato dalla distribution */
    int m_currChMask;

public:
    /**
     * @brief Costruttore.
     *
     * @param isd descrittore dell'input stream
     * @param is puntatore alla input section
     * @param com puntatore al comunicatore
     * @param inItemsNum numero di items da cui sono
     *                   composti i messaggi in input
```

```
 */
InputStream(const InputStreamData &isd, InputSection *is,
            MammuT::Stcom *com, unsigned inItemsNum);

/**
 * @brief Restituisce un puntatore al canale di input.
 */
RRSPEchArray *inputChannel() const;

/**
 * @brief Cambia la distribuzione corrente.
 *
 * @param currDistr tipo della nuova distrbuzione
 */
void setCurrentDistr(Assist::Distr::distrMask currDistr);

/**
 * @brief Restituisce un puntatore
 *        alla distribuzione corrente.
 */
Distribution *currentDistr() const;

/**
 * @brief Restituisce il valore della
 *        maschera dei bit di canale corrente.
 */
int currentChMask() const;

/**
 * @brief Restituisce un puntatore
 *        alla distribuzione ondemand.
 */
OnDemandDistr *ondemand() const;

/**
 * @brief Restituisce un puntatore
 *        alla distribuzione multicast.
 */
```

```
    MulticastDistr *multicast() const;

    /**
     * @brief Restituisce un puntatore
     *        alla distribuzione scheduled.
     */
    ScheduledDistr *scheduled() const;

    /**
     * @brief Restituisce un puntatore
     *        alla distribuzione scatter.
     */
    ScatterDistr *scatter() const;
};
```

3.7.3 Classe GuardDistr

```
class GuardDistr : public MammuT::Guard
{
public:
    /**
     * @brief Costruttore.
     *
     * @param priority priorita' da assegnare alla guardia
     */
    GuardDistr(int priority);

    /**
     * @brief Effettua la receive sui canali in ingresso.
     *
     * @return 0 in caso di successo, -1 altrimenti
     */
    virtual int receive();

    /**
```

```
     * @brief Distribuisce gli ultimi dati ricevuti
     *        sui canali in ingresso.
     */
    virtual void distribute();

    /**
     * @brief Setta la maschera di bit da
     *        inviare come codice operativo.
     *
     * @param opcMask la nuova maschera di bit
     */
    virtual void setOpcMask(int opcMask);

    /**
     * @brief Restituisce la maschera di bit corrispondente
     *        agli input stream gestiti dalla guardia
     *
     * @return la maschera di bit
     */
    virtual int istreamMask() const;

    /**
     * @brief Setta il tipo della distribuzione
     *        associata a un input stream.
     *
     * @param d tipo della nuova distribuzione
     * @param streamIdx indice dello stream di input
     */
    virtual void setDistribution(Assist::Distr::distrMask d,
                                 unsigned streamIdx = 0);
};
```

3.7.4 Classe ISData

```
class ISData
```

```
{
protected:
    /// Range di ID per i canali dedicati ai codici operativi
    MammuT::ChannelIDRange m_chopc;
    /// Range di ID per il processo allocato alla Input Section
    MammuT::ProcIDRange m_isProc;
    /// Range di ID dei VPM connessi alla Input Section
    MammuT::ProcIDRange m_vpms;
    /* Contatore utilizzato per l'assegnamento
     * delle maschere degli input stream */
    int m_istreamBitCount;
    /// Numero di guardie da inserire sul comando alternativo
    unsigned m_nguards;

public:
    /**
     * @brief Costruttore.
     */
    ISData();

    /**
     * @brief Metodo per configurare i dati della Input Section.
     *
     * A partire dal range di ID dei VPM, viene allocato
     * un range diID per i canali dedicati ai codici operativi.
     *
     * @param g grafo Assist sul quale aggiungere
     *                il range di ID di canali
     * @param vpmRange range di ID dei VPM da connettere
     *                 alla Input Section
     */
    void setVpmRange(AssistGraph *g,
                     const MammuT::ProcIDRange &vpmRange);

    /**
     * @brief Restituisce l'ID del processo
     *        allocato alla Input Section.
     */
```

```
unsigned procID() const;

/**
 * @brief Restituisce il range del processo
 *        allocato alla Input Section.
 */
MammuT::ProcIDRange procRange() const;

/**
 * @brief Restituisce il range di ID dei canali dedicati
 *        ai codici operativi.
 */
MammuT::ChannelIDRange channelOpcRange() const;

/**
 * @brief Restituisce il range di ID dei VPM
 *        connessi alla Input Section
 */
MammuT::ProcIDRange vpmRange() const;

/**
 * @brief Alloca un id per un nuovo input stream
 *        e ne restituisce il valore.
 */
int newStreamId();

/**
 * @brief Restituisce il numero di bit gia' allocati
 *        per identificare i canali verso i VPM.
 */
int getStreamBitCounter() const;

/**
 * @brief Incrementa di 1 il numero di guardie presenti
 *        nel comando alternativo.
 */
void addGuard();
```

```
    /**
     * @brief Restituisce il numero di guardie presenti
     *        nel comando alternativo.
     */
    unsigned nguards() const;
};
```

3.7.5 Classe Input Section

```
class InputSection
{
protected:
    /// Descrittore della Input Section
    ISData m_isData;
    /* Comando alternativo per il controllo
     * del nondeterminismo in ingresso */
    MammuT::AltCmd *m_cmd;
    /// Array di canali dedicati ai codici operativi
    Assist::Types::SRChArrayOpc m_chopc;
    /* Puntatore all'area di memoria
     * per la send del codice operativo */
    Assist::Types::Opc *m_opc;
    /// Tag dell'ultima send del codice operativo effettuata
    int m_waitTag;
    /* Puntatore all'ultimo canale utilizzato
     * per la send del codice operativo */
    Assist::Types::SRChArrayOpc::ChType *m_lastChOpc;
    /* Array di booleani per marcare l'invio
     * del codice operativo verso i VPM */
    bool *m_sentOpc;
    /* Valore del campo customID nel prossimo codice
     * operativo da inviare */
    int m_nextOpcID;

public:
```

```
/**
 * @brief Costruttore.
 *
 * @param isData descrittore della input section
 * @param com puntatore al comunicatore
 * @param guardInNum numero di guardie in
 *                   ingresso alla input section
 */
InputSection(const ISData &isData, MammuT::Stcom *com);

/**
 * @brief Aggiunge una guardia in ingresso alla input section.
 *
 * La guardia va creata fuori dalla input section e
 * aggiunta tramite questo metodo.
 *
 * @param gin nuova guardia in ingresso
 */
void addInputGuard(MammuT::Guard *gin);

/**
 * @brief Effettua un passo di distribuzione.
 *
 * Valuta il comando alternativo in ingresso scegliendo
 * una guardia in modo nondeterministico, ricevendo dai
 * suoi canali e distribuendo gli elementi ricevuti ai VP
 * secondo le distribuzioni ad essi associate.
 *
 * @return l'indice della guardia valutata
 * dal comando alternativo
 */
int go();

/**
 * @brief Comunica il codice operativo a uno dei VP.
 *
 * @param vpmID indice del canale su cui spedire
 *              il codice operativo
```

```
     * @param opcMask maschera da inviare nel codice operativo
     */
    void sendCop(int vpmID, int opcMask);

    /**
     * @brief Imposta il valore del campo customID per il
     *        codice operativo da inviare nel passo di
     *        distribuzione corrente. Il valore e' valido
     *        solamente per un passo di distribuzione
     *        (una chiamata al metodo go).
     *
     * @param customID valore da assegnare al campo customID
     *                 del codice operativo
     */
    void setOpcID(int customID);

    /**
     * @brief Funzione di pre-elaborazione della input section.
     *
     * Viene chiamata dopo la receive da un input stream ma
     * prima di distribuire il valore ricevuto.
     *
     * @param guardID id della guardia scelta
     * dal comando alternativo
     */
    virtual void preComputation(int guardID);
};
```

3.7.6 Classe Emit

```
template <class T_chIn>
class Emit
{
public:
    typedef GuardDistrOne<T_chIn> GuardISType;
```

```
    typedef typename T_chIn::ItemType ItemType;

protected:
    /// Guardia collegata allo stream di input
    GuardISType m_guard;

public:
    /**
     * @brief Costruttore.
     *
     * Crea uno stream in input, e lo aggiunge alla input section.
     *
     * @param is puntatore alla input section
     * @param isd descrittore dell'input stream che si vuole
     *            aggiungere alla input section
     * @param com puntatore al comunicatore
     * @param inItemsNum numero di elementi che compongono
     *                   i messaggi in input
     */
    Emit(InputSection *is, const InputStreamData &isd,
         MammuT::Stcom *com, unsigned inItemsNum = 1);

    /**
     * @brief Costruttore.
     *
     * Crea uno stream in input, e lo aggiunge
     * ad uno stream in and.
     *
     * @param isd descrittore dell'input stream che si
     *            vuole aggiungere alla input section
     * @param com puntatore al comunicatore
     * @param inItemsNum numero di elementi che compongono
     *                   i messaggi in input
     */
    Emit(EmitAND *emitAnd, const InputStreamData &isd,
         MammuT::Stcom *com, unsigned inItemsNum = 1);

    /**
```

```
     * @brief Restituisce il puntatore alla guardia.
     */
    GuardISType *guard();

    /**
     * @brief Restituisce l'id della guardia.
     */
    int id() const;

    /**
     * @brief Abilita lo stream.
     */
    void enable();

    /**
     * @brief Disabilita lo stream.
     */
    void disable();

    /**
     * @brief Restituisce l'ultimo messaggio ricevuto sullo stream.
     */
    ItemType *msg() const;
};
```

3.7.7 Classe EmitAND

```
class EmitAND
{
public:
    typedef GuardDistrMulti GuardISType;

protected:
    /* Guardia collegata agli stream di input
     * che si stanno dichiarando */
```

```
    GuardISType m_guard;
    /// Puntatore alla input section
    InputSection *m_is;

public:
    /**
     * @brief Costruttore.
     *
     * Crea la guardia e la aggiunge al comando
     * alternativo della input section.
     *
     * @param is puntatore alla input section
     * @param inputNum numero di input da mettere in AND
     */
    EmitAND(InputSection *is, int inputNum);

    /**
     * @brief Restituisce l'input section associata allo stream.
     */
    InputSection *is() const;

    /**
     * @brief Restituisce il puntatore alla guardia multipla.
     */
    GuardISType *guard();

    /**
     * @brief Restituisce l'id della guardia.
     */
    int id() const;

    /**
     * @brief Abilita lo stream.
     */
    void enable();

    /**
     * @brief Disabilita lo stream.
```

```
     */
    void disable();
};
```

3.7.8 Classe VPSection

```
class VPSection
{
public:
    typedef ChWrapper<Assist::Types::RRChArrayOpc::ChType>
        ChWrapperOpc;

protected:
    /// Array di puntatori a stream in input al vpm corrente
    ChWrapperBase **m_istreams;
    /// Dimensione dell'array m_istreams
    unsigned m_nstream;
    /// Puntatore allo stream su cui ricevere il codice operativo
    ChWrapperOpc *m_opcstream;
    /// Array di maschere di bit, una per ogni stream di input
    int *m_istreamMasks;

public:
    /**
     * @brief Costruttore.
     *
     * @param isd descrittore della input section
     * @param com puntatore al comunicatore
     */
    VPSection(const Assist::ISData &isd, MammuT::Stcom *com);

    /**
     * @brief Lancia l'attivazione del VPM.
     *
     * Effettua una receive sul canale dei codici operativi.
```

```
     * Utilizzando la maschera di bit contenuta nel codice
     * operativo effettua le receive sui canali dati abilitati.
     *
     * @return il codice operativo ricevuto
     */
    Assist::Types::Opc activate();

    /**
     * @brief Aggiunge un wrapper di canale.
     *
     * @param chw wrapper di canale
     * @param index indice nell'array di stream in ingresso,
     *        in cui porre il puntatore
     */
    void addInputStream(ChWrapperBase *chw, unsigned index);
};
```

3.7.9 Classe VPMInputStream

```
template <class T_ch>
class VPMInputStream
{
protected:
    /* Puntatore al wrapper per lo stream in input
     * (distribuzioni non SCATTER) */
    ChWrapper<T_ch> *m_chwOther;
    /* Puntatore al wrapper per lo stream in input
     *(distribuzioni SCATTER) */
    ChWrapper<T_ch> *m_chwScatt;

public:
    typedef typename T_ch::ItemType ItemType;

    /**
     * @brief Costruttore.
```

```
     *
     * @param vps puntatore alla VP Section su cui
     *            registrare lo stream
     * @param isd descrittore dello stream in ingresso
     * @param numItems numero di items di cui sono
     *                 composti i messaggi in input
     *                 (corrisponde al numero utilizzato
     *                 per i canali dell'input stream
     *                 corrispondente sull'Input Section)
     */
    VPMInputStream(VPSection *vps,
                   const Assist::InputStreamData &isd,
                   MammuT::Stcom *com, unsigned numItems = 1);

    /**
     * @brief Restruisce il numero di items di cui sono
     *        composti i messaggi ricevuti sullo stream
     *        (per distribuzioni non SCATTER).
     */
    unsigned numItems() const;

    /**
     * @brief Restituisce il puntatore all'ultimo messaggio
     *        ricevuto sullo stream (per distribuzioni non SCATTER).
     */
    ItemType *msg() const;

    /**
     * @brief Restruisce il numero di items di cui sono
     *        composti i messaggi ricevuti sullo stream
     *        (per distribuzioni SCATTER).
     */
    unsigned numItemsScatter() const;

    /**
     * @brief Restituisce il puntatore all'ultimo messaggio
     *        ricevuto sullo stream (per distribuzioni SCATTER).
     */
```

```
    ItemType *msgScatter() const;
};
```

3.7.10 Classe VPMOutputStream

```
template <class T_ch>
class VPMOutputStream
{
protected:
    /// Puntatore al canale di comunicazione tra VPM e OS
    T_ch *m_chout;
    /// Wait tag per la send del messaggio in output
    int m_waitTag;
    /// Puntatore al messaggio da spedire
    typename T_ch::ItemType *m_msg;

public:
    /**
     * @brief Costruttore.
     *
     * @param ostreamData descrittore dell'output stream
     * @param com puntatore al comunicatore
     * @param numItems numero di items di cui sono composti
     *                 i messaggi in transito sul canale m_chout
     */
    VPMOutputStream(const Assist::OutputStreamData &ostreamData,
                    MammuT::Stcom *com, unsigned numItems = 1);

    /**
     * @brief Effettua la wait sul canale in uscita.
     */
    void wait();

    /**
     * @brief Effettua la send sul canale in uscita.
```

```
     *
     * @param msg puntatore al messaggio da spedire
     */
    void send(typename T_ch::ItemType *msg);

    /**
     * @brief Restituisce il numero di items di cui sono
     *        composti i messaggi inviati sullo stream.
     */
    unsigned chNumItems() const;

    /**
     * @brief Alloca un'area di memoria per l'invio sullo
     *        stream di output.
     *
     * @param alignment allineamento della dimensione del
     *                  messaggio voluto
     *
     * @return il puntatore all'area di memoria allocata
     */
     typename T_ch::ItemType
         *allocMsg(alignment_type_t alignment = ALIGN_128);
};
```

3.7.11 Classe OutputStreamData

```
class OutputStreamData
{
protected:
    /// ID del processo mittente
    int m_senderID;
    /// ID del processo ricevente
    int m_receiverID;
    /// ID del canale di output
    MammuT::ChannelIDRange m_chOut;
```

```
    /// Range di id per i canali di input
    MammuT::ChannelIDRange m_chRangeIn;
    /// Range di id dei VPM del parmod
    MammuT::ProcIDRange m_vpmRange;
    /// ID associato all'output stream
    int m_id;
    /** Maschera di bit per indicare le policy di
     *  collection abilitate */
    unsigned m_maskCollect;
    /** Maschera di bit che indica la policy di collection
     *  di default */
    Assist::Collections::policy m_defaultCollect;

public:
    /**
     * @brief Costruttore.
     */
    OutputStreamData() {}

    /**
     * @brief Metodo per configurare i dati dell'output stream.
     *
     * @param g grafo assist sul quale aggiungere il range
     *          di ID di canali
     * @param osData descrittore della output section
     * @param receiver id del processo destinatario
     * @param maskCollect maschera di bit contenente le policy
     *                    di collection da abilitare
     * @param defaultCollect maschera di bit contenente la
     *                       policy di collection da abilitare
     *                       di default; il valore NONECOLLECT
     *                       indica che come policy di default
     *                       si vuole quella corrispondente al
     *                       primo bit meno significativo
     *                       trovato a 1 nella maschera delle
     *                       policy da abilitare (maskCollect)
     */
    void setOutputStream(AssistGraph *g, OSData &osData,
```

```
                    int receiver, unsigned maskCollect,
                    Assist::Collections::policy defaultCollect =
                       (Assist::Collections::policy) -1);

/**
 * @brief Restituisce il range di ID dei VPM connessi alla
 *        Output Section.
 */
MammuT::ProcIDRange vpmRange() const;

/**
 * @brief Restituisce il range di ID relativo al canale
 *        di output.
 */
MammuT::ChannelIDRange outputChRange() const;

/**
 * @brief Restituisce il range di ID relativo ai canali in
 *        input sull'OS (utilizzati per ricevere i
 *        risultati dai VP).
 */
MammuT::ChannelIDRange inputChRange() const;

/**
 * @brief Verifica se una determinata policy di collection
 *        e' abilitata.
 *
 * @param collectMask policy di collection cui si e'
 *                    interessati
 *
 * @return true se la policy indicata e' abilitata,
 *         false altrimenti
 */
bool enabled(int collectMask) const;

/**
 * @brief Restituisce la maschera relativa alla policy di
 *        collection di default.
```

```
     */
    Assist::Collections::policy defaultCollect() const;

    /**
     * @brief Restituisce l'id dell'output stream.
     */
    int id() const;

    /**
     * @brief Restituisce l'id del processo destinatario.
     */
    int getReceiver() const;
};
```

3.7.12 Classe OutputStream

```
template <class T_chOut, class T_chIn =
    MammuT::Channels::Spe_channel<MammuT::ChRole::RECEIVER,
                                  typename T_chOut::ItemType,
                                  T_chOut::degree, 0,
                                  T_chOut::impl> >
class OutputStream
{
public:
    typedef T_chOut ChOutType;
    typedef T_chIn ChInType;
    typedef MammuT::ChannelArraySPE<T_chOut> SRSPEchArray;
    typedef MammuT::ChannelArraySPE<T_chIn> RRSPEchArray;
    typedef typename T_chIn::ItemType ItemType;

protected:
    /// Canale in uscita che rappresenta lo stream di output
    SRSPEchArray m_chout;
    /// Array di canali dai VP per la collezione
    RRSPEchArray m_chinArray;
```

```
    /// ID dello stream
    int m_id;
    /// Wait tag per la send del messaggio in output
    int m_waitTag;
    /// Puntatore al messaggio da spedire
    ItemType *m_msg;
    /** Puntatore al canale contente l'ultimo msg
     *  (utile per FROM ANY e FORWARD) */
    ChInType *m_lastCh;

public:
    /**
     * @brief Costruttore
     *
     * @param osd descrittore dell'output stream
     * @param os puntatore alla output section
     * @param com puntatore al comunicatore
     * @param inItemsNum numero di items da cui sono composti
     *                   i messaggi in input
     * @param outItemsNum numero di items da cui sono composti
     *                    i messaggi in output
     */
    OutputStream(const OutputStreamData &osd,
                 OutputSection *os, MammuT::Stcom *com,
                 unsigned inItemsNum = 1,
                 unsigned outItemsNum = 1);

    /**
     * @brief Restituisce l'id dell'output stream.
     */
    int id() const;

    /**
     * @brief Restituisce un puntatore all'array di canali
     *        in input.
     */
    RRSPEchArray *chArray();
```

