Technical Report: TR-95-14, October 1, 1995

L'Ambiente di Programmazione P3L nel sistema di supporto PVM su Rete di Calcolatori Eterogenei

Bacci, Bruno Cantalupo, Barbara Guerrini, Nicola Pelagatti, Susanna.

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

In this document, we discuss a prototype compiler of P3L, which is a structured parallel programming language, on a network of heterogeneous workstations. P3L is implemented on top of the abstract parallel machine provided by PVM. In particular, we discuss the definition of a new set of implementation templates, which are designed in order to meet the target features, and we show how the correct configurations for the abstract machine can be automatically generated within the support. (The document is written in Italian.)

ACM–CR Subject Classification:

Keyword and phrases: Skeletons, P3L, PVM, Workstation Network

UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI INFORMATICA

Technical Report: TR- 14/95

L'Ambiente di Programmazione P^3L nel Sistema di Supporto PVM su Rete di Calcolatori Eterogenei

Bruno Bacci, Barbara Cantalupo, Nicola Guerrini & Susanna Pelagatti

Abstract. In questo documento vengono illustrate le tematiche relative alla implementazione del compilatore del P^3L, un linguaggio di programmazione parallela strutturata pensato per realizzare un efficiente portabilità del codice su macchine di tipo DM-MIMD. In particolare vengono analizzate le problematiche, e le relative soluzioni, incontrate in un primo approccio alla implementazione del compilatore per una rete di macchine eterogenee, realizzata attraverso la mediazione di una macchina astratta intermedia definita utilizzando il sistema PVM (un ambiente di sviluppo di applicazioni parallele su reti eterogenee). Più precisamente si discute della definizione di nuovi *template di implementazione* che tengano conto delle particolarità della architettura target considerata e delle funzionalità di cui è stato necessario dotare il compilatore affinché potesse generare un adeguato codice eseguibile, sia in termini di configurazione della macchina astratta che di analisi e generazione delle informazioni necessarie al supporto a tempo di esecuzione. Poiché tale lavoro è tuttora in fase di completamento vengono infine indicate le ulteriori direzioni di sviluppo per l'adozione di eventuali ottimizzazioni da apportare sulla base di studi analitici e/o di effettive misurazioni di tempo.

Keywords. Skeleton Based Parallel Programming, P3L, PVM, Workstation Network.

October 1995

ADDR:Corso Italia 40,56125 Pisa,Italy. TEL:+39-50-887000 - FAX:+39-50-887226
E_MAIL: p3l@di.unipi.it

Indice

1 Introduzione

L'esigenza di una sempre più elevata potenza di calcolo in vari settori applicativi, non soltanto scientifici, unita alla forte spinta tecnologica degli ultimi anni, ha determinato una rapida diffusione di differenti tipi di architetture e modelli di computazione paralleli. Un argomento di ricerca tuttora aperto rimane comunque la definizione di una metodologia di programmazione parallela che unisca alla semplicità d'uso un'efficiente portabilità dei programmi, intesa come la capacità di produrre del codice eseguibile su macchine e/o configurazioni differenti ottenendo allo stesso tempo un adeguato livello di prestazioni.

Il linguaggio di programmazione parallela P^{3}L (*Pisa Parallel Programming Language*) nasce nell'ambito della definizione di una nuova metodologia di programmazione che si propone di raggiungere tale obiettivo affiancando ad un linguaggio ad alto livello, strutturato, esplicitamente parallelo, P^{3}L appunto, un insieme di strumenti statici che automaticamente adattino le caratteristiche strutturali dei programmi alla particolare architettura target considerata. In particolare P^{3}L punta ad ottenere una facile ed efficiente programmabilità su macchine appartenenti alla classe delle architetture MIMD general purpose a memoria distribuita (DM-MIMD). Fino a questo momento lo sviluppo della metodologia si è concentrato essenzialmente sulla programmabilità di applicazioni a *parallelismo massiccio* e, di conseguenza, l'architettura target è stata ristretta a quelle macchine che potessero garantire *scalabilità* e assenza di *colli di bottiglia*. A tale scopo, è stata considerata quale architettura target la classe dei multiprocessori con rete di interconnessione *k-ary n-cube* che include molte interessanti macchine MIMD quali ad esempio i mesh bi-dimensionali e tri-dimensionali o gli ipercubi.

In questo documento descriveremo, analizzandone le problematiche connesse, un primo approccio all'implementazione del compilatore del linguaggio P^{3}L per un'architettura target costituita da una rete di macchine eterogenee. Una tale implementazione costituisce da un lato un'interessante fonte di studio per valutare l'efficienza della metodologia in un ambito sotto molti aspetti differente da quello fino ad ora considerato, dall'altro rappresenta un notevole passo in avanti, se non nella efficienza sicuramente nella portabilità delle applicazioni P^{3}L. Una generica rete eterogenea di macchine è, infatti, un'architettura estremamente diffusa e può fornire, se dotata dei necessari strumenti software, una piattaforma di programmazione parallela a basso costo.

A queste considerazioni di carattere generale va aggiunta la consapevolezza della notevole diffusione che sta ottenendo un sistema per lo sviluppo di applicazioni "portabili" su reti di calcolatori eterogenei denominato PVM (*Parallel Virtual Machine*). Tale sistema consente di considerare una rete eterogenea di macchine UNIX come un'unica risorsa computazionale, supportando l'eterogeneità non solo delle risorse di calcolo ma anche delle reti di comunicazione. In particolare l'interfaccia *message-passing* della *macchina virtuale* che PVM permette di definire è costituita da circa 70 routine invocabili da programmi C o Fortran. Grazie alla particolare classe di architetture considerata, ampiamente utilizzata ed estremamente generale, e nonostante tale classe non sia, di conseguenza, finalizzata all'elaborazione di computazioni parallele, il sistema PVM può essere considerato, in senso lato, la *macchina parallela* attualmente più diffusa. In quest'ottica l'uso di PVM per virtualizzare la macchina astratta, attraverso la quale il compilatore del P^{3}L si interfaccia con la classe di architetture target, costituisce non solo uno strumento estremamente utile dal punto di vista implementativo ma anche un importante veicolo per un più ampio utilizzo del linguaggio stesso.

È evidente che in questo ambito ciascun passo della metodologia deve essere attentamente valutato e verificato. Ad esempio non si può non tenere conto dei valori, tipici dell'architettura considerata ma insolitamente alti in ambito parallelo, del rapporto "costi di comunicazione/costi di computazione". Per tale motivo il lavoro di implementazione del compilatore è stato articolato in due fasi. Nella prima fase, che ci accingiamo a documentare, è stata effettuata una prima implementazione del compilatore sulla base delle precedenti esperienze e di ragionevoli considerazioni. Nella seconda che, costituisce l'ulteriore direzione di sviluppo del lavoro descritto, mediante l'analisi dei modelli di implementazione realizzati e in considerazione di misure reali dei tempi di elaborazione e comunicazione relativi alla nuova implementazione, si definiranno le eventuali ottimizzazioni da apportare al compilatore.

Nel seguito descriveremo essenziamente le problematiche, e le relative soluzioni, incontrate sia al livello di definizione dei modelli di implementazione relativi ai vari costrutti paralleli (definizione dei *template di implementazione*), sia, ad un livello di implementazione più basso, nella generazione delle informazioni necessarie al supporto a tempo di esecuzione (modifiche da apportare al *back_end*). A tale proposito è bene anticipare che alcune sezioni costituiscono la documentazione dettagliata delle soluzioni adottate per i principali problemi di implementazione incontrati e dunque costituiscono una lettura rivolta essenzialmente a coloro che vogliano addentrarsi in una approfondita conoscenza dell'implementazione delle librerie. Ci preme ribadire che tale lavoro è tuttora in fase di completamento e, dunque, questo documento non costituisce un rapporto finale sugli esiti della sua realizzazione ma vuole essere un primo momento di sintesi e riflessione da cui derivare ulteriori spunti di sviluppo ed ottimizzazione.

Il documento si presenta organizzato come segue.

- nella sezione **2** introdurremo le caratteristiche principali del linguaggio P^{3}L ed in particolare quelle caratteristiche del supporto che ne garantiscono la portabilità;
- nella sezione **3** faremo una breve descrizione del sistema di sviluppo PVM concentrandoci sulla configurazione della macchina astratta e sui relativi meccanismi che PVM supporta e che noi abbiamo utilizzato nella nostra implementazione;
- nella sezione **4** descriveremo i *template di implementazione* definiti per la particolare architettura target considerata;

- nella sezione **5** mostreremo le principali modifiche apportate al *back_end* del compilatore per permettere la generazione di un nuovo supporto a tempo di esecuzione;
- nella sezione **6** descriveremo le istruzioni necessarie per l'utilizzo del compilatore P^{3}L, sia in termini di configurazione dell'ambiente di programmazione sia in termini di compilazione ed esecuzione vera e propria del programma;
- nella sezione **7** concluderemo la trattazione indicando gli ulteriori sviluppi del lavoro e le prime ottimizzazioni che andremo ad apportarvi.

2 Il linguaggio P^{3}L e il suo supporto

Il linguaggio di programmazione **P^{3}L** (*Pisa Parallel Programming Language*) è un linguaggio parallelo strutturato, sviluppato presso il Dipartimento di Informatica dell'Università degli Studi di Pisa nell'ambito del progetto P4 (*Pisa Parallel Programming Project*). Esso nasce in accordo con lo sviluppo di una nuova metodologia di programmazione parallela che si pone come obiettivo di garantire una facile programmabilità e una efficiente portabilità delle applicazioni su differenti macchine appartenenti alla classe delle architetture DM-MIMD ([DDO+92, BDO+95]).