```
/**
 * @brief Restituisce un puntatore al canale di output.
 */
ChOutType *chOut();

/**
 * @brief Consente di settare il puntatore al prossimo
 *        messaggio da spedire sul canale di output.
 *
 * @param msg puntatore al prossimo messaggio da spedire
 */
void setMsg(ItemType *msg);

/**
 * @brief Consente di attendere che l'ultima send
 *        effettuata sia stata completata, in modo da
 *        poter aggiornare il valore del messaggio.
 */
void wait();

/**
 * @brief Invia un messaggio sul canale di output.
 *        Il messaggio da spedire va settato col metodo
 *        setMsg.
 */
void send();

/**
 * @brief Restituisce il puntatore al prossimo messaggio
 *        da spedire.
 */
ItemType *msg() const;

/**
 * @brief Consente di settare l'ultimo canale dello stream
 *        da cui si e' effettuata una receive.
 *
 * Metodo utile nel caso in cui si utilizzi una politica
```

```
     * di collect from all o forward.
     *
     * @param lastCh puntatore al canale
     */
    void setLastCh(ChInType *lastCh);

    /**
     * @brief Restituisce un puntatore all'ultimo canale dello
     *        stream da cui si e' effettuata una receive.
     */
    ChInType *lastCh() const;
};
```

3.7.13 Classe OSData

```
class OSData
{
protected:
    /// Range di ID per il processo allocato alla Output Section
    MammuT::ProcIDRange m_osProc;
    /// Range di ID dei VPM connessi alla Output Section
    MammuT::ProcIDRange m_vpms;
    /** Contatore utilizzato per l'assegnamento degli id
     *  degli output stream */
    unsigned m_ostreamCounter;
    /// Numero di guardie da inserire sul comando alternativo
    unsigned m_nguards;

public:
    /**
     * @brief Costruttore.
     */
    OSData();

    /**
```

```
     * @brief Metodo per configurare i dati della
     *        Output Section.
     *
     * @param g grafo Assist sul quale aggiungere il range
     *          di ID di canali
     * @param vpmRange range di ID dei VPM da connettere
     *        alla Input Section
     */
    void setVpmRange(AssistGraph *g,
                     const MammuT::ProcIDRange &vpmRange);

    /**
     * @brief Restituisce l'ID del processo allocato
     *        alla Output Section.
     */
    unsigned procID() const;

    /**
     * @brief Restituisce il range del processo allocato
     *        alla Output Section.
     */
    MammuT::ProcIDRange procRange() const;

    /**
     * @brief Restituisce il range di ID dei VPM connessi
     *        alla Output Section
     */
    MammuT::ProcIDRange vpmRange() const;

    /**
     * @brief Alloca un id per un nuovo output stream e ne
     *        restituisce il valore.
     */
    unsigned newStreamId();

    /**
     * @brief Incrementa il numero di guardie presenti
     *        nel comando alternativo.
```

```
     *
     * @param nguards numero di guardie da aggiungere
     */
    void addGuards(unsigned nguards);

    /**
     * @brief Restituisce il numero di guardie presenti nel
     *        comando alternativo.
     */
    unsigned nguards() const;
};
```

3.7.14 Classe OutputSection

```
class OutputSection
{
protected:
    /// Comando alternativo
    MammuT::AltCmd m_cmd;

public:
    /**
     * @brief Costruttore.
     *
     * @param osdata descrittore dell'output section
     * @param numGuards numero di guardie previste (tenere
     *                  presente che per una collect from all
     *                  o una gather si usa un'unica guardia
     *                  mentre per una collect from any o una
     *                  forward si usa un numero di guardie
     *                  pari al numero di VP)
     */
    OutputSection(const OSData &osData);

    /**
```

```
     * @brief Aggiunge una guardia in ingresso alla
     *        Output Section.
     *
     * La guardia va creata fuori dalla Output Section e
     * aggiunta tramite questo metodo.
     *
     * @param gout nuova guardia in ingresso
     */
    void addOutputGuard(MammuT::Guard *gout);

    /**
     * @brief Effettua un passo di collection.
     *
     * Valuta il comando alternativo in ingresso scegliendo
     * una guardia in modo nondeterministico, ricevendo dai
     * suoi canali (1 nel caso di collect from any o forward,
     * 1 o piu' nel caso di collect from all o gather) e
     * inviando i risultati sul canale di output.
     *
     * @return l'indice della guardia valutata dal comando
     *         alternativo
     */
    int go();

    /**
     * @brief Funzione di post-elaborazione della
     *        Output Section.
     *
     * Viene chiamata dopo aver ricevuto i risultati
     * dell'elaborazione eseguita sui VP ma prima di inviarli
     * sul canale di output.
     * Va overloadata per aggiungere una qualsiasi funzione di
     * post-elaborazione.
     *
     * @param streamID id dello stream cui appartiene la
     *                 guardia scelta dal comando alternativo
     */
    virtual void postComputation(int /*streamID*/);
```

```
};
```

3.7.15 Classe Collect

```
template <class T_ostream>
class Collect
{
public:
    typedef GuardCollectFromAny<T_ostream> GuardFromAnyType;
    typedef GuardCollectForward<T_ostream> GuardForwardType;
    typedef GuardCollectFromAll<T_ostream> GuardFromAllType;
    typedef GuardCollectGather<T_ostream> GuardGatherType;
    typedef typename T_ostream::ChInType::ItemType ItemType;

protected:
    /** Stream di output: contiene i canali in ingresso e
     *  in uscita dalla OS */
    T_ostream *m_ostream;
    /** Array di puntatori per le eventuali guardie per il
     *  from any */
    GuardFromAnyType **m_guardsFromAny;
    /** Array di puntatori per le eventuali guardie per la
     *  forward */
    GuardForwardType **m_guardsForward;
    /// Puntatore all'eventuale guardia per il from all
    GuardFromAllType *m_guardFromAll;
    /// Puntatore all'eventuale guardia per la gather
    GuardGatherType *m_guardGather;
    /// Puntatore per le eventuali guardie per il from one
    GuardFromAnyType **m_guardsFromOne;
    /// Indice dell'eventuale guardia attivata per il from one
    unsigned m_actividGuard;
    /** maschera di bit che identifica la politica di
     *  collezione corrente */
    Assist::Collections::policy m_collectPolicy;
```

```
    /// copia del descrittore
    OutputStreamData m_osd;


    /**
     * @brief Metodo di inizializzazione.
     *
     * @param os puntatore all'Output Section a cui
     *           associare la Collect
     * @param com puntatore al comunicatore
     * @param nItemsIn numero di elementi che compongono
     *                 i messaggi in input
     * @param nItemsOut numero di elementi che compongono
     *                  i messaggi in output
     */
    void init(OutputSection *os, MammuT::Stcom *com,
              unsigned nItemsIn, unsigned nItemsOut);

public:
    /**
     * @brief Costruttore.
     *
     * Crea l'output stream.
     *
     * @param os puntatore all'output section
     * @param ostreamData descrittore dell'output stream da
     *                    creare
     * @param com puntatore al comunicatore
     * @param nItemsIn numero di elementi che compongono
     *                 i messaggi in input
     * @param nItemsOut numero di elementi che compongono
     *                  i messaggi in output
     */
    Collect(OutputSection *os,
            const OutputStreamData &ostreamData,
            MammuT::Stcom *com, unsigned nItemsIn,
            unsigned nItemsOut);
```

```
/**
 * @brief Costruttore.
 *
 * Crea l'output stream. La dimensione dei messaggi in
 * output viene calcolata automaticamente a partire dalla
 * dimensione dei messaggi sui canali in input alla output
 * section.
 *
 * @param os puntatore all'output section
 * @param ostreamData descrittore dell'output stream da
 *                    creare
 * @param com puntatore al comunicatore
 * @param nItemsIn numero di elementi che compongono i
 *                 messaggi in input
 */
Collect(OutputSection *os,
        const OutputStreamData &ostreamData,
        MammuT::Stcom *com, unsigned nItemsIn = 1);

/**
 * @brief Metodo per settare la politica del costrutto
 *        Collect.
 *
 * @param policy Nuova politica
 */
void setPolicy(Assist::Collections::policy policy);

/**
 * @brief Metodo d'accesso per la politica corrente.
 */
Assist::Collections::policy policy();

/**
 * @brief Metodo per settare il VP da cui ricevere in
 *        caso di politica FROMONE.
 *
 * @param vpmFrom Indice del VP
 */
```

```
    void setFrom(unsigned vpmFrom);

    /**
     * @brief Abilita lo stream.
     */
    void enable();

    /**
     * @brief Disabilita lo stream.
     */
    void disable();

    /**
     * @brief Restituisce l'ultimo messaggio ricevuto sullo
     *        stream.
     */
    ItemType *msg() const;

    /**
     * @brief Restituisce l'ultimo messaggio ricevuto da parte
     *        di un determinato VP.
     *
     * @param index indice del VP cui si e' interessati
     */
    ItemType *msg(unsigned index) const;

    /**
     * @brief Restituisce un puntatore all'output stream.
     */
    T_ostream *ostream();

    /**
     * @brief Restituisce l'id dell'output stream.
     */
    int id() const;
};
```

Capitolo 4

Utilizzare la libreria: un esempio

Nel capitolo precedente la libreria è stata analizzata nel dettaglio, sia per quanto riguarda la semantica, sia per quanto riguarda l'implementazione. In questo capitolo verranno utilizzate e valutate le funzionalità della libreria con l'ausilio di un esempio reale.

Si consideri questo semplice programma:

```
int a[ARRAYDIM];
int result[ARRAYDIM];
int counter = 0;
int threshold = ... ;
...

for (int i = 0; i < ARRAYDIM; i++) {
    if (a[i] < threshold) {
        result[i] = 1;
        counter++;
    } else {
        result[i] = 0;
    }
}
```

Dato un array `a` di `ARRAYDIM` interi, il programma genera un array `result` in cui l'i-esimo elemento è 1 se `a[i] < threshold`, e 0 altrimenti. Vengono inoltre contati gli elementi che soddisfano tale condizione. Questo è un tipico caso di *map - reduce*, e l'esempio che verrà analizzato sarà un programma parallelo che eseguirà questa elaborazione su stream [1]. Il programma parallelo gestisce uno stream di array `a`, e uno stream di valori soglia `threshold`, prodotti da un processo *generator*. Quando la *Input Section* riceve un elemento dello stream `a`, effettua una *scatter* dell'array `a`. Quando la *Input Section* riceve un elemento dello stream threshold, distribuisce il dato in *multicast*.

Quando il VP riceve la partizione dell'array `a`, aggiorna la partizione corrente. Il VP effettua il calcolo sulla partizione corrente ogni volta che riceve un valore `threshold`. I VP calcolano la loro partizione dell'array `result`, ed effettuano la *reduce* in locale, generando un proprio `counter`. Questi risultati costituiscono l'output di ogni VP. La *Output Section* effettua una *gather* che ricompone l'array `result`, ed effettua una sommatoria dei `counter` parziali generati dai VP, completando l'operazione di *reduce* globale. La *Output Section* invia i risultati su due output stream, il processo `printer` riceve i risultati ed effettua i controlli di correttezza della computazione.

Il capitolo analizza l'applicazione sotto tre punti di vista. Viene innanzitutto illustrato il codice corrispondente in ASSIST-CL e la sua traduzione concettuale in LC. Infine viene presentata un'applicazione scritta utilizzando la libreria CellAssist, analizzando la corrispondenza dei costrutti nei diversi livelli e l'impatto del design della libreria CellAssist nei confronti di un eventuale compilatore.

4.1 Codice ASSIST-CL

In questa sezione viene illustrato il listato ASSIST dell'applicazione.

[1] Questo esempio è disponibile nei sorgenti, in `assist/samples/mapreduce`.

```
#define ARRAYDIM ...
#define TIMES ...

generic main()
{
    stream int[ARRAYDIM] a;
    stream int threshold;
    stream int[ARRAYDIM] result;
    stream int counter;

    generator(output_stream a, threshold);
    mapreduce(input_stream a, threshold
              output_stream result, counter);
    printer(input_stream result, counter);
}


generator(output_stream int a[ARRAYDIM], int threshold)
$c++{
    ...
    assist_out(a, ...);

    for (int i = 0; i < TIMES; i++) {
        ...
        assist_out(threshold, ...);
    }
}c++$


parmod mapreduce(input_stream int a[ARRAYDIM], int threshold
                 output_stream int result[ARRAYDIM], int counter)
{
    topology array Pv[i:N];

    attribute int iteration;
    attribure int a[ARRAYDIM] scatter onto Pv;
```

```
init {
    iteration = 0;
}

do input_section {
    a_guard: on, , a {
        distribution a scatter to Pv;
    }

    threshold_guard: on, , threshold {
        distribution threshold broadcast to Pv;
        operation {
            iteration++;
        } <use {iteration}>
    }
} while(iteration < TIMES)


virtual_processors {
    elab_threshold(in threshold_guard out result, counter) {
        VP i {
            vpCode(in a[i], threshold
                   output_stream result, counter);
        }
    }
}


output_section {
    collects counter from ALL Pv[i] {
        int _partCounter;
        int _counter;
        AST_FOR_EACH(_partCounter) {
            _counter += _partCounter;
        }

        assist_out(counter, _partCounter);
    }<>;
```

```
        collects result from ALL Pv[i] {
            int _partResult;
            int _result[ARRAYDIM];
            AST_FOR_EACH(_partResult) {
                _result[i] = _partResult;
            }

            assist_out(result, _result);
        }<>;
    }
}


printer(input_stream int result[ARRAYDIM], int counter)
$c++{
    ...
}c++$


proc vpCode(in int a, int threshold
            output_stream int result, int counter)
$c++{
    int _result = 0;
    int _counter = 0;

    if (a < threshold) {
        _result = 1;
        _counter = 1;
    }

    assist_out(result, _result);
    assist_out(counter, _counter);
}c++$
```

Le direttive `#define` introducono alcune costanti caratteristiche dell'applicazione.

- `ARRAYDIM`: dimensione dell'array `a`;
- `TIMES`: lunghezza dello stream `threshold`;

Il blocco `generic` definisce la struttura dell'applicazione come combinazione di stream, processi sequenziali e *parmod*. Il modulo sequenziale `generator` produce gli stream `a` e `threshold` in ingresso al *parmod* `mapreduce`, che a sua volta genera gli output sugli stream `result` e `counter`, in input al sequenziale `printer`.

Il *parmod* `mapreduce` è definito attraverso i tre blocchi `input_section`, `virtual_processors` e `output_section`. Nel blocco `input_section` è presente una guardia `a_guard` che distribuisce gli elementi ricevuti sullo stream `a` tramite una `scatter`. La guardia `threshold_guard` distribuisce gli elementi ricevuti sullo stream `threshold` in *broadcast* ai VP del *parmod*. Nella sezione `virtual_processors` viene espressa la computazione effettuata dal i-esimo VP. E' da sottolineare che viene espressa la computazione a grana più fine in assoluto. Il motivo è che si presuppone l'esistenza di un compilatore che stabilisca il mapping tra processi reali e risorse fisiche. L'elaborazione del VP è contenuta in una `proc` che esegue il calcolo su una sola posizione dell'array. Nella `output_section` vengono raccolti i dati relativi ai due stream, effettuando la sommatoria dei contatori parziali, e la `gather` della matrice `result`. Queste due operazioni corrispondono ai due costrutti `collects from ALL`.

4.2 Codice LC

E' possibile scrivere un programma LC corrispondente al programma ASSIST descritto. Questo traduzione potrebbe essere effettuata da un ipotetico compilatore. In LC non esiste l'astrazione dei processori virtuali, dunque il passo di traduzione comprende l'elaborazione di uno schema di mapping dei processori virtuali in processi LC reali. Ipotizzando che lo schema stabilito

dal passo di compilazione accorpi tutti i processori virtuali su due processori, il programma LC ottenuto è il seguente.

```
#define ARRAYDIM ...
#define N 2
#define KK ...
#define PARTDIM (ARRAYDIM / N)
#define TIMES ...

#define MASK_A          0x0001
#define MASK_THRESHOLD 0x0002

typedef struct {
    int mask;
    int customID;
} Opc;

parallel generator, is, w[N], os, printer;
         channel a, threshold, a_vp[N], threshold_vp[N],
                 vp_result[N], vp_counter[N], result, counter,
                 chopc[N];

generator::
    channel out a, threshold;
{
    int _a[ARRAYDIM];
    int _threshold;

    ...
    send(a, _a);

    for (int i = 0; i < TIMES; i++) {
        ...
        send(threshold, _threshold);
    }
}
```

```
is::
    channel in a(KK), threshold(KK);
    channel out a_vp, threshold_vp, chopc;
{
    int _a[ARRAYDIM];
    int _threshold;
    int count = 0;

    Opc opc;

    for (int i = 0; i < TIMES; i++) {
        alternative {
            priority(0), true, receive(a, _a) do {
                opc.mask = MASK_A;
                multicast(chopc, opc);

                scatter(a_vp, _a);
            } or priority(0), true,
                    receive(threshold, _threshold) do {
                opc.mask = MASK_THRESHOLD;
                multicast(chopc, opc);

                multicast(threshold_vp, _threshold);
            }
        }
    }
}

w[i]::
    channel in a_vp[i](KK), threshold_vp[i](KK), chopc[i](KK);
    channel out vp_result[i], vp_counter[i];
{
    int partA[PARTDIM];
    int threshold;

    int partResult[PARTDIM];
    int counter = 0;
```

```
    Opc opc;

    channelname chVPSection[N];
    Msg msg[N];
    chVPSection[1] = a_vp[i];
    chVPSection[2] = threshold_vp[i];

    unsigned recvThresholds = 0;

    while (recvThresholds < TIMES) {
        /* parte relativa all'implementazione
         * della VP section */
        receive(chopc[i], opc);
        for (int index = 0; index < N; index++)
            if (opc.mask & (1 << index))
                receive(chVPSection[index], msg[index]);

        if (opc.mask == MASK_THRESHOLD) {
            partA = msg[1];
            threshold = msg[2];

            for (int w = 0; w < PARTDIM; w++) {
                if (partA[w] < threshold) {
                    partResult[w] = 1;
                    counter++;
                } else {
                    partResult[w] = 0;
                }
            }

            recvThresholds++;
            send(vp_result[i], partResult);
            send(vp_counter[i], counter);
        }
    }
}
```

```
os::
    channel in vp_result(KK), vp_counter(KK);
    channel out result[i], counter[i];
{
    int _result[ARRAYDIM];
    int _partResult[N][PARTDIM];

    int _counter;
    int _partCounter[N];

    for (int i = 0; i < TIMES * 2; i++) {
        _counter = 0;

        alternative {
            , true, receive(vp_result[0], _partResult[1][]),
                    ... ,
                    receive(vp_result[N - 1],
                            _partResult[N - 1][]) do {

                for (int i = 0; i < N; i++)
                    for (int j = 0; j < PARTDIM; j++)
                        _result[i * PARTDIM + j] = _partResult[i][j];

                send(result, _result);
            } or, true, receive(vp_counter[0], _partCounter[0]),
                    ... ,
                    receive(vp_counter[N - 1],
                            _partCounter[N - 1]) do {

                for (int k = 0; k < N; k++)
                    _counter += _partCounter[k];

                send(counter, _counter);
            }
        }
    }
}
```

```
printer::
    channel in result(KK), counter(KK);
{
    int _result;
    int _counter;

    for (int i = 0; i < TIMES; i++) {
        receive(result, _result);
        receive(counter, _counter);
        ...
    }
}
```

Le direttive `#define` introducono alcuni parametri caratteristici dell'applicazione, così come nel listato ASSIST. Le costanti `ARRAYDIM` e `TIMES` sono presenti anche nel codice ASSIST e hanno lo stesso significato. La costante `N` esprime il numero di "worker" stabilito dal compilatore. Dal numero di processi è possibile ricavare la dimensione delle partizioni da generare per effettuare in LC la `scatter` espressa nel codice ASSIST. La costante `KK` è utilizzata per parametrizzare il grado di asincronia dei canali dell'applicazione.

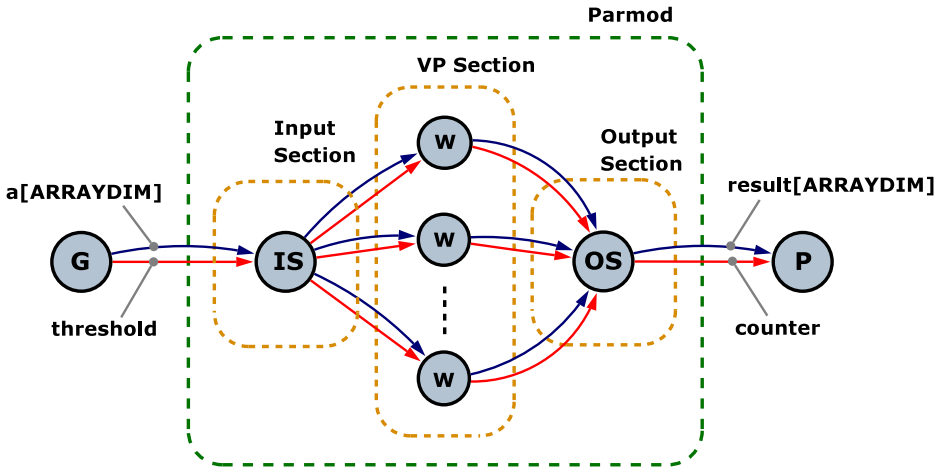

Figura 4.1: Schema del *parmod* utilizzando processi e canali LC

Il blocco `parallel` ha una funzione simile al blocco `generic` nel codice ASSIST, e permette di esprimere la serie di nomi dei processi e dei canali che costituiscono la struttura dell'applicazione. Ovviamente in LC vengono espressi processi e canali, mentre nel codice ASSIST l'applicazione è formata da stream, moduli sequenziali e *parmod*. I processi `generator` e `printer` corrispondono ai moduli sequenziali omonimi nel codice ASSIST. I processi `is` e `os` sono previsti per l'esecuzione del codice relativo alla *Input Section* e alla *Output Section* del *parmod* `mapreduce` nel codice ASSIST. I VP del *parmod* sono definiti dall'array di processi `w`.

L'array di canali `a_vp` verrà utilizzato dal processo `is` per distribuire in *scatter* gli elementi ricevuti sul canale `a`. Analogamente, l'array di canali `threshold_vp` verrà utilizzato dal processo `is` per distribuire in *multicast* i valori ricevuti sul canale `threshold`. In generale, ad ogni stream in ingresso al *parmod* in ASSIST corrisponde un canale verso il processo `is` e un array di canali per la distribuzione degli elementi ricevuti ai VP. Gli array di canali hanno dimensione `N`, e il canale i-esimo ha come destinatario il processo i-esimo dell'array di processi `w`. Anche l'array di canali `chopc` ha queste caratteristiche, ma non corrisponde a nessuno stream in ingresso al *parmod* e viene utilizzato per distribuire i codici operativi ai VP. La modalità di gestione dei codici operativi diverrà più chiara analizzando il codice del processo `is`. Le comunicazioni verso il processo `os` seguono uno schema esattamente speculare. Per ogni stream in uscita dal *parmod* è previsto un canale in uscita dal processo `os`, e un array di canali che raccoglie i dati dai VP. L'array di canali `vp_result` verrà utilizzato dal processo `os` per effettuare la *gather* degli elementi da inviare sul canale `result`, e l'array di canali `vp_counter` raccoglie i contatori parziali su cui effettuare la sommatoria, da inviare sul canale `counter`.

Il processo `is` esegue un comando alternativo, composto da una guardia per ognuno dei canali in ingresso (`a` e `threshold`). Entrambe le guardie effet-

tuano inizialmente una *multicast* del codice operativo [2]. Il valore assegnato è una costante, `MASK_A` nel caso del canale `a`, e `MASK_THRESHOLD` nel caso del canale `threshold`. Tramite queste costanti il processo `is` può comunicare ai processi `w` qual'è il canale su cui si presenterà il prossimo input. Il codice di entrambe le guardie prosegue con la distribuzione degli elementi ricevuti (*scatter* nel caso del canale `a`, *multicast* nel caso del canale `threshold`).

Il processo i-esimo dell'array `w`, riceve gli input dagli i-esimi canali degli array `a_vp` e `threshold_vp`, utilizzati dal processo `is` per la distribuzione dei dati. Oltre a questi, è presente in input anche l'i-esimo canale dell'array `chopc`, utilizzato dal processo `is` per comunicare il codice operativo. Ad ogni iterazione, ogni processo dell'array `w` effettua la `receive` dal canale `chopc[i]` ed esamina i bit attivi nella maschera associata al codice operativo; per ogni bit attivo segue una `receive` sul canale corrispondente (vedi la sezione 3.5). Il codice prosegue esaminando la maschera contenuta nel codice operativo. Nel caso sia attivato il bit espresso dalla costante `MASK_A`, la partizione dell'array `a` viene salvata in una variabile apposita. Nel caso sia attivato il bit espresso dalla costante `MASK_THRESHOLD`, viene eseguito il calcolo e i risultati vengono inviati sui canali in uscita `vp_result[i]` e `vp_counter[i]`.

Il processo `os` è più o meno speculare al processo `is`. Gli array di canali in ingresso `vp_result` e `vp_counter` permettono di ricevere i risultati parziali di tutti i processi dell'array `w`. Il processo esegue un comando alternativo con due guardie. La prima guardia è su tutti i canali appartenenti all'array `vp_result`, in and. Questa guardia effettua la *gather* del messaggio da spedire sul canale `result`. La seconda guardia è su tutti i canali appartenenti all'array `vp_counter`, sempre in and. In questo caso viene effettuata la sommatoria dei risultati parziali, e il risultato viene spedito sul canale `counter`. In generale, un costrutto `collects` con politica `from ALL` può essere tradotto tramite un'unica guardia con i canali in ingresso in and, e in caso di politica `from ANY` può essere tradotto tramite una serie di guardie, ognuno

[2] Si supponga che esistano le primitive *ondemand*, *multicast*, *scheduled* e *scatter* che distribuiscono un dato utilizzando un array di canali.

su un canale, e con lo stesso corpo.

4.3 Codice con la libreria CellAssist

Nella sezione precedente si è passati da una definizione dell'applicazione in ASSIST-CL ad una definizione in LC. In questa sezione verrà analizzato il codice del programma corrispondente in C++, utilizzando la libreria CellAssist. Come è stato possibile vedere, il linguaggio ASSIST-CL permette di esprimere in modo semplice ed elegante un'applicazione parallela, ma il linguaggio LC permette una flessibilità e un livello di personalizzazione maggiore. L'obiettivo della libreria CellAssist è quello di fornire i costrutti di ASSIST-CL, conservando al contempo la flessibilità di LC. Nel capitolo 3 è stata anticipata la struttura della libreria. Tramite un esempio sarà possibile vedere la corrispondenza tra un'applicazione scritta con la libreria CellAssist e le controparti in ASSIST-CL e LC. Sarà anche possibile evidenziare il design della libreria, in rapporto alle operazioni che un futuro compilatore dovrà effettuare per tradurre un programma scritto in ASSIST-CL in un programma scritto in C++ che fa uso della libreria.

Lo schema prevede un processo `generator`, un processo per la *Input Section*, due per i VP, uno per la *Output Section* e uno per il processo *printer* [3]. Tutti i processi dell'applicazione vengono eseguiti sugli SPE, con il PPE che esegue solo il codice relativo all'inizializzazione del supporto. Il codice di ogni processo risiede in un programma compilato separatamente. Come illustrato nel capitolo 1.6, compilare il codice dei processi SPE in programmi separati produce un beneficio diretto in termini di occupazione di memoria. La struttura della directory contente il codice dell'applicazione è la seguente:

[3] In alternativa, sarebbe stato possibile spostare sul PPE i processi *generator* e *printer* in *thread* separati.

```
.
|-- Makefile
|-- common.h
|-- mygraph.h
|-- spe
|   |-- Makefile
|   |-- generator
|   |   |-- Makefile
|   |   `-- generator.cpp
|   |-- inputsection
|   |   |-- Makefile
|   |   `-- inputsection.cpp
|   |-- outputsection
|   |   |-- Makefile
|   |   `-- outputsection.cpp
|   |-- printer
|   |   |-- Makefile
|   |   `-- printer.cpp
|   `-- worker
|       |-- Makefile
|       `-- worker.cpp
`-- test.cpp
```

4.3.1 Makefile

Il Makefile principale contiene tutte le impostazioni necessarie per la compilazione dell'eseguibile che dovrà essere eseguito sul PPE.

```
ROOTPATH := ../../..

include $(ROOTPATH)/make.defs

DIRS := spe

INCLUDE          := -I$(ROOTPATH)/mammut/include \
```

```
                    -I$(ROOTPATH)/assist/include
MAMMUT_PROCESSOR := ppu
PROGRAM          := mapreduce
IMPORTS          := $(ROOTPATH)/mammut/src/ppe/libmammut_ppe.a \
                    spe/generator/libgenerator_spe.a \
                    spe/inputsection/libinputsection_spe.a \
                    spe/worker/libworker_spe.a \
                    spe/outputsection/liboutputsection_spe.a \
                    spe/printer/libprinter_spe.a \
                    -lspe2 -lpthread

include $(ROOTPATH)/make.footer
```

Tramite la variabile `PROGRAM` viene impostato il nome dell'eseguibile da generare. La variabile `MAMMUT_PROCESSOR` in questo caso indica che si vuole generare un eseguibile per il PPE. La variabile `INCLUDE` indica le directory dove risiedono gli header della libreria MammuT e CellAssist. La variabile `IMPORTS` infine consente di passare al linker una serie di librerie statiche, tra cui la libreria MammuT per il PPE (`libmammut_ppe.a`) e tutte le librerie statiche contenenti i programmi separati per i processi; questo esempio infatti è compilato in modo da includere in un unico eseguibile il codice del PPE e il codice per gli SPE, utilizzando il meccanismo delle *SPU embed library* (vedi capitolo 1.6).

Il Makefile della directory `spe` è semplice, e serve solamente per discendere nel livello successivo delle directory. Ogni sottodirectory contiene un proprio Makefile, come esempio si può vedere il file `spe/worker/Makefile`.

```
ROOTPATH := ../../../../..

include $(ROOTPATH)/make.defs

INCLUDE              += -I$(ROOTPATH)/mammut/include \
                        -I$(ROOTPATH)/assist/include
```

```
MAMMUT_PROCESSOR      := spu
PROGRAM               := worker_spe
MAMMUT_LIBRARY_EMBED  := libworker_spe.a
IMPORTS               := $(ROOTPATH)/mammut/src/spe/libmammut_spe.a

include $(ROOTPATH)/make.footer
```

In questo caso, la variabile `MAMMUT_PROCESSOR` indica che si vuole ottenere un eseguibile per l'`SPE`. La variabile `PROGRAM` indica il nome dell'eseguibile, e deve coincidere con il nome del `spu_program_handle_t` che dovrà essere dichiarato sul PPE. La variabile `MAMMUT_LIBRARY_EMBED` indica il nome per l'*embed static library* da generare. Infine, nella variabile `IMPORTS` viene indicata la libreria statica per il supporto MammuT lato SPE. Per le altre sottodirectory il Makefile è praticamente identico, ovviamente cambiano i nomi degli eseguibili e delle librerie statiche prodotte. Questa struttura di compilazione è regolare e potrebbe essere generata direttamente a partire dall'analisi del blocco `generic` di ASSIST-CL.

4.3.2 Header comune

La funzione delle direttive `#define` in ASSIST-CL e in LC può essere svolta da un header comune, da includere in tutti i sorgenti del programma. Nell'header `common.h` vengono espresse queste costanti globali.

```
#define KK 1
#define TIMES 10
#define ARRAYDIM 4096

typedef int Item;
```

Le costanti `KK`, `TIMES` e `ARRAYDIM` hanno la stessa funzione vista nel codice LC. La dimensione degli array è di 4096 interi, occupando esattamente 16

KB, il limite massimo per la dimensione dei messaggi sui canali MammuT. Le costanti `N` e `PARTDIM` non sono presenti; il numero di VP e la dimensione della partizione compariranno più avanti nel codice.

4.3.3 Grafo

Nel capitolo 3 è stata anticipata la necessità di avere una descrizione dell'applicazione conosciuta da tutti i processi, tramite una classe apposita. Questa classe svolge lo stesso ruolo del costrutto `generic` in ASSIST-CL, e del costrutto `parallel` in LC: rendere noto a tutti i processi la struttura dell'applicazione. Questa corrispondenza segue ancora una volta uno schema regolare, simile a quello visto per la traduzione da ASSIST-CL ad LC. Il grafo dell'applicazione è espresso nell'header `mygraph.h`.

```
#pragma once
#include <common/assistgraph.h>

class MyGraph : public Assist::AssistGraph
{
public:
    /* processi */
    MammuT::ProcIDRange generator;
    MammuT::ProcIDRange w;
    MammuT::ProcIDRange printer;

    /* Input Section */
    Assist::ISData is;

    /* Output Section */
    Assist::OSData os;

    Assist::InputStreamData aIStream;
    Assist::InputStreamData thresholdIStream;
```

```
    Assist::OutputStreamData resultOStream;
    Assist::OutputStreamData counterOStream;


    MyGraph(unsigned spes) : Assist::AssistGraph(spes)
    {
        using namespace Assist::Distr;
        using namespace Assist::Collections;

        setProcIDRange(generator, 1);
        setProcIDRange(w, 2);
        setProcIDRange(printer, 1);

        is.setVpmRange(this, w);
        os.setVpmRange(this, w);

        aIStream.setInputStream(this, is, generator[0], SCATTER);
        thresholdIStream.setInputStream(this, is, generator[0],
                                        MULTICAST);

        resultOStream.setOutputStream(this, os, printer[0],
                                      GATHER);
        counterOStream.setOutputStream(this, os, printer[0],
                                       FROMALL);
    }
};
```

Gli array di processi `generator`, `w` e `printer` corrispondono esattamente ai processi definiti in LC. Per i processi `is` e `os` invece vengono definiti i descrittori per la *Input Section* e per la *Output Section* corrispondenti. Il membro `aIStream` è il descrittore dell'*input stream* corrispondente allo stream `a` nel codice ASSIST, e il membro `thresholdIStream` è il descrittore corrispondente allo stream `threshold`. I membri `resultOStream` e `counterOStream` sono i descrittori per gli *output stream* del *parmod*, corrispondenti rispettivamente agli stream `result` e `counter` nel codice ASSIST.

Nel costruttore della classe `MyGraph`, ogni membro viene inizializzato con il corrispondente metodo `set*`. Tramite le chiamate ai metodi `setProcIDRange` vengono impostati gli id per gli array di processi dell'applicazione. Gli array di id `generator` e `printer` hanno dimensione 1, mentre l'array `w` ha dimensione 2. La costante `N` non è più necessaria perchè il numero di elementi di un array di processi è ottenibile tramite il metodo `size` invocato sugli oggetti corrispondenti ai range di id.

Tramite il metodo `setVpmRange` viene inizializzato il descrittore `is`, connettendo la *Input Section* ai relativi VP e allocando l'id per il processo corrispondente. Il descrittore `os` viene inizializzato allo stesso modo. Nella libreria il costrutto *parmod* viene implementato "agganciando" una *Input Section* ed una *Output Section* ad un array di processi VP, con la stessa struttura vista nel codice LC.

Il codice del costruttore prosegue con la configurazione dei descrittori degli stream. Il descrittore `aIStream` viene impostato tramite il metodo `setInputStream`. Il processo `generator[0]` viene indicato come mittente dello stream, e l'unica politica di distribuzione attivata è la `SCATTER`. Per il descrittore `thresholdIStream` viene attivata esclusivamente la politica di distribuzione `MULTICAST`. L'inizializzazione dei descrittori per gli *output stream* segue lo stesso schema. Ogni membro della classe è `public`, in modo da offrire un accesso diretto ai descrittori contenuti nel grafo.

Un compilatore può ricavare la classe per il grafo a partire dal costrutto `generic` in ASSIST, e dalle interfacce dei sequenziali e dei *parmod* che compongono l'applicazione. La traduzione è agevolata dal fatto che gli array di id di canali per la distribuzione del dato (`a_vp` e `threshold_vp`) e per il collezionamento dei risultati (`vp_a` e `vp_threshold`), presenti nel codice LC, vengono allocati automaticamente durante l'impostazione degli stream tramite

i metodi `setInputStream` e `setOutputStream`. Il metodo `setInputStream` inoltre assegna automaticamente i bit per identificare i vari array di canali per la distribuzione degli stream in ingresso alla *Input Section*, perciò le costanti `MASK_A` e `MASK_THRESOLD` non sono più necessarie.

4.3.4 Codice lato PPE

Il processo sul PPE deve registrare i processi e tutti i canali dell'applicazione. Mentre nel caso della libreria MammuT le registrazioni sono a carico del programmatore, nel caso della libreria CellAssist vengono fornite alcune classi per gestire questa fase in maniera dichiarativa.

```
#include <cstdio>
#include <ppe/libcom.h>
#include <ppe/channels/ppe_channels.h>
#include <ppe/inputsectionspe.h>
#include <ppe/inputstreamspe.h>
#include <ppe/outputstreamspe.h>
#include <ppe/procarray.h>
#include <ppe/types.h>
#include "common.h"
#include "mygraph.h"

/* DEVE ESSERE il nome del binario spu */
extern spe_program_handle_t generator_spe;
extern spe_program_handle_t inputsection_spe;
extern spe_program_handle_t worker_spe;
extern spe_program_handle_t outputsection_spe;
extern spe_program_handle_t printer_spe;

using namespace MammuT;
using namespace MammuT::Channels;
using namespace Assist::IS;
using namespace Assist::OS;
```

```
typedef Ppe_channel<NONE, Item, KK, 0> PPEchType;
typedef InputStreamSPE<PPEchType> InputStreamType;
typedef OutputStreamSPE<PPEchType> OutputStreamType;


int main()
{
    Stcom com;
    MyGraph graph(6);

    /* registrazione degli array di processi */
    registerProcArray(&com, graph.generator, &generator_spe);
    registerProcArray(&com, graph.is.procRange(),
                      &inputsection_spe);
    registerProcArray(&com, graph.w, &worker_spe);
    registerProcArray(&com, graph.os.procRange(),
                      &outputsection_spe);
    registerProcArray(&com, graph.printer, &printer_spe);

    InputSectionSPE isSpe(graph.is, &com);
    InputStreamType aIStream(graph.aIStream, graph.is, &com);
    InputStreamType thresholdIStream(graph.thresholdIStream,
                                     graph.is, &com);

    OutputStreamType resultOStream(graph.resultOStream,
                                   graph.os, &com);
    OutputStreamType counterOStream(graph.counterOStream,
                                    graph.os, &com);

    com.start();

    return 0;
}
```

Le dichiarazioni delle variabili globali `extern spe_program_handle_t` indicano i descrittori di ogni *embed static library* inclusa nell'eseguibile del PPE. Il nome di ogni descrittore deve essere uguale a quello espresso nel

Makefile relativo (variabile `PROGRAM`).
I canali da registrare (`PPEchType`) hanno grado di asincronia `KK`, e messaggio di tipo `Item`. Il primo parametro del *template* indica il ruolo svolto dal PPE nei canali di questo tipo. Il valore `NONE` in questo caso indica che il PPE non svolge alcun ruolo nella comunicazione, dunque si tratta di canali SPE - SPE.

Il processo crea subito un'istanza del supporto (`com`) e un'istanza del grafo (`graph`). Nelle righe successive viene invocata la funzione `registerProcArray` su ogni array di processo. La fase successiva corrisponde alla registrazione dei canali. La classe `InputSectionSPE` permette la registrazione automatica sia del processo allocato per la *Input Section*, sia dell'array di canali per i codici operativi. Le classi `InputStreamSPE` e `OutputStreamSPE` permettono la registrazione automatica dei canali associati agli stream.

Anche in questo caso il compilatore può ricavare agevolmente il codice, producendo una chiamata al metodo `registerProcArray` per ogni array di processi, e la dichiarazione di un oggetto per ogni stream e per ogni *Input Section* nell'applicazione.

4.3.5 Processo `generator`

Il processo `generator` genera gli elementi sugli stream in ingresso alla *Input Section*.