Tale metodologia si basa fondamentalmente su:

- il linguaggio strutturato ed esplicitamente parallelo P^{3}L;
- un insieme di strumenti di compilazione che, esplicitando le caratteristiche strutturate del linguaggio, realizzano l'efficiente portabilità delle applicazioni;
- una macchina astratta che astrae le caratteristiche ed i meccanismi relativi alla macchina fisica sottostante.

L'aspetto caratterizzante del linguaggio P^{3}L consiste nella possibilità di descrivere esplicitamente il parallelismo presente in una applicazione attraverso la composizione gerarchica di costrutti, corrispondenti ciascuno ad una particolare forma di parallelismo (linguaggio parallelo *strutturato*).

La scelta di definire dei costrutti paralleli come primitive del linguaggio deriva dalla considerazione che le applicazioni parallele, ed in particolare quelle che esplicitano *parallelismo massiccio*, utilizzano un numero limitato di paradigmi di parallelismo, caratterizzati da una struttura regolare sia di distribuzione dei dati che di interconnessione tra processi ([Hey89, Kun88]). D'altra parte, la limitazione della struttura computazionale dei programmi pur comportando una perdita di generalità permette, però, agli strumenti di compilazione di sfruttare la conoscenza delle caratteristiche specifiche del modello di computazione espresso per ottimizzare l'implementazione del programma. Alla base di questo approccio vi è la convinzione che l'unico modo di risolvere il problema del mapping, e dunque in buona parte quello implementativo, con delle assicurazioni sull'accuratezza del risultato è quello di sviluppare algoritmi per architetture con una topologia fissata e per programmi *la cui struttura di interazione appartenga ad una classe fissata* ([Pel93]).

È evidente che le premesse che hanno portato alla definizione di P^{3}L conservano il loro valore e significato anche in un ambito più ampio che non il parallelismo massiccio. Si tratterà eventualmente di aggiungere ulteriori costrutti che esplicitino forme di parallelismo con caratteristiche più affini al tipo di architettura considerata o forme di controllo che rispondano ad esigenze diverse.

Allo stato attuale P^{3}L include, tra gli altri, i costrutti corrispondenti ad alcune delle forme di parallelismo più diffuse:

- il costrutto **sequential**, che corrisponde ad un semplice processo sequenziale ed include il codice scritto dall'utente in un linguaggio di programmazione ad alto livello, denominato linguaggio *host* (nella versione attuale, il linguaggio C o Fortran);
- il costrutto **pipe**, che modella il parallelismo *pipeline*;
- il costrutto **farm**, che modella il parallelismo di tipo *processor farm*;
- il costrutto **map**, che modella una particolare forma di computazione *data parallel*;
- il costrutto **loop**, che modella computazioni *iterative*.

Per ciascuno dei costrutti paralleli e relativamente alla macchina target considerata, il compilatore del P^{3}L dispone di differenti **template di implementazione** che definiscono le informazioni relative all'insieme dei processi che realizzano l'implementazione di ciascun costrutto. Essi si compongono di **template dei processi**, che definiscono il codice dei singoli processi, parametrizzato rispetto ai tipi, alle macro e a tutte quelle informazioni tipiche della particolare istanza considerata, e di **template di mapping**, che ne definiscono la struttura topologica. In definitiva ad ogni costrutto viene associata una rete parametrica di processi, il cui codice è definito nei corrispondenti *template dei processi*, che descrive il mapping dei processi e dei canali di comunicazione sulla macchina astratta. Ad ogni costrutto parallelo possono essere associati più template che risultano diversi tra loro per caratteristiche topologiche ma implementano il medesimo schema di interazione tra processi. Il compilatore è in grado di determinare per ciascuna implementazione i parametri opportuni, quali ad esempio il grado di parallelismo o la granularità dei processi, sulla base delle caratteristiche della particolare istanza del costrutto da implementare e della conoscenza di **formule analitiche di performance** associate ai vari template.

Schematicamente le funzionalità implementate dal compilatore possono essere distinte in tre fasi, cui corrispondono altrettanti elementi del compilatore stesso:

- il **front-end**, che effettua il parsing del codice sorgente e ne costruisce una rappresentazione interna, in forma di *albero dei costrutti*, che ne evidenzia la struttura parallela; ciascun nodo dell'albero è etichettato con informazioni riguardanti il passaggio dei parametri tra i costrutti composti e, nel caso dei costrutti `sequential`, con il nome dei file e delle corrispondenti procedure contenenti il codice sequenziale scritto dall'utente;

- il **middle-end**, che con l'elaborazione dell'albero dei costrutti produce una nuova rappresentazione interna contenente informazioni relative al nome dei processi che implementano l'applicazione, al *mapping* dei processi sull'architettura della macchina astratta, ai tipi dei dati scambiati nelle comunicazioni. Il *middle-end* definisce il mapping logico dei costrutti sulla macchina astratta accedendo ai template di mapping contenuti in una *libreria di mapping*. Ciascun elemento della libreria è associato a delle formule di performance parametrizzate rispetto ai costi dei meccanismi esportati dalla macchina astratta ed utilizzate per l'ottimizzazione di ciascuna implementazione.

- il **back-end**, che utilizzando le strutture di mapping definite dal *middle-end* genera il codice eseguibile per la macchina astratta. Questo viene effettuato definendo il mapping fisico dei processi e dei canali di comunicazione sulla macchina astratta, utilizzando i meccanismi di interfaccia che essa fornisce. In particolare il codice eseguibile dei singoli processi viene generato facendo uso di processi predefiniti contenuti nella *libreria dei template di processo*, e aggiungendo a ciascuno di questi le informazioni riguardanti la trasmissione dei dati (ad esempio i tipi dei dati o l'identificatore del processo mittente e del destinatario) ed alcune funzioni speciali (ad esempio partizionamento dei dati, impacchettamento e spacchettamento dei buffer di comunicazione) relative alle particolari istanze da implementare.

È interessante notare come la mediazione della macchina astratta da un lato, e l'utilizzo di librerie, dall'altro, permettano di ottenere una notevole indipendenza dalla architettura target considerata. In quest'ottica la variazione dell'architettura target comporta essenzialmente una variazione delle *librerie di mapping* e *dei template di processo* e di alcune funzionalità del back_end che è in più stretto contatto con la struttura della macchina astratta ed i suoi meccanismi.

Allo stato attuale esiste una versione del compilatore che produce codice per una macchina astratta *mesh-based* che esporta un insieme di meccanismi estremamente ridotto. Il codice finale è eseguibile su un emulatore di architetture parallele e su una reale macchina parallela, la `Meiko CS1` basata su `transputer T800`. L'implementazione di una versione del compilatore P^3L, per una macchina astratta di calcolatori eterogenei connessi da una rete uniforme e virtualizzata dal sistema PVM, consiste essenzialmente nella definizione di nuove librerie di *template di mapping* (con relative formule di performance e ottimizzazioni) e di *template di processi* e nella modifica della implementazione delle funzionalità del *back_end*, in dipendenza della nuova struttura della macchina astratta e dei meccanismi che essa esporta.

3 L'architettura target e l'ambiente di sviluppo PVM

Abbiamo visto che P^{3}L è un linguaggio nato con l'obiettivo di programmare facilmente applicazioni a parallelismo massiccio ma l'implementazione di una versione del compilatore P^{3}L, per una rete eterogenea di macchine seriali e/o parallele, costituisce un interessante oggetto di studio. Esso permette infatti di verificare il comportamento dei programmi in un ambiente sotto molti aspetti differente da quello inizialmente considerato e contribuisce quindi all'estensione della portabilità delle applicazioni P^{3}L. È evidente infatti che una generica rete eterogenea di macchine può considerarsi un'architettura estremamente diffusa e può fornire, se dotata dei necessari strumenti software, una piattaforma di programmazione parallela a basso costo, in grado di gestire problemi di dimensioni anche notevoli.

La classe di architetture cui faremo riferimento presenta le seguenti caratteristiche generali:

- *nodi di elaborazione eterogenei*,
- *rete di interconnessione uniforme*,
- *memoria distribuita*,
- *controllo distribuito*.

Per garantire la portabilità dei programmi e in accordo con lo spirito della metodologia sviluppata intorno a P^{3}L, la classe di architteture considerata viene vista dal compilatore del linguaggio attraverso una macchina astratta che esporta dei meccanismi di interfaccia uniformi per ciascun membro della classe. Tali meccanismi non sono utilizzati a livello delle applicazioni ma vengono esplicitati dai template di implementazione dei vari costrutti che il compilatore utilizza per produrre il codice oggetto finale. La macchina astratta cui facciamo riferimento è virtualizzata da un sistema software chiamato **PVM** (Parallel Virtual Machine) che permette di considerare una rete eterogenea di macchine UNIX come un'unica risorsa computazionale parallela.

Nel seguito del paragrafo ci limiteremo a dare alcune informazioni di carattere generale riguardanti tale sistema e ci soffermeremo su alcuni dei meccanismi principali del controllo dei processi e delle comunicazioni che il sistema supporta (facendo riferimento alla versione più recente al momento dello sviluppo del nostro codice, ovvero **pvm3.3.7**). È evidente che questa non vuole essere una descrizione dettagliata del sistema PVM, per la quale rimandiamo alla bibliografia ([GBD$^+$94, Sun90, SGDM94]), ma una semplice introduzione informale alle funzionalità cui siamo ricorsi per l'implementazione del nostro compilatore.