```
#include <common/assistconstants.h>
#include <spe/libcom.h>
#include <spe/channels/spe_channels.h>
#include <spe/channelarrayspe.h>
#include "../../common.h"
#include "../../mygraph.h"

using namespace MammuT::Channels;
```

```
using namespace MammuT::Debug;

/* canali dati */
typedef Spe_channel<SENDER, Item, KK, 0, IMP_WITH_SIGNAL> SRSPEch;
typedef ChannelArraySPE<SRSPEch> SRSPEchArray;


int main()
{
    Stcom com;
    MyGraph graph(6);

    /* dichiarazione canali verso la input section */
    SRSPEchArray chA(graph.aIStream.inputChRange(),
                     &com, ARRAYDIM);
    SRSPEchArray chThreshold(graph.thresholdIStream.inputChRange(),
                             &com);

    /* registrazione canali */
    chA.registerAll(&com);
    chThreshold.registerAll(&com);

    Item *msgA = chA[0]->allocMsg();
    Item *msgThreshold = chThreshold[0]->allocMsg();

    int waitTagS = -1;

    /* costruzione messaggi: array */
    for (int i = 0; i < ARRAYDIM; i++)
        msgA[i] = i;

    chA[0]->send(msgA);

    /* invio dei dati TIMES volte alla IS */
    for (int i = 0; i < TIMES; i++) {
        if (waitTagS != -1)
            chThreshold[0]->wait(msgThreshold, waitTagS);
```

```
        *msgThreshold = ARRAYDIM - i;
        waitTagS = chThreshold[0]->send(msgThreshold);
    }

    return 0;
}
```

Come in tutti i processi dell'applicazione, vengono create le istanze del supporto e del grafo. I canali verso la *Input Section* vengono dichiarati tramite i due array di canali `chA` e `chThreshold`. Ogni array di canali è istanziato a partire dal range di id corrispondente al canale in ingresso alla *Input Section* per quel particolare stream (metodo `inputChRange` invocato sul descrittore dell'input stream). Tramite il metodo `registerAll` vengono registrati tutti i canali dei due array (gli array `chA` e `chThreshold` hanno entrambi dimensione uno). I messaggi `msgA` e `msgThreshold` vengono allocati tramite il metodo `allocMsg` invocato sul canale corrispondente, e tramite un ciclo `for` viene inizializzato l'array da spedire. A questo punto viene inviato uno stream di valori crescenti sul canale contenuto nell'array `chThreshold`.

Il dettaglio importante nel codice del processo `generator` ai fini di un'ipotetica compilazione è la prima parte di dichiarazione e registrazione dei canali. Questa parte può essere ricavata dall'interfaccia del modulo sequenziale `generator` nel codice ASSIST.

4.3.6 Codice del processo per la Input Section

La *Input Section* deve gestire due stream, dunque si dovranno istanziare due costrutti `Emit`. Per definire una *Input Section* personalizzata, è possibile definire una nuova classe `CustomIS` eredita dalla classe `InputSection`.

```
#include <common/assistconstants.h>
#include <spe/libcom.h>
```

```
#include <spe/channels/spe_channels.h>
#include <spe/channelarrayspe.h>
#include <spe/emit.h>
#include <spe/inputsection.h>
#include <spe/inputstream.h>
#include "../../common.h"
#include "../../mygraph.h"

using namespace MammuT::Channels;
using namespace MammuT::Debug;
using namespace Assist::IS;
using namespace Assist::Distr;
using namespace Assist::Types;

/* canale dati */
typedef Spe_channel<RECEIVER, Item, KK, 0, IMP_WITH_SIGNAL>
                    RRSPEch;
typedef ChannelArraySPE<RRSPEch> RRSPEchArray;


class CustomIS : public InputSection
{
public:
    typedef Emit<RRSPEch> InputType;

protected:
    InputType *m_emitA;
    InputType *m_emitThreshold;
    unsigned m_sendThresholds;

public:
    CustomIS(MyGraph &graph, Stcom *com) :
             InputSection(graph.is, com)
    {
        m_sendThresholds = 0;

        m_emitA =
            new InputType(this, graph.aIStream, com, ARRAYDIM);
```

```
        m_emitThreshold =
            new InputType(this, graph.thresholdIStream, com);
    }


    virtual ~CustomIS()
    {
        delete m_emitA;
        delete m_emitThreshold;
    }


    void preComputation(int guardID)
    {
        /* disattiva lo stream di input quando sono stati
         * spediti tutti i messaggi */
        if (guardID == m_emitThreshold->id() &&
                (++m_sendThresholds == TIMES)) {
            m_emitA->disable();
            m_emitThreshold->disable();
        }
    }
};


int main()
{
    Stcom com;
    MyGraph graph(6);

    /* dichiarazione input section */
    CustomIS is(graph, &com);

    /* sino a quando l'input stream e' abilitato */
    while (is.go() >= 0);

    return 0;
}
```

Gli oggetti `m_emitA` e `m_emitThreshold` corrispondono alle guardie `a_guard` e `threshold_guard` nel costrutto `input_section` specificato nel codice ASSIST. Il parametro `ARRAYDIM` nell'inizializzazione del membro `m_emitA` indica il numero di elementi che compongono l'array distribuito sullo stream.

Il metodo `preComputation` conta il numero di elementi ricevuti sull'oggetto `m_emitThreshold` e disattiva tutti i costrutti `Emit` connessi alla *Input Section* quando tutti gli elementi degli stream sono stati ricevuti. Quando tutti i costrutti `Emit` o `EmitAND` associati ad una *Input Section* sono disattivati, il metodo `go` restituisce -1. Il `main` sfrutta questa condizione per gestire la terminazione del processo, invocando il metodo `go` sino a quando il valore restituito è diverso da -1.

Lo schema utilizzato in questo esempio è valido in generale. Ogni costrutto `input_section` nel sorgente ASSIST può essere tradotto in una nuova classe che eredita dalla classe `InputSection`, istanziata su un processo apposito.

4.3.7 Codice per i processi VP

Nell'esempio considerato, i *worker* devono ricevere in input la propria partizione e il valore di soglia.

```
#include <common/assistconstants.h>
#include <spe/libcom.h>
#include <spe/channels/spe_channels.h>
#include <spe/channelarrayspe.h>
#include <spe/vpsection.h>
#include <spe/vpminputstream.h>
#include <spe/vpmoutputstream.h>
#include "../../common.h"
#include "../../mygraph.h"
```

```
using namespace MammuT::Channels;
using namespace MammuT::Debug;
using namespace Assist::VPS;
using namespace Assist::Distr;
using namespace Assist::Types;

/* canali dati */
typedef Spe_channel<SENDER, Item, KK, 0, IMP_WITH_SIGNAL>
                    SRSPEChType;
typedef Spe_channel<RECEIVER, Item, KK, 0, IMP_WITH_SIGNAL>
                    RRSPEChType;

/* stream VPM */
typedef VPMInputStream<RRSPEChType> VPMInputStreamMsg;
typedef VPMOutputStream<SRSPEChType> VPMOutputStreamType;


int main()
{
    Stcom com;
    MyGraph graph(6);

    Opc opc;

    VPSection vps(graph.is, &com);
    VPMInputStreamMsg
        vpmIStreamA(&vps, graph.aIStream, &com, ARRAYDIM);
    VPMInputStreamMsg
        vpmIStreamThreshold(&vps, graph.thresholdIStream, &com);

    VPMOutputStreamType
        vpmOStreamResult(graph.resultOStream, &com,
                         vpmIStreamA.numItemsScatter());
    VPMOutputStreamType
        vpmOStreamCounter(graph.counterOStream, &com);

    Item *array;
```

```
    Item *threshold;
    Item *msgCounter = vpmOStreamCounter.allocMsg();

    unsigned recvThresholds = 0;

    while (recvThresholds < TIMES) {
        opc = vps.activate();

    while (recvThresholds < TIMES) {
        opc = vps.activate();

        if (opc.mask == graph.thresholdIStream.getMask(MULTICAST)) {
            vpmOStreamResult.wait();
            vpmOStreamCounter.wait();

            array = vpmIStreamA.msgScatter();
            threshold = vpmIStreamThreshold.msg();

            *msgCounter = 0;
            for (int i = 0; i < vpmIStreamA.numItemsScatter();
                    i++) {
                array[i] = (array[i] < *threshold) ? 1 : 0;
                *msgCounter += 1;
            }

            vpmOStreamResult.send(array);
            vpmOStreamCounter.send(msgCounter);

            recvThresholds++;
        }
    }

    return 0;
}
```

Dopo la dichiarazione del supporto e del grafo, viene creata un'istanza della classe `VPSection`. L'oggetto viene creato passando al costruttore

il descrittore della *Input Section*. Durante l'inizializzazione viene registrato il canale per il codice operativo in ingresso. La composizione della *VP Section* prosegue registrando i canali utilizzati dalla *Input Section* per distribuire gli elementi degli stream. Nel codice LC questi canali sono dichiarati in ingresso al processo `w[i]` (i canali sono `a_vp[i]` e `threshold_vp[i]`). Tramite la libreria è possibile ottenere lo stesso risultato dichiarando un oggetto istanza della classe `VPMInputStream` per ogni stream gestito dalla *Input Section*. Questi oggetti vengono istanziati passando al costruttore il puntatore all'oggetto `VPSection`. In questo modo le `receive` sui canali in ingresso possono essere gestite automaticamente tramite il metodo `activate` della classe `VPSection`. Gli stream di output dei VP sono dichiarati tramite gli oggetti `vpmOStreamA` e `vpmOStreamThreshold`, istanze della classe `VPMOutputStream`. Questi oggetti corrispondono alla dichiarazione dei canali in output `vp_result[i]` e `vp_counter[i]`.

La dichiarazione degli stream `vpmIStreamA` e `vpmOStreamResult` richiede un approfondimento. L'oggetto `vpmIStreamA` viene istanziato passando la costante `ARRAYDIM` al costruttore. Questa costante indica la dimensione del messaggio che arriva sulla *Input Section*, e non la dimensione del messaggio che arriva sul VP. Nel caso di distribuzione *scatter* attivata su uno stream, come in questo caso, i canali vengono istanziati calcolando automaticamente la grandezza della partizione. Nel caso dell'oggetto `vpmOStreamResult` invece la dimensione del messaggio da specificare nel costruttore deve essere quella della partizione, ricavabile direttamente dallo stream in input tramite il metodo `numItemsScatter`.

Il codice prosegue con un ciclo `while` che termina quando il numero di elementi ricevuti sullo stream `vpmIStreamA` raggiunge il limite richiesto. Ad ogni iterazione, viene invocato il metodo `activate` sulla *VP Section*, e viene salvato il codice operativo ottenuto durante l'attivazione. In generale, le regole di attivazione della *VP Section* devono essere scritte come una sequenza di *branch* condizionali sulla base del valore della maschera contenuta nel codice operativo. Le maschere degli stream sono accessibili invocando il metodo

`getMask` sui descrittori degli stream. Ogni ramo corrisponde ad una regola di attivazione presente nel costrutto `virtual_processors`. Dopo aver invocato il metodo `activate`, il programmatore può accedere ai dati ricevuti tramite i metodi `msg` e `msgScatter` della classe `VPMInputStream`. Viene effettuato il calcolo e vengono inviati i risultati sugli stream di output.

Il codice è molto simile al listato LC. Anche in questo caso, lo schema di traduzione è generalizzabile e può essere sfruttato da un compilatore. La libreria cerca di facilitare la "connessione" tra *Input Section* e *VP Section* tramite le classi `VPMInputStream` e `VPMInputStream`, mentre la classe `VPSection` automatizza le `receive` dei dati proveniente dalla *Input Section*.

4.3.8 Codice del processo per la Output Section

Sul processo che esegue la *Output Section* occorre combinare i risultati parziali prodotti dai VP. Nel caso dell'array `result`, il risultato finale viene prodotto tramite un'operazione di *gather*, nel caso del *counter* il risultato è una sommatoria dei risultati ricevuti. Occorre dunque dichiarare una nuova classe che erediti dalla classe `OutputSection` e che gestisca due costrutti `Collect` per i due diversi stream in output.

```
#include <common/assistconstants.h>
#include <spe/libcom.h>
#include <spe/channels/spe_channels.h>
#include <spe/channelarrayspe.h>
#include <spe/outputsection.h>
#include <spe/outputstream.h>
#include <spe/collect.h>
#include "../../common.h"
#include "../../mygraph.h"

using namespace MammuT::Channels;
using namespace Assist::OS;
```

```
using namespace Assist::Collections;

/* canali dati */
typedef Spe_channel<SENDER, Item, KK, 0, IMP_WITH_SIGNAL>
                    SRSPEChType;
typedef Spe_channel<RECEIVER, Item, KK, 0, IMP_WITH_SIGNAL>
                    RRSPEChType;

/* output stream */
typedef OutputStream<SRSPEChType> OStreamType;

/* collect */
typedef Collect<OStreamType> CollectType;


class CustomOS : public OutputSection
{
protected:
    CollectType *m_collectCounter;
    CollectType  *m_collectResult;
    int m_recvs;
    unsigned m_vpmNum;
    Item *m_msgCounter;

public:
    CustomOS(MyGraph *g, MammuT::Stcom *com) : OutputSection(g->os)
    {
        m_collectResult =
            new CollectType(this, g->resultOStream, com,
                            ARRAYDIM / g->os.vpmRange().size());
        m_collectCounter =
            new CollectType(this, g->counterOStream, com);

        m_msgCounter =
            m_collectCounter->ostream()->chOut()->allocMsg();
        m_collectCounter->ostream()->setMsg(m_msgCounter);

        m_vpmNum = g->os.vpmRange().size();
```

```
        m_recvs = 0;
    }


    void postComputation(int streamID)
    {
        if (streamID == m_collectCounter->id()) {
            //m_collectCounter->ostream()->wait();
            *m_msgCounter = 0;
            for (int i = 0; i < m_vpmNum; i++)
                *m_msgCounter += *(m_collectCounter->msg(i));
        }

        /* disattiva tutte le guardie quando tutti i messaggi
         * sono arrivati */
        if (++m_recvs == TIMES * 2) {
            m_collectResult->disable();
            m_collectCounter->disable();
        }
    }
};


int main()
{
    MammuT::Stcom com;
    MyGraph graph(6);

    /* definizione struttura input section */
    CustomOS os(&graph, &com);

    /* sino a quando l'output section non trova tutte le
     * guardie disattivate */
    while (os.go() != -1);

    return 0;
}
```

Il costruttore della classe `CustomOS` inizializza i due costrutti `Collect`. Il membro `m_collectResult` viene inizializzato a partire dal descrittore `g->resultOStream`. Tale descrittore era stato inizializzato nel grafo, attivando la politica `GATHER`. L'ultimo parametro del costruttore è la dimensione del messaggio; ogni canale riceve la partizione da un VP. Il membro `m_collectCounter` viene inizializzato allo stesso modo, a partire dal descrittore `g->counterOStream`, su cui era stata attivata la politica `FROMALL`. In questo caso la dimensione del messaggio non viene specificata (il valore di default è 1).

Il membro `m_msgCounter` è un puntatore ad un area di memoria allocata per contenere il messaggio da inviare sullo stream gestito dal membro `m_collectCounter`. L'allocazione esplicita del messaggio in questo caso deve essere effettuata solo per l'utilizzo dell'oggetto `m_collectCounter`, che ha attivata la politica `FROMALL`; in caso di politica `GATHER` invece il messaggio viene costruito automaticamente. Il metodo `setMsg` viene invocato sul membro `m_collectCounter` per indicare l'area di memoria che conterrà il messaggio da inviare sullo stream.

L'implementazione del metodo `postComputation` effettua innanzitutto la sommatoria corrispondente all'ultimo passo della *reduce*. Ovviamente questo passo va effettuato solamente quando tutti i risultati delle *reduce* parziali effettuate sui VP sono stati ricevuti. A programma è possibile sapere qual'è la `Collect` verificata nell'attivazione corrente della *Output Section*, confrontando il valore del parametro `streamID` con l'id delle `Collect` che compongono la *Output Section*. In questo caso, il parametro `streamID` viene confrontato con l'id della `Collect` `m_collectCounter`: se l'id corrisponde, viene effetuata la sommatoria.

La seconda parte del metodo gestisce la terminazione nello stesso modo in cui viene gestita sulla *Input Section*. Vengono contate le attivazioni, disattivando i costrutti `Collect` quando il contatore raggiunge il valore limite. Il metodo `go` della classe `OutputSection` è analogo al metodo della classe *Input Section*, perciò anche in questo caso nel `main` viene invocato il metodo

go sino a quando il valore restituito è diverso da -1.

Lo schema utilizzato in questo esempio è valido in generale. Ogni costrutto output_section nel sorgente ASSIST può essere tradotto in una nuova classe che eredita dalla classe OutputSection, istanziata su un processo apposito. Ogni guardia nel costrutto output_section viene tradotta in un oggetto Collect.

4.3.9 Processo printer

Il processo printer riceve semplicemente i risultati e ne verifica la correttezza.