3.1 Generalità del sistema

Il sistema PVM, ideato e sviluppato presso l'Oak Ridge National Laboratory, è attualmente ampiamente diffuso (circa 9000 copie rilasciate) ed è stato adottato da alcuni fornitori come interfaccia *message-passing* per le loro piattaforme (`CRAY T3D`, `IBM SP1/SP2`, `CONVEX`

PVM_ARCH	Macchina	PVM_ARCH	Macchina
AFX8	Alliant FX/8	KSR1	Kendall Square KSR-1
ALPHA	DEC Alpha	LINUX	80386/486 LINUX box
BAL	Sequent Balance	MASPAR	MASPAR host
BFLY	BBN Butterfly TC2000	MIPS	MIPS 4680
BSD386	80386/486 Unix box	NEXT	NeXT
CM2	Thinking Machine CM2	PGON	Intel Paragon
CM5	Thinking Machine CM5	PMAX	DECstation 3100, 5100
CNVX	Convex C-series	RS6K	IBM/RS6000
CNVXN	Convex C-series	RT	IBM RT
CRAY	C-90, YMP, Cray-2	SGI	Silicon Graphics IRIS
CRAY2	Cray-2	SGI5	Silicon Graphics IRIS
CRAYSMP	Cray S-MP	SUN3	Sun 3
DGAV	Data General Avion	SUN4	Sun 4, SPARCstation
E88K	Encore 88000	SUN4SOL2	Sun 4, SPARCstation
HP300	HP-9000 model 300	SYMM	Sequent Symmetry
HPPA	HP-9000 PA-RISC	U370	IBM 370
I860	Intel iPSC/860	TITN	Stardent Titan
IPSC2	Intel iPSC/2 386 host	UVAX	DEC Micro VAX

Tabella 1. Architetture testate con PVM 3 e relativi nomi utilizzati (PVM_ARCH).

`MP`). Esso permette di interpretare un insieme, definito dall'utente, di macchine di tipo differente (seriali, parallele, vettoriali) come un'unica macchina a memoria distribuita la cui struttura logica viene indicata con il nome di *macchina virtuale*. Ciascuno degli elementi che compongono la macchina virtuale viene indicato con il termine *host*.

PVM fornisce le funzioni per attivare automaticamente dei *task*, intesi come unità computazionali analoghe ai processi UNIX, sulla macchina virtuale e per permettere loro la reciproca comunicazione e/o sincronizzazione. Le applicazioni possono essere scritte in Fortran o C e possono essere parallelizzate usando i più comuni costrutti **message-passing**. Il sistema supporta l'eterogeneità sia al livello di macchine che a livello di rete. Ovvero è in grado di effettuare tutte le conversioni di dati necessarie tra macchine (incluse tra quelle su cui PVM è funzionante, vedi tab.1) che eventualmente utilizzano differenti rappresentazioni dei tipi e permette alla macchina virtuale di essere interconnessa mediante differenti reti di comunicazione (`MPS`, `Ethernet`, `Token Ring`).

PVM si compone essenzialmente di due parti: un insieme di processi **daemon**, denominati *pvmd3* (o più semplicemente *pvmd*), che risiedono ciascuno su di un host della macchina virtuale e una libreria di circa 70 routine C e Fortran che esplicitano le funzionalità di controllo dei processi, di comunicazione e sincronizzazione, di configurazione della macchina virtuale, ecc.. Essendo il C il linguaggio sequenziale di base utilizzato da P^3L, (sez.2) nel seguito faremo riferimento esclusivamente alle routine della libreria corrispondente.

3.2 Configurazione della Macchina Virtuale

È possibile configurare, e successivamente attivare, il sistema PVM secondo due diverse modalità: attivando la console di PVM (`pvm`), un monitor di comandi interattivo, e aggiungendo manualmente gli host appartenenti alla configurazione desiderata oppure, automaticamente, eseguendo il comando `pvmd3` con un *hostfile*.

Mentre, ai nostri fini, la console risulta un'utile strumento di controllo dell'esecuzione del programma, l'*hostfile*, essendo un vero e proprio file di configurazione della macchina virtuale, è fondamentale per l'esecuzione automatica delle applicazioni sulla macchina virtuale. Esso contiene la lista dei nomi degli host che costituiscono la macchina virtuale ma anche di quelli che eventualmente l'utente (nel nostro caso il compilatore) vorrà aggiungere in seguito. Oltre all'indicazione dell'host su cui PVM sarà inizialmente attivato (il primo della lista), per ciascun host è possibile eventualmente specificare, mediante le opzioni disponibili, indicazioni relative all'accesso ad una macchina specifica (informazioni di login o di password) o alla collocazione dei files che dovranno essere eseguiti su di essa quali il pvmd3 (*dx*) o i files eseguibili corrispondenti all'applicazione (*ep*) e la relativa directory di lavoro (*wd*).

Osserviamo infine che PVM supporta anche delle routine di controllo della macchina virtuale che permettono l'eliminazione o l'aggiunta di ulteriori host, delle quali attualmente non abbiamo fatto uso ma che potrebbero risultare utili in una successiva fase di ottimizzazione.

3.3 Controllo dei processi

PVM dispone di routine che permettono ad un processo utente di divenire un task PVM e, viceversa, di ritornare di nuovo un normale processo UNIX.

Mediante la routine `pvm_spawn (task, argv, flag, where, ntask, tids)` è possibile attivare *ntask* copie del file eseguibile *task* sulla macchina virtuale, specificando eventualmente un vettore *argv* di argomenti.

Assegnando un opportuno valore al campo *flag* è possibile effettuare:

- un **mapping automatico**, affidando a PVM la scelta dell'host su cui attivare i processi in base alle misure del carico delle macchine;
- un **mapping esplicito**, indicando nel campo *where* l'host su cui attivare i processi;
- un **mapping dipendente dall'architettura**, affidando a PVM la scelta dell'host su cui attivare i processi, in funzione del tipo di architettura specificata nel campo *where*.

Una volta attivati gli *ntask* processi, la routine restituisce il vettore *tids* contenente gli identificatori corrispondenti a ciascuno di essi o eventuali codici di errori se l'attivazione non ha avuto successo. In particolare i **tid** sono numeri interi mediante i quali ciascun *task* PVM è identificato univocamente all'interno dell'intera macchina virtuale. Per garantirne l'unicità essi vengono generati dal pvmd locale e non possono essere assegnati dall'utente. Oltre alla `pvm_spawn()` anche altre routine di PVM restituiscono come valori i *tid* dei processi, permettendo dunque a tutti i processi la reciproca identificazione. Ad esempio le routine `pvm_mytid()` e `pvm_parent()` restituiscono rispettivamente il tid del processo chiamante e del processo che ha attivato il chiamante.

In PVM non esistono routine di terminazione dei processi ma ad un processo è consentito lasciare il sistema comunicandolo al pvmd locale mediante la routine `pvm_exit()`. Il processo può quindi continuare la sua elaborazione come un qualsiasi processo UNIX o terminare normalmente mediante la funzione `exit()`.

3.4 Modalità di comunicazione

Il modello di comunicazione cui PVM fa riferimento prevede che un qualsiasi task possa comunicare con un qualunque altro senza alcun limite sulla grandezza o sul numero dei messaggi. Le limitazioni di memoria risultano dunque legate esclusivamente alle limitazioni fisiche dello spazio del buffer potenziale su ciascuna particolare macchina considerata.

I protocolli di invio e ricezione dei messaggi all'interno del modello possono essere schematizzati come segue:

INVIO:	**RICEZIONE:**
i_1 `* predisposizione di un buffer di invio;`	r_1 `* ricezione del messaggio;`
i_2 `*` *impacchettamento* `dei dati nel buffer;`	r_2 `*` *spacchettamento* `dei dati nel buffer;`
i_3 `* invio del messaggio;`	

Figura 1. Protocollo di invio e ricezione di messaggi

Per poter realizzare la comunicazione tra macchine differenti, PVM utilizza uno schema di codifica dei dati (XDR) che permette di superare le eventuali differenze nella rappresentazione dei dati stessi. Per realizzare tale conversione è necessario che i dati siano *impacchettati* prima del loro invio e *spacchettati* al momento della ricezione del messaggio.

PVM permette di gestire buffer di comunicazione multipli tra i quali, però, ad ogni istante, uno solo può essere considerato *attivo*. Il buffer di invio attivo viene ripulito e preparato per l'impacchettamento di un nuovo messaggio mediante la routine `pvm_initsend(encoding)` in cui è possibile specificare il tipo di opzione di codifica, *encoding*, da utilizzare per l'*impacchettamento*[1].

[1]È possibile, ad esempio, richiedere che la codifica non sia effettuata, nel caso in cui il messaggio dovrà

Per impacchettare un singolo messaggio è necessario invocare una routine di **impacchettamento** per ciascun vettore di dati di tipo differente (quali ad esempio *byte*, *float*, *int*, *double* ecc.). Per ciascun tipo di dato *type* è definita, infatti, una routine `pktype (type, nitem, stride)` che impacchetta un vettore di *nitem* elementi del tipo *type* con un passo pari a *stride*. Non ci sono limiti alla complessità del messaggio da *impacchettare* ma è necessario che esso sia *spacchettato* esattamente come è stato *impacchettato*.

Per quel che riguarda più strettamente la fase di comunicazione vera e propria, PVM supporta funzioni di comunicazione punto-a-punto e funzioni di invio a insiemi di task. In particolare le routine che noi abbiamo utilizzato sono:

- `pvm_send (tid, msg_tag)` che invia, secondo la modalità di comunicazione asincrona, il messaggio memorizzato nel buffer attivo ed etichettato con *msg_tag*, al processo indicato dall'identificatore *tid*;

- `pvm_mcast (tids, ntask, msg_tag)` che invia il messaggio, memorizzato nel buffer attivo di invio ed etichettato con *msg_tag*, agli *ntask* specificati nel vettore *tids*; l'invio avviene definendo un albero di copertura tra i pvmd che contengono i task specificati ed inviando ad essi il messaggio; sono i pvmd a distribuire a loro volta il messaggio ai task locali senza determinare ulteriore traffico sulla rete;

- `pvm_recv (tid, msg_tag)` che blocca il processo fino a quando dal mittente *tid* non giunge un messaggio con tag *msg_tag* ed esso non viene posto nel buffer di ricezione attivo.