```
#include <common/assistconstants.h>
#include <common/debug/test.h>
#include <spe/libcom.h>
#include <spe/channels/spe_channels.h>
#include <spe/channelarrayspe.h>
#include "../../common.h"
#include "../../mygraph.h"

using namespace MammuT::Channels;
using namespace MammuT::Debug;

/* canali dati */
typedef Spe_channel<RECEIVER, Item, KK, 0, IMP_WITH_SIGNAL>
                    RRSPEch;
typedef ChannelArraySPE<RRSPEch> RRSPEchArray;


int main()
{
    Stcom com;
    MyGraph graph(6);
```

```
    /* dichiarazione canali verso la input section */
    RRSPEchArray chResult(graph.resultOStream.outputChRange(),
                          &com, ARRAYDIM);
    RRSPEchArray chCounter(graph.counterOStream.outputChRange(),
                           &com);

    /* registrazione canali */
    chResult.registerAll(&com);
    chCounter.registerAll(&com);

    Item *msgResult;
    Item *msgCounter;
    bool result = true;

    for (int i = 0; i < TIMES; i++) {
        /* ricezione risultati */
        msgResult = chResult[0]->receive();
        msgCounter = chCounter[0]->receive();

        /* costruzione messaggi: array */
        Item verifier = 0;
        for (int j = 0; j < ARRAYDIM; j++)
            verifier += msgResult[j];

        result = result && verifier == *msgCounter;
    }

    checkPoint(result, "MAP + REDUCE");

    return 0;
}
```

Come in tutti i processi dell'applicazione, vengono create le istanze del supporto e del grafo. I canali in ingresso vengono dichiarati tramite i due array di canali `chResult` e `chCounter`. Ogni array di canali è istanziato a partire dal range di id corrispondente al canale in uscita dalla *Output Section*

per quel particolare stream (metodo `outputChRange` invocato sul descrittore dell'output stream). Tramite il metodo `registerAll` vengono registrati tutti i canali dei due array (gli array `chResult` e `chCounter` hanno entrambi dimensione uno). Il resto del codice effettua le `receive` dei risultati ed effettua un test di correttezza su ogni dato ricevuto. La chiamata alla funzione `checkPoint` serve solo a stampare l'esito della computazione totale.

Anche in questo caso, come per il processo `generator`, il dettaglio importante ai fini di un'ipotetica compilazione è la prima parte di dichiarazione e registrazione dei canali. Questa parte può essere ricavata dall'interfaccia del modulo sequenziale `printer` nel codice ASSIST.

Capitolo 5

Valutazione delle prestazioni

Nelle sezioni precedenti è stato presentato il lavoro svolto da un punto di vista implementativo e funzionale. In questo capitolo saranno analizzate le prestazioni dei costrutti implementati, considerando che la gestione del non-determinismo rende a volte impossibile la formalizzazione di un modello dei costi esaustivo.

Nel capitolo 3 è stato presentato il costrutto *parmod* di ASSIST spiegando che le tre sezioni che lo costituiscono cooperano secondo uno schema a pipeline. Utilizzando la teoria delle code è possibile dedurre un metodo formale per il calcolo del tempo di servizio di un pipeline, come descritto in [8]. Nel caso del *parmod* si avrebbe il seguente risultato:

$$T_{parmod} = max\{T_A, T_{IS}, T_{VPS}, T_{OS}\}$$

con T_A che rappresenta il tempo di interarrivo al modulo parallelo e T_{IS}, T_{VPS} e T_{OS} che rappresentano rispettivamente i tempi di servizio di *Input Section*, *VP Section* e *Output Section*.

Se si considera il *parmod* come modulo isolato [1] è possibile ricondurre il

[1] Considerare un sottosistema come isolato significa supporre che il sottosistema IN (che genera lo stream di ingresso) e il sottosistema OUT (che consuma lo stream di uscita) abbiano banda alta a piacere, così da considerare il sottosistema in questione come "collo di bottiglia".

suo tempo di servizio al massimo tra i tempi di servizio delle tre sezioni che lo compongono.

$$T_{parmod} = max\{T_{IS}, T_{VPS}, T_{OS}\}$$

Una parallelizzazione su pipeline per essere efficiente richiede che i suoi stadi siano bilanciati. Questo è chiaramente vero anche per il *parmod* ma in questo caso è possibile fare una ulteriore considerazione. L'unica sezione in cui è possibile agire sul grado di parallelismo è la *VP Section* quindi si dovrebbero evitare situazioni in cui la *Input Section* o la *Output Section* siano i colli di bottiglia del pipeline. Quando lo stadio più lento coincide con la *VP Section* si può agire sul numero dei VP per ridurre il tempo di servizio e bilanciare il pipeline. Ovviamente c'è un limite al numero di VP istanziabili, che coincide col numero di processori disponibili: nel caso di una architettura Cell questo limite è dettato dal numero di SPE non ancora allocati per gli altri processi dell'applicazione.

La macchina utilizzata per i test è una PlayStation 3 della Sony. Il processore Cell presente sulla PS3 differisce da quello visto nel capitolo 1 per il numero di SPE utilizzabili (solo 6): uno dei due SPE non disponibili è adibito all'esecuzione di un processo hypervisor che regola l'accesso all'hardware del sistema (ad esempio impedendo l'utilizzo della scheda grafica) mentre l'altro è disabilitato a livello hardware.

Nel seguito del capitolo si procederà all'analisi di *Input Section*, *VP Section* e *Output Section*. Infine saranno analizzate le prestazioni di un'applicazione data parallel con stencil fisso, mettendo a confronto la versione sequenziale con quella parallela.

Tutti i tempi saranno espressi in cicli di clock τ del processore. Le misurazioni sono state effettuate utilizzando un timer messo a disposizione dall'architettura, chiamato *decrementer*, che viene aggiornato ogni 40τ. L'errore ϵ di una misurazione è quindi sempre di $\pm 40\tau$.

5.1 Input Section

La *Input Section* gestisce gli stream di input e la distribuzione dei valori ai VP. Un passo di distribuzione corrisponde a un passo del relativo comando alternativo, che utilizza guardie di tipo `GuardDistr`. Il comando alternativo è il costrutto che permette di gestire il nondeterminismo e in quanto tale non è possibile ricavare un suo modello dei costi preciso. L'esecuzione di un passo del comando alternativo è implementata nel metodo `eval` della classe `AltCmd` e consiste nel trovare una guardia verificata ed eseguire il blocco di codice ad essa associato. Una guardia è verificata quando il predicato booleano è vero (metodo `evaluate` della guardia) ed è possibile effettuare la comunicazione sul canale associato alla guardia (metodo `slotAvailable` della guardia); il codice associato alla guardia è contenuto nel suo metodo `compute`. Ipotizzando che le guardie siano sempre verificate il tempo di servizio del comando alternativo viene modellato nel modo seguente:

$$T_{eval} = T_{evaluate} + T_{slotAvailable} + T_{compute}$$

Il tempo di servizio dei metodi `evaluate` e `slotAvailable` è fisso e corrisponde a circa 240τ.

$$T_{evaluate} + T_{slotAvailable} = 240\tau$$

Il tempo di servizio del metodo `compute` varia a seconda del tipo di distribuzione utilizzato e può essere approssimato come:

$$T_{compute} = T_{waitDistr} + T_{preComputation} + T_{distr}$$

$T_{preComputation}$ rappresenta il tempo di calcolo relativo alla fase di pre-elaborazione definita dal programmatore. T_{distr} rappresenta il tempo di servizio di un'operazione di distribuzione mentre $T_{waitDistr}$ rappresenta il tempo passato ad attendere la terminazione della precedente operazione di distribuzione dall'input stream corrente: entrambi dipendono dalla politica di distribuzione utilizzata. L'operazione di *wait* è necessaria perchè i messaggi inviati