Nella versione più recente del sistema PVM (**pvm3.3.7**), sono disponibili delle routine che combinano i passi del protocollo di invio e ricezione in una singola chiamata, rispettivamente `pvm_psend()` e `pvm_precv()`, garantendone una implementazione interna più veloce. Poiché tali routine possono essere utilizzate esclusivamente per il trasferimento di messaggi costituiti da vettori contigui di un singolo tipo di dato e noi siamo interessati a minimizzare il numero delle comunicazioni, nella implementazione del compilatore P^3L abbiamo utilizzato il protocollo di comunicazione generale.

Al livello implementativo PVM trasmette i messaggi sulla rete sottostante utilizzandone i meccanismi più veloci disponibili quali ad esempio i protocolli TCP e UDP, nel caso di reti basate sui protocolli Internet. Su queste reti la comunicazione tra pvmd avviene generalmente utilizzando i **sockets** (punti di comunicazione) con protocollo di comunicazione UDP. I *sockets* con protocollo TCP vengono utilizzati, invece, tra il pvmd e i task locali ma è possibile utilizzarli anche per la comunicazione tra task, non necessariamente presenti sullo stesso host, specificando l'opzione opportuna nella routine `pvm_setopt(Pvm_route, route_mode)`. I *sockets* TCP risultano più veloci in quanto prevedono la determinazione di *link diretti* tra i processi comunicanti senza che la comunicazione venga mediata dai processi daemon e, dunque, comportano il raggiungimento di valori maggiori anche di un fattore

essere inviato ad un processo in esecuzione su una macchina avente struttura analoga a quella su cui è in esecuzione il processo mittente due rispetto a quelli relativi all'utilizzo dei canali di comunicazione indiretti via pvmd. Esiste, però, una mancanza di scalabilità dei *link diretti* in quanto il numero dei socket che è possibile utilizzare risulta limitato, ad esempio in alcune versioni di UNIX a non più di 30.

In conclusione è necessario aggiungere che PVM supporta in realtà molte altre routine, non solo per la gestione delle comunicazioni (ad esempio per la gestione di buffer multipli, o per effettuare delle receive non bloccanti) ma anche per la raccolta di informazioni sullo stato dei processi, dei buffer di comunicazione o della configurazione del sistema o per la definizione di gruppi dinamici di processi da sincronizzare automaticamente o ai quali inviare automaticamente messaggi in *broadcast*.

4 Template di implementazione

Come abbiamo visto nell'introduzione al linguaggio P^3L, i **template di implementazione** sono reti statiche parametriche di processi comunicanti che vengono associate ai costrutti paralleli del linguaggio per definirne un'efficiente implementazione sull'architettura target. Il compilatore, infatti, in base alla conoscenza di questi template e di informazioni supplementari fornite dai corrispondenti modelli analitici, associa ad un intero programma P^3L un'istanza di una rete di processi comunicanti ottenuta componendo opportunamente i template stessi.

Un passo fondamentale nell'implementazione del compilatore per una nuova architettura target consiste, dunque, nella definizione di template che siano in grado di esplicitare il parallelismo presente nelle applicazioni in maniera efficiente in relazione alle caratteristiche proprie della nuova architettura. In particolare le informazioni che costituiscono i *template di implementazione* sono essenzialmente di due tipi: informazioni topologiche, che definiscono l'insieme dei processi con cui viene implementato il programma e il corrispondente mapping sulla macchina astratta (**template di mapping**), ed informazioni per l'esecuzione dei processi considerati, costituite dal codice dei processi parametrizzato mediante l'uso di tipi e di macro da definire in fase di compilazione in dipendenza della specifica istanza del processo (**template dei processi**).

Trattandosi di un primo approccio implementativo le scelte che abbiamo effettuato per la definizione dei template sono state guidate da considerazioni di carattere generale più che da attenti studi analitici. Allo stato attuale, ad esempio il compilatore effettua la distribuzione dei processi su processori mediante un'euristica estremamente semplice ed anche alcune scelte relative ai procedimenti di distribuzione delle strutture dati sono basate su considerazioni la cui validità potrà essere confermata solo dallo studio successivo dei modelli implementativi. Inoltre la attuale mancanza di modelli analitici attendibili impedisce al compilatore di determinare automaticamente alcuni parametri dei template stessi, quali ad esempio il grado di parallelismo ottimo nell'implementazione dei costrutti farm e map. Ricordiamo che una caratteristica fondamentale della metodologia di programmazione che stiamo considerando consiste proprio nella capacità del compilatore di derivare da template di implementazione generali delle istanze di implementazione il più efficienti possibili. È con questo obiettivo che l'implementazione attualmente realizzata dovrà essere analizzata ed ottimizzata.