ai VP sono le variabili targa dei canali in ingresso alla *Input Section*; ulteriori receive sui canali invaliderebbero il messaggio ricevuto con l'ultima receive prima che sia stato correttamente distribuito ai VP (vedi sezione 2.2.2). La tabella seguente mostra un'approssimazione di questi valori per ogni tipo di distribuzione.

Politica	T_{distr}	$T_{waitDistr}$
On Demand	$T_{sendCop} + T_{eval}$ [a]	T_{wait}
Multicast	N [b] $*(T_{sendCop} + T_{send}(sizeof(item) * M$ [c] $))$	$N * T_{wait}$
Scatter	$N * (T_{sendCop} + T_{send}(sizeof(item) * g$ [d] $))+$ $(x$ [e] $* g * T_{copy}$ [f] $)$	$N * T_{wait}$
Scheduled	$T_{sendCop} + T_{send}(sizeof(item) * M)$	T_{wait}

[a] con $T_{compute} = T_{send}(sizeof(item) * M)$
[b] numero di VP
[c] dimensione del messaggio (numero di elementi)
[d] dimensione della partizione (numero di elementi) calcolata come $\frac{M}{N}$
[e] numero di partizioni non allineate ai 16 Byte
[f] tempo per la copia (per valore) di un singolo elemento

Tabella 5.1: Politiche di distribuzione: modelli dei costi

Ogni distribuzione, oltre a distribuire i dati ai VP, si occupa dell'invio del codice operativo; il tempo di comunicazione per l'invio del codice operativo a un VP può essere approssimato come

$$T_{sendCop} = T_{wait} + T_{send}(sizeof(int) * 2)$$

con T_{wait} che rappresenta il tempo in attesa della terminazione del trasferimento del precedente codice operativo verso quel determinato VP.

Le prestazioni delle singole distribuzioni verranno ora illustrate attraverso i risultati ottenuti dai test effettuati. Il programma sviluppato per i test è costituito da un processo generatore, una input section e un numero di VP variabile. Il processo generatore invia alla *Input Section* uno stream di dati;

la *Input Section* gestisce lo stream di ingresso e distribuisce i valori ricevuti a un insieme di VP. L'applicazione è parametrica e dipendente dal numero di VP, dalla dimensione dei messaggi e dalla politica di distribuzione da adottare sulla *Input Section*. Sono state testate tutte le possibili combinazioni dei tre parametri, considerando messaggi di dimensione compresa tra 128 Byte e 16 KByte con step di 128 Byte, e un numero di VP compreso tra 1 e 4. I canali di comunicazione utilizzati sono quelli basati sui segnali per la sincronizzazione tra processi, con grado di asincronia pari a 1. Le misurazioni effettuate hanno riguardato la banda e il tempo di interpartenza dai moduli considerati. Il tempo di interpartenza è stato calcolato come l'inverso della banda.

La politica di distribuzione *Multicast* è implementata tramite una serie di `send`, due ad ogni VP, per codice operativo e dati. È ragionevole aspettarsi quindi un incremento dei tempi di comunicazione proporzionale al numero di VP.

Il grafico in figura 5.1 evidenzia come per messaggi di dimensione inferiore a una certa soglia i tempi di interpartenza siano quasi costanti mentre oltre questa soglia crescano linearmente con la dimensione del messaggio. Dal punto di vista della banda (figura 5.2) questo comporta una crescita lineare fino al valore di soglia per la dimensione del messaggio; poi la pendenza della curva si addolcisce indicando che la parte della comunicazione non sovrapposta al calcolo è sempre maggiore. Queste osservazioni rimangono valide per tutte le distribuzioni, come si vedrà nei relativi grafici.

La politica di distribuzione *Scheduled* effettua l'invio di dati e codice operativo a un solo VP, scelto a programma. Dal punto di vista delle prestazioni, è sicuramente la distribuzione meno onerosa tra quelle presenti nella libreria.

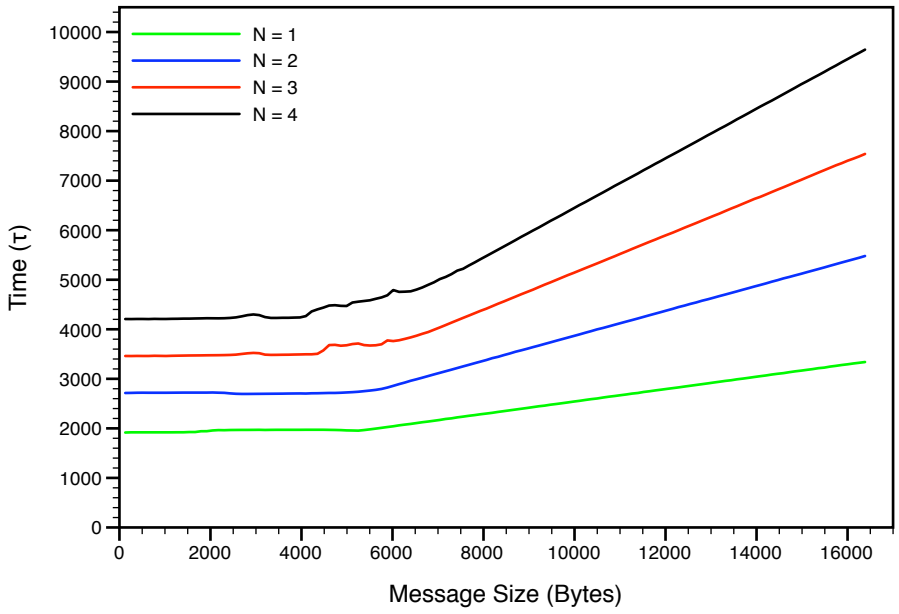

Figura 5.1: Tempi di interpartenza, distribuzione *Multicast*

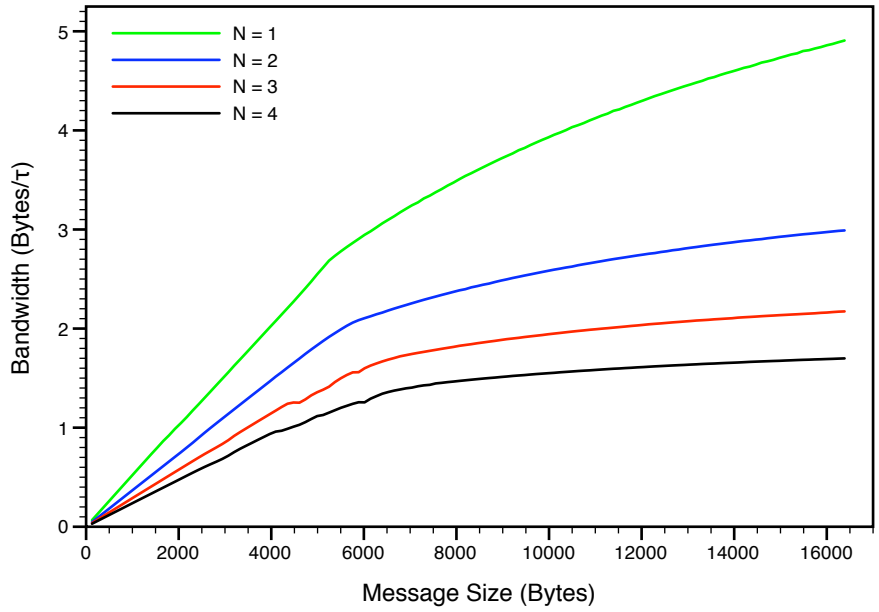

Figura 5.2: Banda, distribuzione *Multicast*

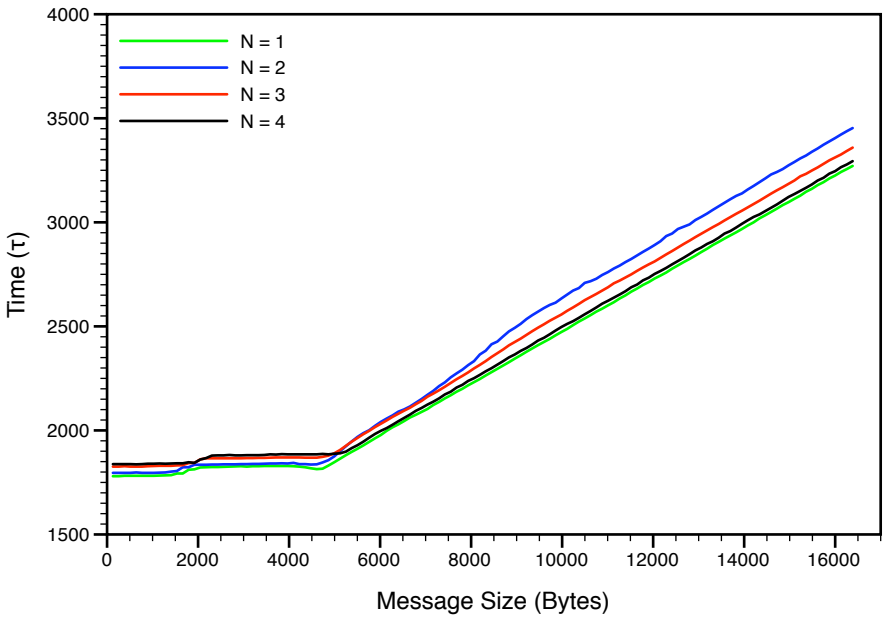

Figura 5.3: Tempi di interpartenza, distribuzione *Scheduled*

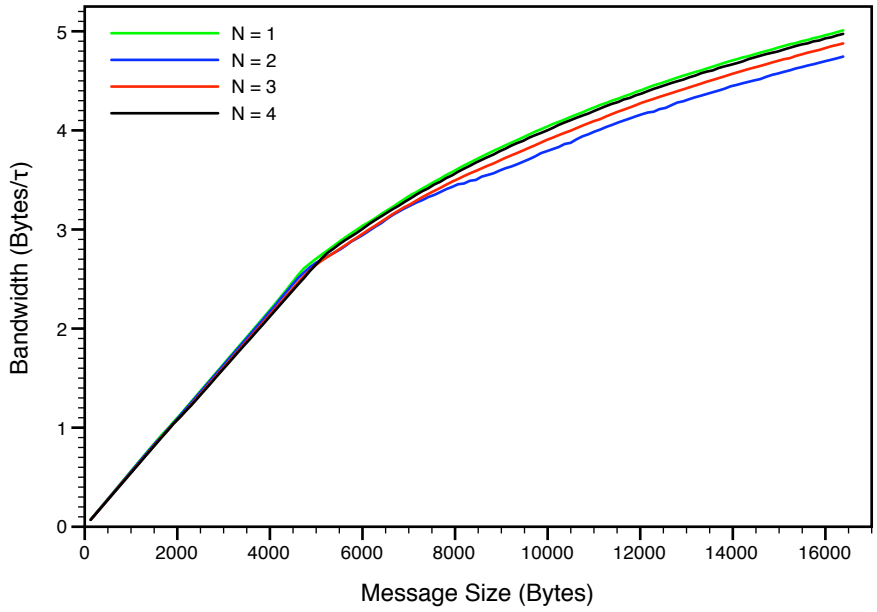

Figura 5.4: Banda, distribuzione *Scheduled*

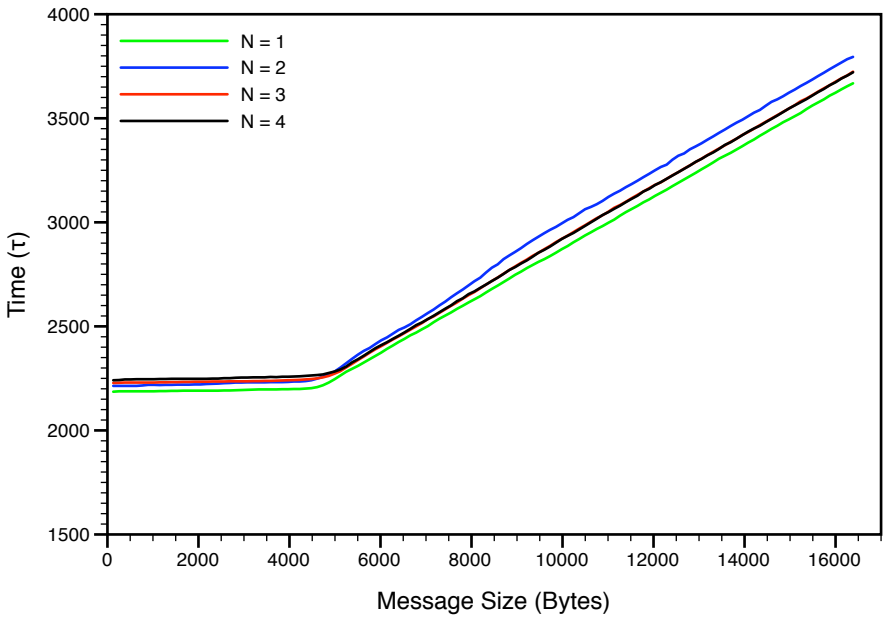

Figura 5.5: Tempi di interpartenza, distribuzione *On Demand*

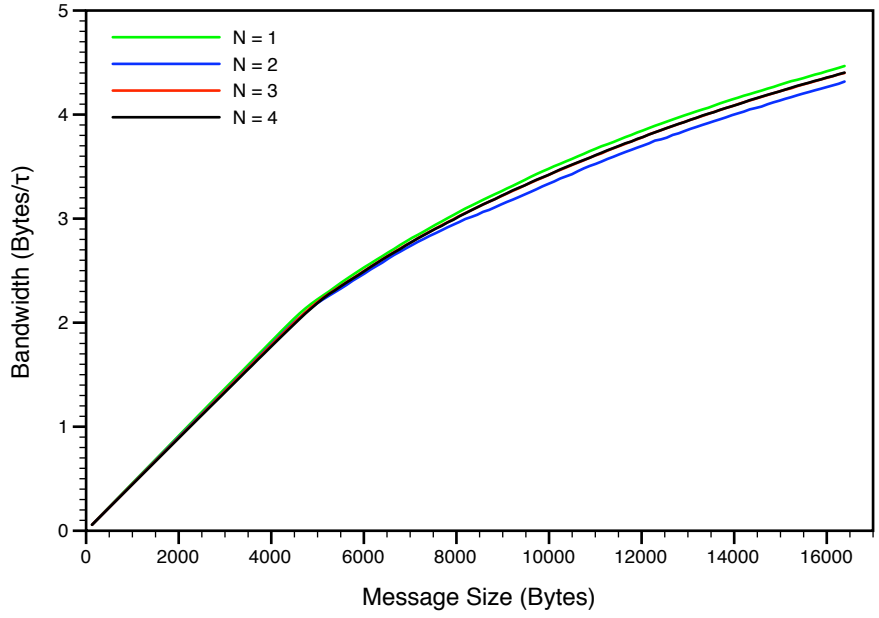

Figura 5.6: Banda, distribuzione *On Demand*

Il grafico in figura 5.3 evidenzia che la variabilità dei tempi di interpartenza tra il caso in cui siano presenti 4 VP e il caso in cui il VP presente sia solo 1, è molto bassa. Questo è dovuto al fatto che le operazioni svolte sono indipendenti dal numero di VP utilizzati; inoltre è necessario considerare anche gli eventuali piccoli errori introdotti dalle misurazioni effettuate col decrementer.

La politica di distribuzione *On Demand* è implementata tramite un comando alternativo in uscita: la *Input Section* individua il primo VP libero e gli invia il codice operativo e il valore su cui effettuare la computazione. Il comando alternativo è formato da guardie apposite in cui il metodo `compute` si occupa di effettuare la `send` dei dati. Le figure 5.5 e 5.6 mostrano rispettivamente l'andamento del tempo di interpartenza e della banda in base ai test effettuati. L'andamento della distribuzione *On Demand* è molto simile a quello della distribuzione *Scheduled*. La differenza nelle prestazioni è costante ed è imputabile al costo del comando alternativo in uscita.

La politica di distribuzione *Scatter* partiziona il valore strutturato ricevuto e invia ogni partizione a un VP diverso, insieme al codice operativo. Le figure 5.7, 5.8, 5.9 e 5.10 mostrano l'andamento del tempo di interpartenza e della banda in base ai test effettuati. Le prestazioni sono molto più irregolari rispetto alle altre distribuzioni. Questo è dovuto proprio a problemi legati all'operazione di partizionamento. Nella sezione 1.2, parlando di trasferimenti in DMA sulla struttura di interconnessione, si è detto che gli indirizzi base delle aree di memoria sorgente e destinazione devono essere allineati ai 16 Byte; si è detto inoltre che, per incrementare la performance del trasferimento, il produttore suggerisce di utilizzare aree di memoria con indirizzi di base allineati ai 128 Byte, in quanto gestiti più efficientemente dal protocollo della rete di interconnessione. I messaggi distribuiti dalla *Input Section* ai VP sono contenuti nelle variabili targa dei canali in ingresso alla *Input Section*: questo obbliga ad aspettare la terminazione della distribuzione del valore precedentemente ricevuto sullo stesso input stream (tramite la primitiva `wait`) ma al tempo stesso evita una copia per valore del messaggio

che, come si vedrà, risulta ben più pesante dal punto di vista computazionale.

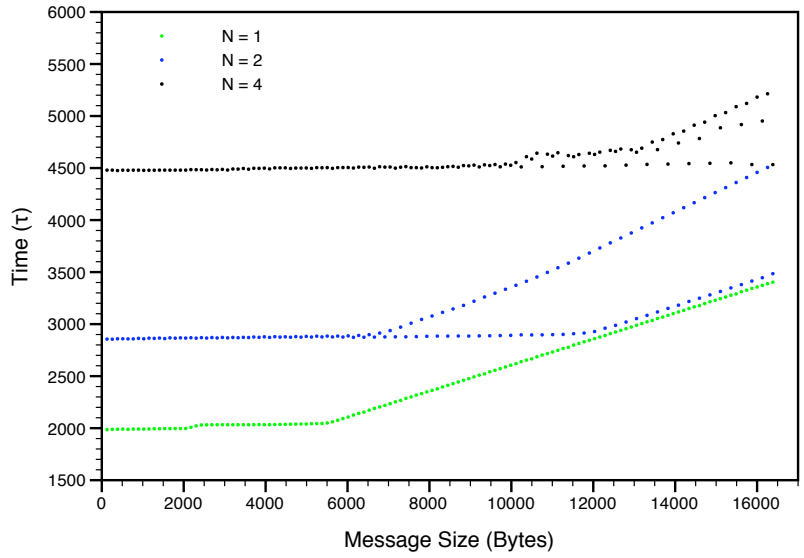

Figura 5.7: Tempi di interpartenza, distribuzione *Scatter*, $N = 1, 2, 4$
Le biforcazioni nei casi $N = 2$ e $N = 4$ sono dovute a diverse combinazioni di allineamenti in memoria (16 o 128 Byte) delle partizioni da distribuire.

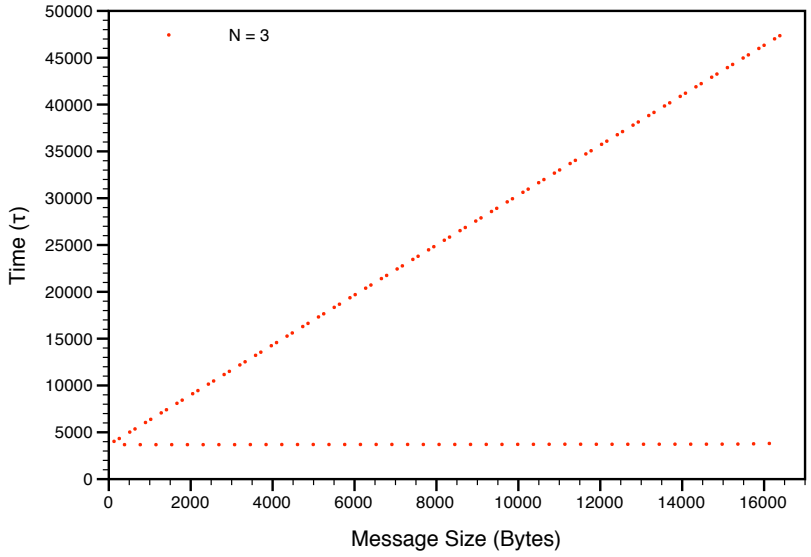

Figura 5.8: Tempi di interpartenza, distribuzione *Scatter*, $N = 3$
La biforcazione è dovuta all'introduzione delle copie in presenza di partizioni non allineate in memoria ai 16 Byte.

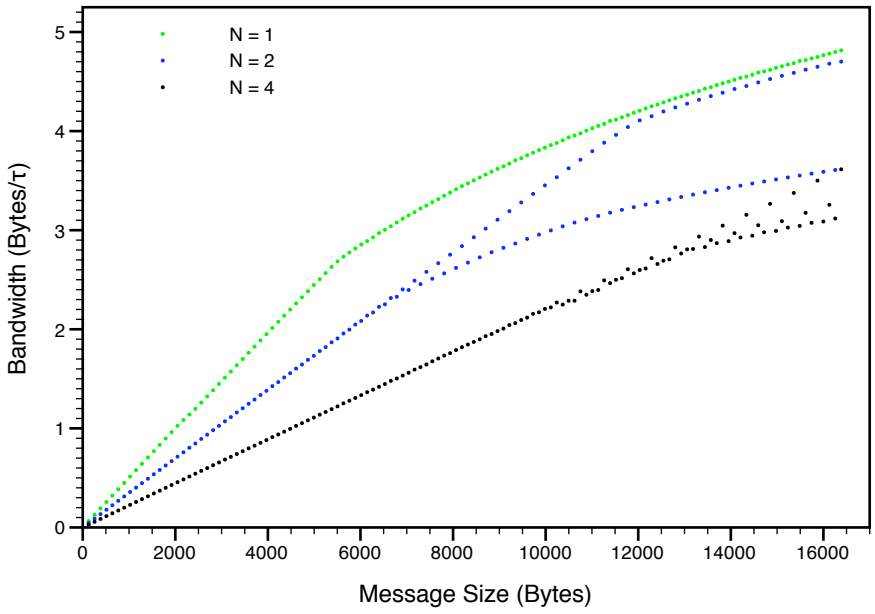

Figura 5.9: Banda, distribuzione *Scatter*, $N = 1, 2, 4$

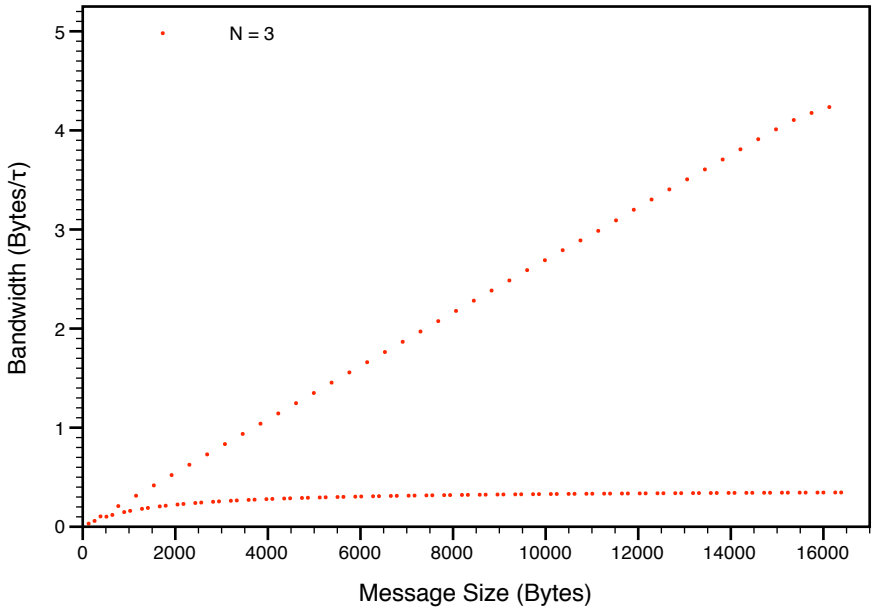

Figura 5.10: Banda, distribuzione *Scatter*, $N = 3$

Gli indirizzi base delle variabili targa dei canali sono sempre allineati ai 128 Byte: questo garantisce che l'indirizzo della prima partizione sia sempre allineato correttamente; lo stesso non può essere detto per le altre partizioni. Il calcolo della dimensione delle partizioni può essere schematizzato nel modo seguente:

```
g = M / N;
if (M % N) g++;
```

Questo significa che potrebbero verificarsi delle situazioni in cui l'ultima partizione risulta più piccola delle altre e il messaggio contiene degli elementi non significativi.

L'indirizzo di una generica partizione successiva alla prima si può trovare in uno dei seguenti stati:

- allineato a 128 Byte;
- allineato a 16 Byte ma non a 128 Byte;
- non allineato a 16 Byte.

Nei primi due casi per distribuire la partizione al VP corretto è sufficiente utilizzare l'indirizzo base della partizione calcolato: si utilizza quindi il messaggio contenuto nella variabile targa del canale associato all'input stream. L'ultimo caso merita invece un discorso a parte in quanto i dati, per poter essere spediti vanno copiati in una locazione di memoria il cui indirizzo base sia allineato quantomeno ai 16 Byte.

Una volta calcolate le dimensioni delle partizioni quindi si è in grado di ricavare gli indirizzi base di ognuna di esse e capire quali sono allineati almeno ai 16 Byte e quali invece no. Per ogni partizione non allineata viene allocata un'apposita area di memoria allineata ai 128 Byte da utilizzare per l'invio dei dati al relativo VP. Il fatto che i canali siano tipati staticamente rende possibile effettuare queste operazioni nella fase di inizializzazione del supporto: nel costruttore della classe che gestisce la distribuzione *Scatter* infatti,

si ricava dal canale in ingresso la dimensione del messaggio da scatterizzare, dal grafo il numero di VP e in base a queste informazioni si calcola il numero di partizioni non allineate e si allocano le aree di memoria necessarie per la loro gestione.

I grafici mostrano l'andamento di tempi di interpartenza e bande al verificarsi delle varie situazioni. I grafici relativi ai test effettuati utilizzando tre VP svelano in modo inequivocabile il peso delle copie per valore degli elementi delle partizioni non allineate. La differenza rispetto ai test effettuati con un numero di VP diverso è talmente grande che si è deciso di fare dei grafici separati.

Il caso che prevede l'utilizzo di un unico VP è poco significativo in termini di valori assoluti per testare la distribuzione *Scatter* ma è stato incluso per uniformità rispetto alle altre distribuzioni testate e comunque per dare un'idea dell'andamento delle performance al crescere del numero di VP.

Nel caso in cui siano presenti due VP il grafico evidenzia un duplice comportamento per messaggi con dimensioni maggiori di una certa soglia; questo è dovuto al diverso allineamento della seconda partizione. Le dimensioni dei messaggi nei test effettuati partono da 128 Byte e aumentano progressivamente con step di 128 Byte: utilizzando due VP quindi, si ha che la seconda partizione è sempre allineata ai 16 Byte. La differenza sta nel fatto che in alcuni casi l'allineamento è solo ai 16 Byte mentre nei casi rimanenti le partizioni sono allineate anche ai 128 Byte, dando luogo a performance migliori nei trasferimenti.

Il caso che prevede l'utilizzo di 4 VP fornisce dei risultati altrettanto altalenanti. La spiegazione è la stessa data per le oscillazioni riscontrate nel caso con due VP: le oscillazioni sono dovute a diversi allineamenti delle partizioni da inviare.

Osservando attentamente il grafico in figura 5.7 per $N = 4$, si nota che quando tutte le partizioni da spedire sono allineate ai 128 byte il tempo di interpartenza dalla *Input Section* è praticamente costante. Questo andamento è giustificato dal fatto che le comunicazioni sono interamente sovrapposte

all'elaborazione della *Input Section*. Sotto le stesse premesse è lecito attendersi lo stesso comportamento al crescere del numero di VP e le prestazioni possono essere approssimate da un modello lineare nella forma:

$$T_{scatter} = c_1 + c_2 N$$

Dall'analisi dei dati sperimentali sono stati ricavati i valori delle due costanti e l'espressione risultante è la seguente:

$$T_{scatter} = 1180 + (830 * N)$$

In generale si può concludere che le performance della distribuzione *Scatter* presentano un'alta variabilità in base alla dimensione del messaggio da partizionare e al numero dei VP tra cui deve essere partizionato. Il programmatore deve esserne cosciente e tenere in considerazione questi aspetti quando utilizza questa particolare politica di distribuzione dei dati.

5.2 VP Section

Nella sezione 3.6.3 è stata descritta l'implementazione del supporto per la VP Section dicendo che l'unico aspetto gestito è quello legato all'attivazione dei VP, quindi `receive` del codice operativo e `receive` dai canali dei dati indicati nel codice operativo. L'impatto della libreria sulle performance di un generico VP è quindi praticamente nullo: tutto dipende dal codice scritto dal programmatore.

5.3 Output Section

La *Output Section* gestisce il collezionamento dei risultati dai VP e il loro invio sugli stream di output. Un passo di collezionamento corrisponde a un passo del relativo comando alternativo, che utilizza guardie di tipo `GuardCollect*`. Per quanto riguarda la valutazione del comando alternativo vale il discorso fatto per la *Input Section*; l'unica differenza è il metodo `compute` che può essere approssimato come segue:

$$T_{compute} = T_{OutStreamWait} + T_{postComputation} + T_{send}(sizeof(Item) * M)$$

dove M è il numero di elementi da cui è composto il messaggio spedito sullo stream di output.

Nel caso della politica *Gather*, al $T_{compute}$ va aggiunto il costo della ricomposizione del messaggio, considerando la copia di tutti gli elementi in un'unica locazione di memoria da utilizzare per l'invio sullo stream di output. Quando sono state presentate le prestazioni della politica di distribuzione *Scatter* si è discusso di quanto siano dannose le copie dei messaggi; il risultato è che il tempo di interpartenza nel caso di politica di collection *Gather* è dominato dal tempo delle copie.

$T_{postComputation}$ rappresenta il tempo di calcolo relativo alla fase di post-elaborazione definita dal programmatore.

$T_{OutStreamWait}$ rappresenta il tempo passato ad attendere la terminazione della precedente operazione di collection relativa a un determinato output stream. Questo perchè per l'invio del messaggio sul canale in uscita si utilizzano le variabili targa dei canali in arrivo dai VP, e prima di ricevere altri dati dai VP bisogna assicurarsi che l'operazione di collezionamento precedente sia completamente terminata.

Per valutare le prestazioni delle varie politiche di collezionamento dei dati è stato sviluppato un programma di test costituito da un certo numero variabile di VP, una output section e un processo che riceve lo stream di output. Come nel caso della *Input Section*, l'applicazione è parametrica e dipendente dal numero di VP, dalla dimensione dei messaggi e dalla politica di collezionamento da adottare sulla *Output Section*. Sono state testate le politiche *Gather*, *FromOne* e *Forward*, considerando messaggi di dimensione compresa tra 128 Byte e 16 KByte con step di 128 Byte, e un numero di VP compreso tra 1 e 4. La politica *FromAll* è analoga alla *Gather* con la differenza che l'operazione di ricomposizione del messaggio deve essere effettuata dal programmatore nel metodo `postComputation`; discorso analogo per la politica *FromAny* nei confronti della *Forward*.

Lo schema dell'applicazione è il seguente: i processi VP generano i messaggi e li inviano alla *Output Section* che li colleziona secondo la politica impostata e li invia sullo stream di output a un processo che fa solo `receive`. I canali di comunicazione utilizzati sono quelli basati sui segnali per la sincronizzazione tra processi, con grado di asincronia pari a 1. Le misurazioni effettuate hanno riguardato la banda e il tempo di interpartenza dai moduli considerati. Il tempo di interpartenza è stato calcolato come l'inverso della banda. I test effettuati hanno mostrato che i tempi delle politiche di collect sono sostanzialmente indipendenti dal numero di VP presenti. Si ricorda che, per quanto riguarda la valutazione del comando alternativo, vale sempre l'ipotesi fatta nella *Input Section*, per cui tutte le guardie sono sempre considerate verificate.

Come anticipato in precedenza, il tempo di interpartenza nel caso di politica *Gather* è dominato dai tempi delle copie degli elementi per la ricomposizione del messaggio. Nel caso di politica *Forward*, con l'ipotesi fatta per il comando alternativo, il tempo di interpartenza è praticamente identico a quello della distribuzione *Scheduled* della *Input Section*. La politica *FromOne* costituisce l'unica eccezione in quanto una piccola parte del tempo di interpartenza dipende dal numero di VP. Questo perchè è implementata tramite un insieme di guardie, una per ogni VP, di cui solo una abilitata: ogni guardia non abilitata comporta un overhead di circa 110τ dovuto al test del predicato locale (metodo `evaluate`).

I grafici relativi alle politiche *Forward* e *FromOne* evidenziano come, per messaggi di dimensione inferiore a una certa soglia, i tempi di interpartenza siano quasi costanti mentre oltre questa soglia crescano linearmente con la dimensione del messaggio. Lo stesso comportamento può essere osservato anche nei grafici delle bande: si ha una crescita lineare fino al valore di soglia per la dimensione del messaggio e, una volta superato tale valore, la crescita della curva diventa sempre minore, ad indicare l'aumento della parte della comunicazione non sovrapposta al calcolo.

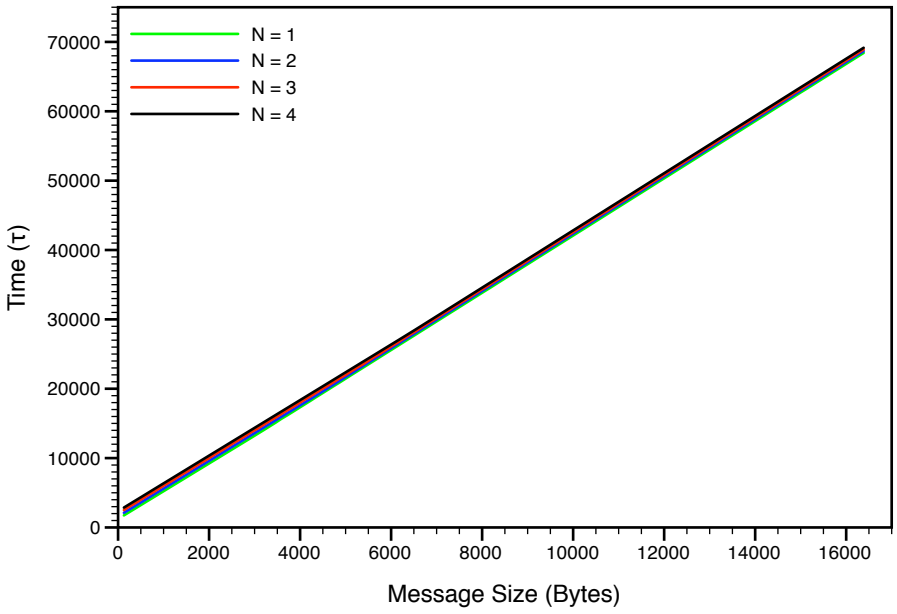

Figura 5.11: Tempi di interpartenza, politica *Gather*

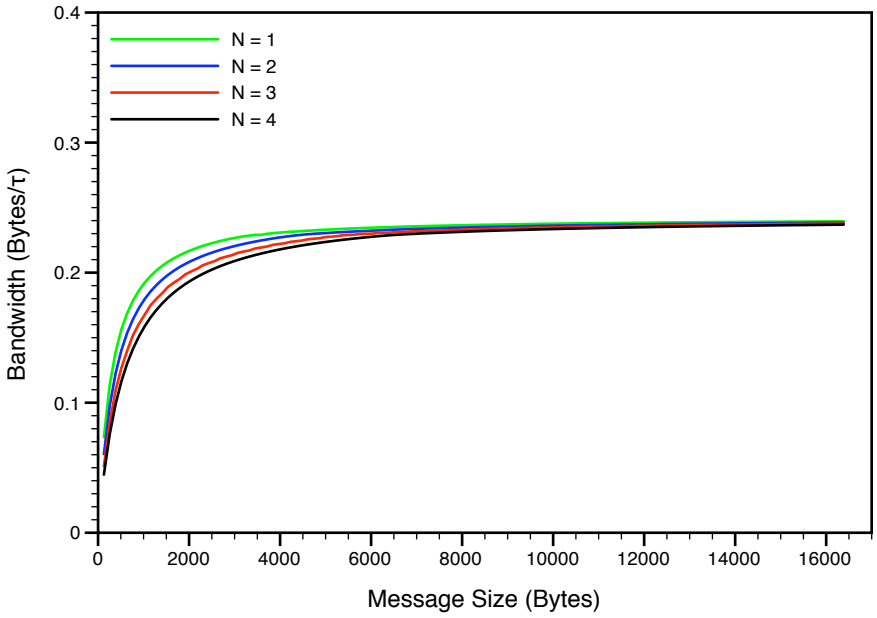

Figura 5.12: Banda, politica *Gather*

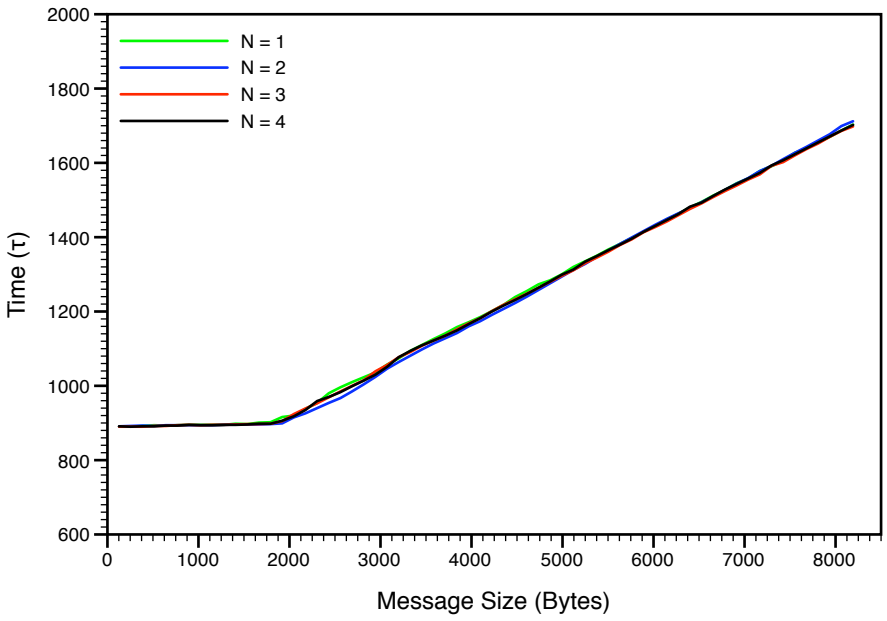

Figura 5.13: Tempi di interpartenza, politica *Forward*

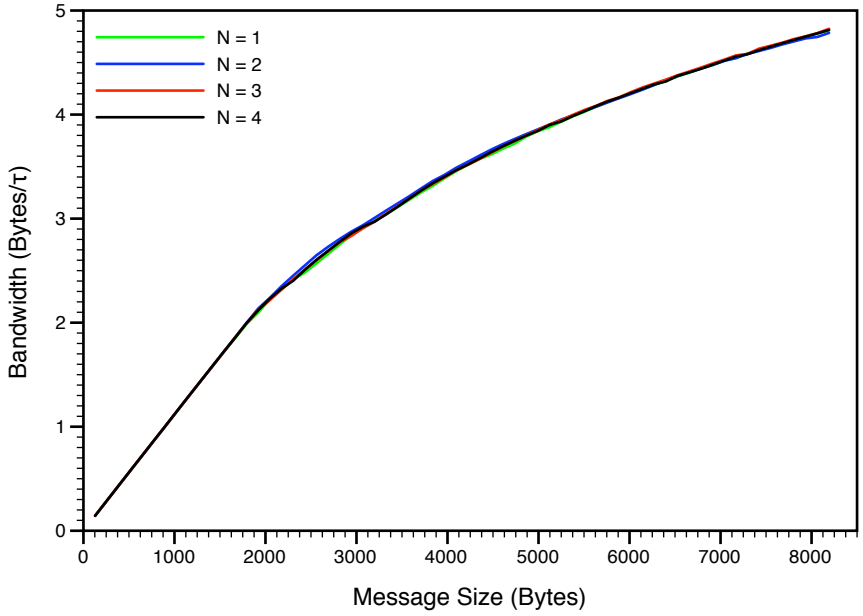

Figura 5.14: Banda, politica *Forward*

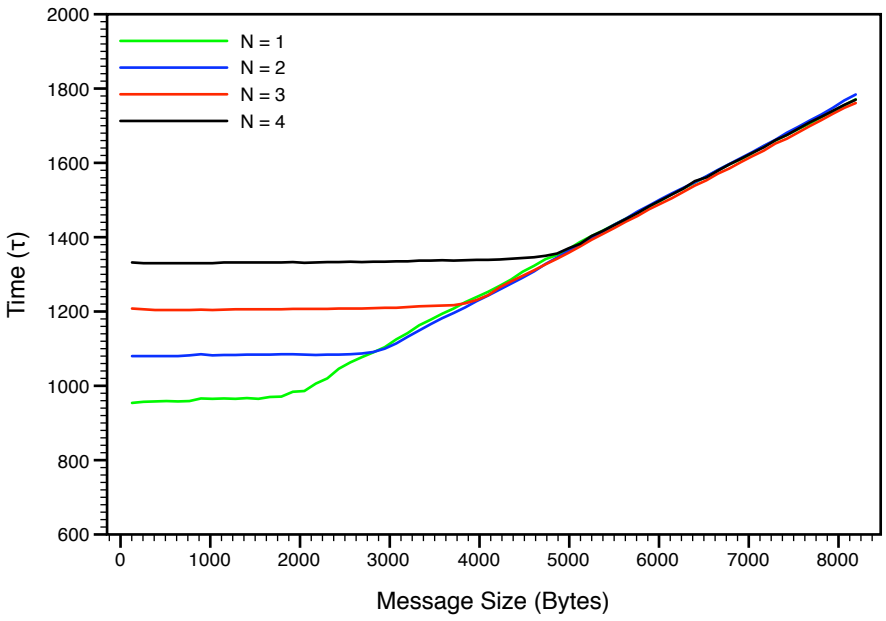

Figura 5.15: Tempi di interpartenza, politica *FromOne*

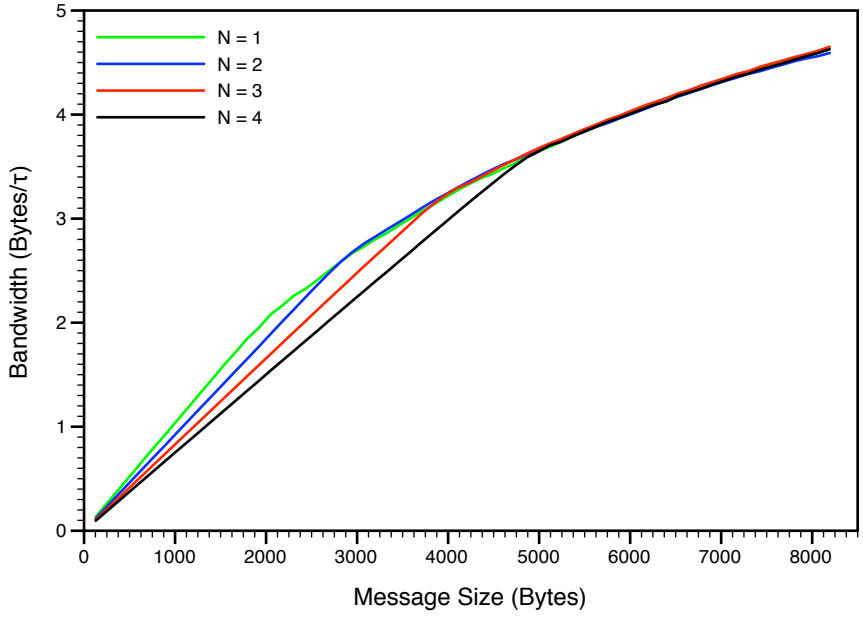

Figura 5.16: Banda, politica *FromOne*

5.4 Test: data parallel con stencil fisso

L'applicazione utilizzata per testare le performance della libreria è un semplice algoritmo data parallel con stencil. L'input del programma è una matrice quadrata, e l'elaborazione effettuata sull'input è la seguente:

```
for (int k = 0; k < ITERATIONS; i++) {

    for (int i = 0; i < ARRAYDIM; i++)
        for (int j = 0; j < ARRAYDIM; j++) {
            new_a[i][j] = 4 * a[i][j] -
                          (a[(j - 1 + ARRAYDIM) % ARRAYDIM][j] +
                           a[(i + 1) % ARRAYDIM][j] +
                           a[i][(j - 1 + ARRAYDIM) % ARRAYDIM] +
                           a[i][(j + 1) % ARRAYDIM]);

            /* scambio puntatori */
            swap(a, new_a);
        }

}
```

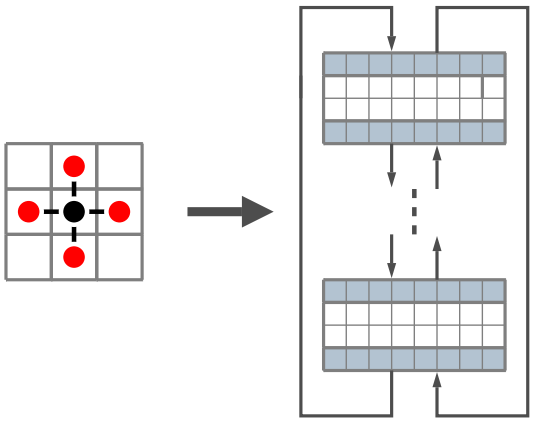

Figura 5.17: Stencil, partizionamento per righe e comunicazioni tra processi

Ai fini della valutazione delle prestazioni si considera la computazione su uno stream sufficientemente lungo di matrici. I test sono stati effettuati per matrici di dimensione 16×16, 32×32, 64×64, con un numero di iterazioni $k = 1000$.

Un generico modulo `S` (parallelo o sequenziale) che esegua la computazione precedentemente descritta avrà uno stream di matrici in ingresso e produrrà in output un altro stream di matrici della stessa dimensione.

Il tempo di interarrivo al modulo è stato valutato come il tempo di interparenza da un processo generatore il cui unico compito è inviare dati.

$$T_{A-S} = T_{P-gen} = T_{send}(sizeof(Item) * M)$$

Il tempo di interpartenza dal modulo è pari al massimo tra il tempo di interarrivo e il suo tempo di servizio.

$$T_{P-S} = max\{T_{A-S}, T_{S-S}\}$$

Nel caso `S` sia un modulo sequenziale, il suo tempo di servizio è pari al tempo di calcolo necessario per eseguire la computazione su tutta la matrice, più il tempo di servizio della `send` (cioè la parte della `send` non sovrapponibile al calcolo) per l'invio del risultato sul canale di output.

$$T_{S-S} = T_{calc}(M) + T_{send}(sizeof(Item) * M)$$

La tabella seguente mostra i risultati ottenuti nei vari test effettuati e comprende anche i tempi di interarrivo, servizio e interpartenza dal modulo sequenziale.

Dim. Matrice	T_{A-S} [a]	T_{S-S} [b]	T_{P-S} [c]
16×16	432 τ	11596180 τ	11596180 τ
32×32	844 τ	46301140 τ	46301140 τ
64×64	2384 τ	184670180 τ	184670180 τ

[a] tempo di interarrivo al modulo `S`
[b] tempo di servizio del modulo `S`
[c] tempo di interpartenza dal modulo `S`

Tabella 5.2: Test: valutazione del modulo sequenziale

I tempi ottenuti mostrano come il tempo di calcolo sia dominante rispetto alle comunicazioni: le comunicazioni sono interamente sovrapposte al calcolo eccetto per il tempo di servizio della `send`, che risulta comunque ininfluente rispetto al tempo di calcolo.

Verrà presentata ora una versione parallela dell'applicazione per elaborare lo stream di matrici in input. Ogni matrice viene distribuita in *scatter* sui VP, e il risultato viene ricostruito tramite un'operazione di *gather*. La partizione su ogni VP sarà composta da un insieme di righe della matrice. Il partizionamento a gruppi di righe comporta uno schema di comunicazione in cui il VP i-esimo comunica con il VP a nord ed il VP a sud, seguendo un andamento circolare.

L'elaborazione sui VP consiste nella ripetizione dei seguenti passi per un numero fissato di iterazioni:

1. Send della prima riga al VP a nord, e dell'ultima riga al VP a sud;
2. Calcolo sul range di righe centrale (completamente locale);
3. Receive delle righe limitrofe dal VP a nord e dal VP a sud;
4. Calcolo sulla prima e sull'ultima riga.

Le prestazioni dipendono dalla possibilità di sovrapporre le comunicazioni verso gli altri VP al calcolo successivo.

I canali di comunicazione utilizzati sono quelli basati sui segnali per la sincronizzazione tra processi, con grado di asincronia $k = 1$. Sono stati effettuati alcuni test per valutare tempi di servizio e latenze delle `send` utilizzate per lo stencil e i risultati sono riportati nella tabella seguente.

Dim. Dati	*Tempo Servizio*	*Latenza*
16 (64 Byte)	320 τ	500 τ
32 (128 Byte)	320 τ	500 τ
64 (256 Byte)	336 τ	560 τ

Tabella 5.3: Test: valutazione canali implementazione basata su segnali

I test sono stati effettuati su una Playstation 3 utilizzando un numero di VP pari a 2 e 4.

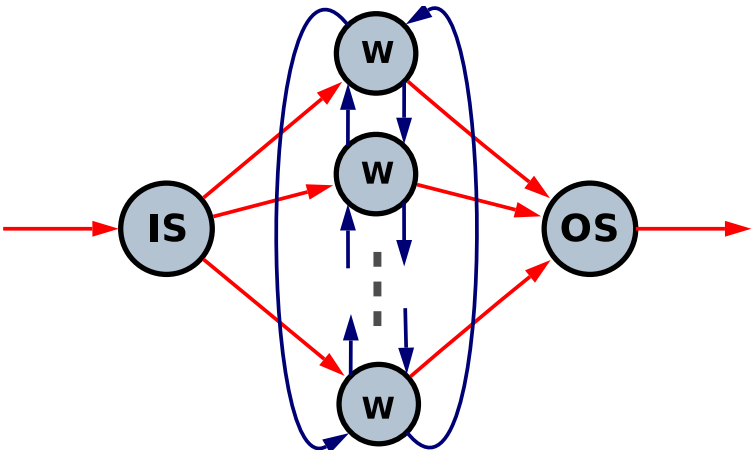

Figura 5.18: Schema del programma data parallel di test

Come visto all'inizio di questo capitolo, il tempo di servizio del modulo parallelo è pari al massimo tra i tempi di servizio delle singole sezioni e il tempo di interarrivo.

Il tempo di servizio della *Input Section* è pari al tempo necessario per scatterizzare la matrice di `M` elementi tra gli `N` VP:

$$T_{S-IS} = T_{scatter}(M, N)$$

Per vedere il modello dei costi della distribuzione *Scatter* si faccia riferimento alla tabella 5.1.

I risultati sono in linea con quelli presentati nella sezione 5.1 quindi per avere un'idea dell'andamento dei tempi al crescere dei VP e della dimensione del messaggio si può fare riferimento alle figure 5.7 e 5.8; con un unico stream di input, l'unica parte della computazione sovrapponibile alla comunicazione è il comando alternativo.

La tabella seguente offre un riepilogo delle prestazioni e comprende anche il tempo di interpartenza dalla *Input Section*.

VP	*Dim. Matrice*	T_{A-IS} [a]	T_{S-IS} [b]	T_{P-IS} [c]
2	16×16	432 τ	2859 τ	2859 τ
	32×32	844 τ	2872 τ	2872 τ
	64×64	2384 τ	3485 τ	3485 τ
4	16×16	432 τ	4478 τ	4478 τ
	32×32	844 τ	4492 τ	4492 τ
	64×64	2384 τ	4533 τ	4533 τ

[a] tempo di interarrivo alla *Input Section*
[b] tempo di servizio della *Input Section*
[c] tempo di interpartenza dalla *Input Section*

Tabella 5.4: Test: tempo di servizio della *Input Section*

Il tempo di interarrivo a un VP è pari al tempo di interpartenza dalla *Input Section*. Il tempo di servizio di un VP è invece pari al tempo di attivazione, più il tempo di calcolo e i tempi di servizio per le comunicazioni dello stencil (ogni comunicazione per lo stencil trasferisce una riga della matrice cioè $\sqrt{M}$ elementi) e la comunicazione del risultato alla *Output Section*:

$$T_{S-VP} = T_{activate} + k * T_{stencil} + T_{send}(sizeof(Item) * g)$$

con

$$T_{stencil} = 2 * T_{send}(sizeof(Item) * \sqrt{M}) + T_{calc}(g)$$

Ai fini della valutazione delle prestazioni è fondamentale verificare la sovrapposizione di tutte le comunicazioni (stencil e invio finale dei risultati alla *Output Section*) al calcolo.

La figura seguente mostra l'andamento del tempo di servizio del modulo sequenziale, e dei VP nei casi in cui la *VP Section* sia costituita da due e quattro VP. I tre punti si riferiscono ai casi delle matrici di dimensione 16×16, 32×32, 64×64.

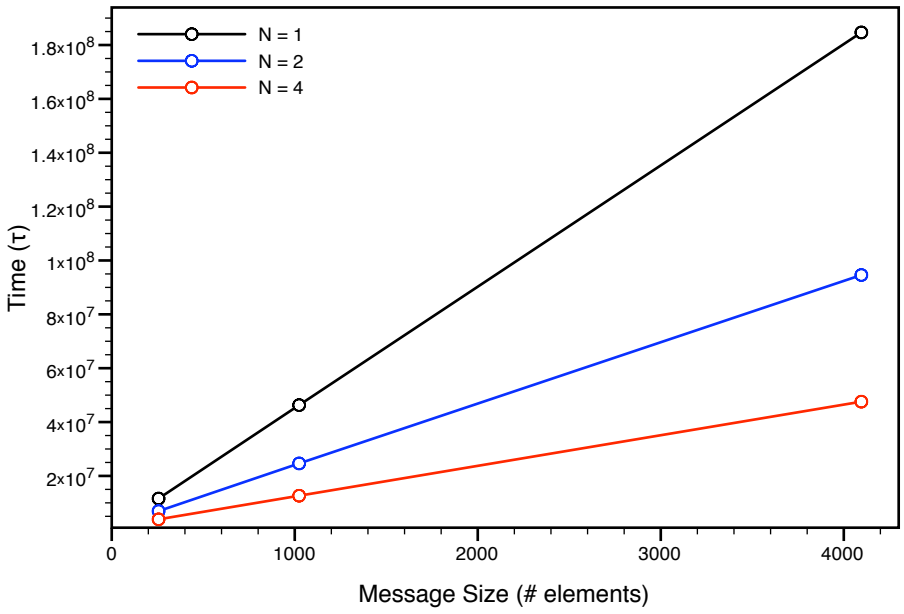

Figura 5.19: Tempo di servizio modulo sequenziale e parmod con 2 e 4 VP

Le stime effettuate tramite il modello dei costi fornito sono molto vicine ai valori misurati; la tabella seguente offre un riepilogo delle prestazioni e comprende anche il tempo di interpartenza da ogni singolo VP.

VP	*Dim. Matrice*	T_{A-VP} [a]	$T_{calc(g)}$ [b]	$T_{stencil}$ [c]	T_{S-VP} [d]	T_{P-VP} [e]
2	16×16	2859 τ	6131 τ	6940 τ	6942750 τ	6942750 τ
	32×32	2872 τ	23790 τ	24640 τ	24640706 τ	24640706 τ
	64×64	3485 τ	93702 τ	94597 τ	94597769 τ	94597769 τ
4	16×16	4478 τ	3000 τ	3886 τ	3887199 τ	3887199 τ
	32×32	4492 τ	11791 τ	12650 τ	12651232 τ	12651232 τ
	64×64	4533 τ	46642 τ	47570 τ	47570746 τ	47570746 τ

[a] tempo di interarrivo al singolo VP
[b] tempo di calcolo su g elementi
[c] tempo di un'iterazione della computazione
[d] tempo di servizio del singolo VP
[e] tempo di interpartenza dal singolo VP

Tabella 5.5: Test: tempo di servizio del singolo VP

Dalla tabella è possibile notare che il tempo di servizio di un generico VP è dominato dal tempo di calcolo e la latenza della `send` del risultato è sovrapposta al calcolo. Considerati i tempi di calcolo delle singole iterazioni della computazione, risulta che anche le comunicazioni relative allo stencil sono completamente sovrapposte al calcolo. Come al solito, per ogni comunicazione occorre considerare il tempo di servizio della `send` non sovrapponibile al calcolo (tabella 5.3).

Il tempo di interarrivo alla *Output Section* è pari al tempo di interpartenza da ogni singolo VP, in quanto la politica di collezionamento *Gather* prevede l'attivazione della *Output Section* al ricevimento di un messaggio da parte di ogni VP. Il tempo di servizio della *Output Section* è pari al tempo necessario ad effettuare l'operazione di *Gather*, che comprende il tempo di servizio della `send` per inviare la matrice risultato sul canale di output.

$$T_{S-OS} = T_{gather}(M)$$

Come discusso nella sezione 5.3, le prestazioni della politica di collezionamento *Gather* variano linearmente con la dimensione del messaggio in quanto

il fattore dominante è il costo per la copia degli elementi delle partizioni ricevute dai VP. Per avere un'idea dell'andamento dei tempi al crescere della dimensione del messaggio si può fare riferimento alla figura 5.11.

La tabella seguente offre un riepilogo delle prestazioni e comprende anche il tempo di interpartenza dalla *Output Section*.

VP	*Dim. Matrice*	T_{A-OS} [a]	T_{S-OS} [b]	T_{P-OS} [c]
2	16×16	6942750 τ	5699 τ	6942750 τ
	32×32	24640706 τ	17999 τ	24640706 τ
	64×64	94597769 τ	68689 τ	94597769 τ
4	16×16	3887199 τ	6451 τ	3887199 τ
	32×32	12651232 τ	18742 τ	12651232 τ
	64×64	47570746 τ	69156 τ	47570746 τ

[a] tempo di interarrivo alla *Output Section*
[b] tempo di servizio della *Output Section*
[c] tempo di interpartenza dalla *Output Section*

Tabella 5.6: Test: tempo di servizio della *Output Section*

È immediato notare che il tempo di servizio della *Output Section* è molto inferiore al tempo di interarrivo: il tempo di interpartenza, che sarà anche il tempo di interpartenza dal *parmod*, sarà quindi pari al tempo di interarrivo.

L'andamento dell'efficienza relativa è legato al rapporto tra costo delle comunicazioni per lo stencil e costo del calcolo interno. Al crescere della dimensione delle partizioni le comunicazioni risultano sempre meno influenti e l'effcienza relativa è quindi più alta.

I grafici seguenti mostrano l'andamento dell'efficienza relativa e della scalabilità con matrici di dimensione 16×16, 32×32, 64×64, nei casi in cui la *VP Section* sia costituita da due e quattro VP.

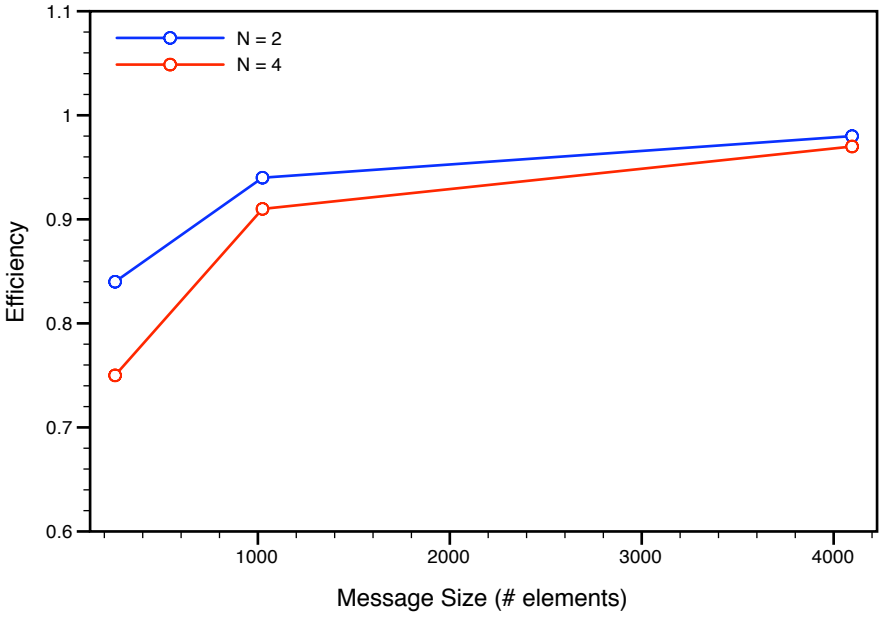

Figura 5.20: Efficienza relativa parmod con 2 e 4 VP

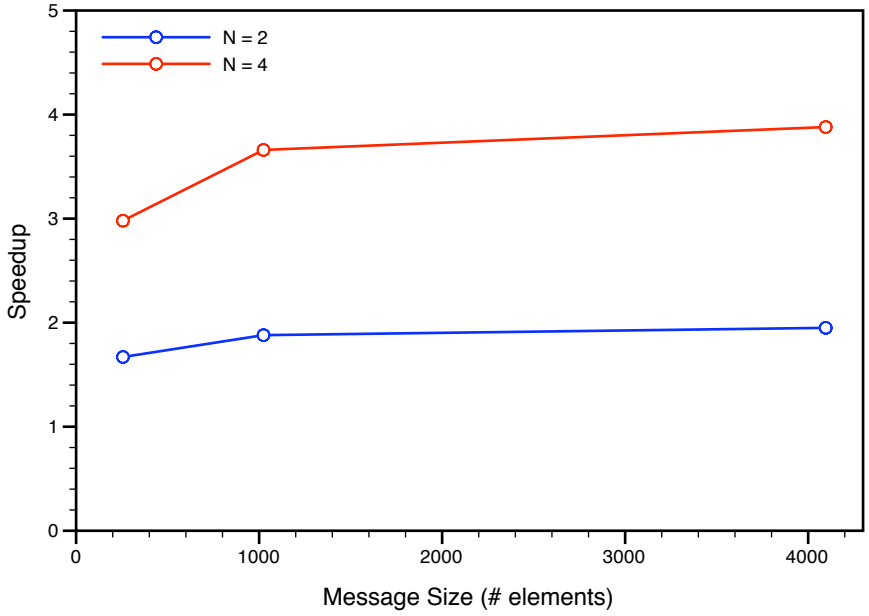

Figura 5.21: Scalabilità parmod con 2 e 4 VP

In tutti i casi testati il modulo rimane il collo di bottiglia dell'applicazione. In questa situazione è tuttavia possibile fare una valutazione dell'andamento delle prestazioni al crescere del numero di VP per trovare la grana minima del calcolo che consente ancora di sovrapporre completamente la comunicazione.

Il punto cruciale nella valutazione delle performance sta nella sovrapposizione delle comunicazioni relative allo stencil al calcolo interno dei VP. In generale per poter avere la sovrapposizione deve risultare che il calcolo della parte centrale della partizione, per il quale il VP non ha bisogno dei dati contenuti in altre partizioni, sia maggiore del tempo necessario a spedire le due righe di bordo ai due VP vicini. Dai tempi di calcolo rilevati nel caso sequenziale si ricava che il tempo medio di elaborazione su un singolo elemento della matrice è pari a circa 45.15τ; tale valore sarà indicato nel seguito con T_{calc_el}. La parte centrale di una partizione contiene tutti gli elementi della partizione ad eccezione di quelli nella prima e nell'ultima riga; il numero di tali elementi può essere calcolato nel modo seguente:

$$g_{int} = \frac{M}{N} - 2 * \sqrt{M}$$

Inoltre, il tempo di comunicazione necessario ad effettuare la `send` della prima riga al VP a nord e la `send` dell'ultima riga al VP a sud è pari a un tempo di servizio più una latenza in quanto la prima comunicazione è in parte sovrapposta alla seconda:

$$L_{com} = T_{send}(sizeof(Item) * \sqrt{M}) + L_{send}(sizeof(Item) * \sqrt{M})$$

Considerando tutte queste premesse, si può dire che la comunicazione degli stencil è sovrapposta al calcolo se è verificata la seguente:

$$T_{calc}(g_{int}) = T_{calc\_el} * g_{int} > L_{com}$$

Per quanto riguarda le performance del *parmod* la scalabilità è limitata anche dal tempo di servizio della *Output Section*: in altre parole, se il tempo di interpartenza dai VP diventa più basso del tempo di servizio della *Output*

Section, quest'ultima diventa il collo di bottiglia all'interno del *parmod*, dettando il tempo di interpartenza dal modulo parallelo. Quindi il numero di VP ideale è il più piccolo numero che verifica la seguente:

$$T_{S-VP} < T_{Gather}(M)$$

Ipotizzando di avere matrici di dimensione 64×64, si proverà a calcolare le prestazioni attese del modulo parallelo al crescere del grado di parallelismo. Con un numero di VP pari a 16 ogni partizione è composta da quattro righe e sulla *Input Section* ognuna di esse risulta allineata ai 128 Byte. Questo significa che le prestazioni della *Input Section* possono essere stimate utilizzando il modello semplificato per la distribuzione *Scatter* mostrato in 5.1.

$$T_{scatter} = 1180 + (830 * 16) = 14460\tau$$

Tale valore rappresenta quindi il tempo di interpartenza dalla *Input Section* e il tempo di interarrivo ai VP.

Ogni partizione è costituita da quattro righe quindi si avrà che

$$g = 4 * 64 = 256$$
$$g_{int} = 2 * 64 = 128$$
$$L_{com} = T_{send}(sizeof(Item)*\sqrt{M})+L_{send}(sizeof(Item)*\sqrt{M}) = (320+500)\tau$$
$$T_{calc}(g_{int}) = T_{calc\_el} * g_{int} = 45.15\tau * 128 = 5779\tau > L_{com}$$

Le comunicazioni per lo stencil sono quindi sovrapposte al calcolo interno. Il tempo di servizio di un VP è pari a:

$$T_{stencil} = 2*T_{send}(sizeof(Item)*\sqrt{M})+T_{calc}(g) = (2*320+11558)\tau = 12198\tau$$
$$T_{S-VP} = T_{activate} + k * T_{stencil} + T_{send}(sizeof(Item) * g) =$$
$$(100 + 1000 * 12198 + 432)\tau = 12198532\tau > T_{Gather}(M)$$

che coincide anche col tempo di interpartenza dal modulo parallelo.

L'efficienza relativa è pari a 0.95, la scalabilità è pari a 15.14. Rispetto al caso in cui la *VP Section* era costituita da quattro VP l'efficienza relativa è più bassa perchè la partizione è più piccola e le comunicazioni relative allo stencil incidono maggiormente sul tempo di servizio complessivo.

Conclusioni e sviluppi futuri

Questo lavoro di tesi introduce il supporto al costrutto *parmod* di ASSIST per architetture Cell, mediante un approccio a libreria. L'implementazione è basata su MammuT, libreria di comunicazione per Cell sviluppata presso il Laboratorio di Architetture Parallele del Dipartimento di Informatica dell'Università di Pisa. La libreria MammuT implementa un modello di programmazione parallela indipendente dall'architettura, che si ispira al linguaggio LC presentato in [8] e descritto brevemente nella sezione 2.1.

Una prima parte del lavoro svolto ha riguardato l'estensione di MammuT con l'introduzione dei costrutti *Array di processi* e *Array di canali*. È stato introdotto inoltre il concetto di grafo dell'applicazione, per rappresentare in modo univoco i processi e i canali di un'applicazione parallela: il grafo viene definito una volta sola dal programmatore e viene poi incluso da tutti i processi dell'applicazione. Queste estensioni di MammuT hanno consentito di semplificare la fase di inizializzazione della libreria tramite l'automatizzazione di molte delle operazioni di registrazione che prima dovevano essere eseguite esplicitamente dal programma utente.

La seconda parte del lavoro ha riguardato l'implementazione della *Input Section* del *parmod* di ASSIST, partendo dalla versione sviluppata da Silvia Lametti durante il suo tirocinio presso il Laboratorio di Architetture Parallele.

L'ultima parte del lavoro ha riguardato infine l'implementazione di *VP Section* e *Output Section* per completare il supporto al costrutto *parmod*.

Durante la progettazione e lo sviluppo si è posta l'enfasi soprattutto sulla flessibilità, ma con un occhio di riguardo alle prestazioni e alla semplicità di utilizzo. Esempi di flessibilità sono le fasi di pre-elaborazione e post-elaborazione sui dati personalizzabili dal programmatore oppure la possibilità di cambiare a programma le politiche di distribuzione sulla *Input Section* o quelle di collezionamento sulla *Output Section*. I processi della *VP Section* sono programmati come normali processi LC utilizzando la libreria MammuT, tranne per quanto riguarda l'attivazione, che viene gestita dal supporto del livello ASSIST.

Per quanto riguarda le prestazioni i test effettuati hanno dimostrato che il nuovo livello di astrazione ha introdotto dei ritardi abbastanza ragionevoli stimati nel caso medio tra i 1000 e i 1500 cicli di clock.

L'obiettivo della flessibilità è importante in ottica futura in vista della realizzazione di un compilatore che a partire da un programma scritto nel linguaggio di coordinamento ASSIST-CL produca un programma C++ che utilizzi la libreria MammuT e il supporto per il *parmod* realizzati.

Un altro obiettivo interessante per il futuro consiste nella realizzazione del supporto per le comunicazioni inter-chip in modo da poter lavorare su un cluster di Cell in maniera completamente trasparente.

Ci sono poi una serie di piccole feature che potrebbero essere implementate come ulteriori estensioni di MammuT. Tra queste vale la pena menzionare il supporto a messaggi di dimensione maggiore di 16 KB oppure la possibilità di allocare "a mano" le variabili targa, passando poi ai canali gli indirizzi delle variabili targa da utilizzare. Attualmente l'allocazione delle variabili targa avviene in maniera automatica all'interno dei canali stessi, ovviamente solo lato ricevente. Ci si potrebbe chiedere che vantaggi possa portare una simile feature rispetto all'allocazione automatica fatta all'interno dei canali. Per trovare la risposta basta pensare un attimo alla politica di collezionamento *Gather* e ai ritardi dovuti alle copie degli elementi per ricomporre il messaggio: dando al programmatore la possibilità di allocare le variabili targa

dei canali sarebbe possibile avere una *Gather* senza copie in quanto ogni VP depositerebbe la propria partizione nella posizione corretta all'interno del messaggio da spedire sullo stream di output.

Bibliografia

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Communications of the ACM, 1985.

[2] J. Darlington, M. Ghanem, H.W. To. *Structured parallel programming*. Programming Models for Massively Parallel Computers, 1993.

[3] Marco Vanneschi. *The programming model of ASSIST, an environment for parallel and distributed portable applications*. Parallel Computing, 2002.

[4] M. Aldinucci, M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, C. Zoccolo. *High level grid programming with ASSIST*. Computational Methods in Science and Technology, 2006.

[5] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, C. Zoccolo. *ASSIST as a research framework for high-performance Grid programming environments*. Grid Computing: Software environments and Tools, 2006.

[6] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, C. Zoccolo. *Structured implementation of component based grid programming environments*. Future Generation Grids, 2005.

[7] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo. *Components for high performance Grid programming in Grid.it*. Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications, 2005.

[8] Marco Vanneschi. *Architetture Parallele e Distribuite*. SEU, 2008.

[9] Alessandro Piras. *Meccanismi di comunicazione per architetture multicore (CELL)*. Tesi di laurea, 2008.

[10] Silvia Lametti. *Libreria di programmazione parallela strutturata per multiprocessor-on-chip*. Tesi di laurea, 2008.

[11] M. Kistler, M. Perrone, F. Petrini. *Cell Multiprocessor Communication Network: Built for Speed*. IEEE MICRO, 2006.

[12] T. Chen, R. Raghavan, J.N. Dale, E. Iwata. *Cell Broadband Engine Architecture and its first implementation - A performance view*. IBM, 2007.

[13] T.W. Ainsworth, T.M. Pinkston. *Characterizing the Cell EIB On-Chip Network*. Micro, IEEE, 2007.

[14] C.H. Crawford, P. Henning, M. Kistler, C. Wright. *Accelerating Computing with the Cell Broadband Engine Processor*. CF '08: Proceedings of the 2008 conference on Computing frontiers, 2008.

[15] A. Buttari, J. Dongarra, J. Kurzak. *Limitations of the PlayStation 3 for High Performance Cluster Computing*. Technical Report UT-CS-07-597, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007.

[16] M. McCool. *Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform*. GSPx Multicore Applications Conference, 2006.

[17] M. Alind, M. Eriksson, C. Kessler. *BlockLib: A Skeleton Library for Cell Broadband Engine*. IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering, 2008.

[18] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, T. Nakatani. *MPI microtask for programming the Cell Broadband Engine processor*. IBM Systems Journal, 2006.

[19] T. Chen, H. Lin, T. Zhang. *Orchestrating Data Transfer for the Cell/B.E. Processor*. ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, 2008.

[20] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu. *Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture*. IBM Systems Journal, 2006.

[21] IBM. *Cell Broadband Engine Programming Handbook*. IBM, 2007.

[22] IBM. *Programming the Cell Broadband Engine Architecture - Examples and Best Practices*. IBM, 2008.

[23] IBM. *CBEA JSRE Series - SPE Runtime Management Library Version 2.3*. IBM, 2008.