I *template di mapping* di ciascun costrutto sono rappresentati mediante un grafo diretto in cui i nodi rappresentano i processi che devono essere allocati sui nodi della macchina astratta e gli archi le comunicazioni che intercorrono tra processi. Attualmente la macchina astratta cui il compilatore fa riferimento coincide con la macchina virtuale esplicitata da PVM a meno della multiprogrammazione dei nodi. In particolare la rete di processi generata nella fase di mapping è pensata per una macchina astratta definita come una rete uniforme composta da un numero illimitato di macchine *omogenee* [2]. È solo in una fase succesiva che la rete di processi generata, viene partizionata sulle macchine effettivamente presenti, adottando, in generale, una politica di multiprogrammazione. L'adozione di tale soluzione è dettata dal limitato numero di elementi di elaborazione che, ragionevolmente, costituiranno la macchina fisica considerata, ben lontana dal poter essere annoverata tra quelle a parallelismo massiccio. Lo studio dei modelli di implementazione realizzati e dei corrispondenti modelli analitici, permetteranno, comunque, di integrare opportunamente le due fasi, sfruttando le conoscenze acquisite sui tempi di comunicazione locali o meno e ottimizzandone in tal modo i risultati. Ad esempio si cercherà di limitare, e se possibile eliminare, la multiprogrammazione utilizzando degli opportuni algoritmi di riduzione del grafo dei processi giaà sviluppati nella versione del compilatore per la Meiko.

Per finire osserviamo che, considerato il particolare tipo di macchina astratta cui facciamo riferimento, gli archi di comunicazione tra i nodi rappresentano esclusivamente dei canali di comunicazione logici cui non è possibile associare veri e propri canali fisici. In quest'ottica la definizione, cui abbiamo accennato, di *link diretti*, non implica assolutamente la creazione di vie di comunicazioni *esclusive* tra due processi ma semplicemente determina un'ottimizzazione del protocollo di comunicazione che determina a sua volta un'ottimizzazione della banda di comunicazione tra i processi.

Nella seguente descrizione dei template commenteremo velocemente i *template di mapping* adottati e descriveremo le caratteristiche essenziali dei relativi *template dei processi* facendo riferimento ai meccanismi esportati dalla macchina astratta che abbiamo precedentemente descritto. Dettagli implementativi, relativi ad esempio al passaggio dei parametri o al partizionamento dei dati, saranno discussi nella sezione successiva.

4.1 Struttura generale dei template

La struttura dei **template di mapping** che abbiamo definito è, allo stato attuale, analoga alla struttura logica dei costrutti considerati(fig.2) Questa scelta è stata effettuata con

[2] È importante non confondere l'architettura fisica, costituita dalle singole macchine eterogenee, e la macchina astratta virtualizzata da PVM, costituita da elementi di elaborazione omogenei. In particolare osserviamo che mentre lo sviluppo di PVM ha tra i suoi principali obiettivi la realizzazione di una ragionevole cooperazione di risorse computazionali esplicitamente differenti nelle caratteristiche di elaborazione, noi ci limitiamo, almeno allo stato attuale, a sfruttarne la capacità di virtualizzare una macchina con risorse computazionali omogenee.

l'obiettivo di minimizzare il numero delle comunicazioni mantenendo il parallelismo tra le differenti funzionalità dei costrutti. Si potrebbero, comunque, studiare delle ottimizzazioni che, tenendo conto soprattutto dell'alto costo delle comunicazioni, tendano a minimizzare opportunamente il trasferimento dei dati tra i processi componendo in un unico processo fasi diverse di elaborazione di uno stesso costrutto, nei casi in cui i vantaggi ricavabili dalla loro esecuzione parallela risultino trascurabili, o funzionalità di costrutti differenti nei casi in cui siano annidati o in pipeline tra loro.

Per quel che riguarda i **template di processo** essi presentano una struttura abbastanza regolare, schematizzabile mediante una fase di inizializzazione, un ciclo infinito che implementa la computazione del particolare tipo di processo ed una fase di terminazione. All'interno del codice sono presenti all'inizio e alla fine della fase di computazione vera e propria delle istruzioni per il calcolo dei tempi di esecuzione (la cui esecuzione è condizionata dalla selezione dell'opzione *profiling)*.

Nella fase di inizializzazione, oltre naturalmente alla definizione di tutte le variabili utilizzate successivamente, un generico processo effettua le seguenti operazioni:

- determina mediante la routine `pvm_parent()` il tid del processo *p3lrun* che lo ha generato;

- riceve dal processo *p3lrun*, mediante la routine di comunicazione `pvm_recv()`, il vettore contenente i *tid* di tutti i processi, etichettato con TIDS_TAG;

- invia al processo *p3lrun* un segnale di avvenuta ricezione, mediante la routine `pvm_send()` e con il tag di identificazione del canale START_ACK_TAG;

- inizializza con i tid opportuni le variabili con cui vengono identificati i processi mittenti e destinatari delle successive comunicazioni; osserviamo che la determinazione di questi tid (che sarà descritta nella sezione successiva) avviene utilizzando il numero di istanza del processo, passato al programma come primo argomento tramite *argv*, e le informazioni contenute nel file header *proc_name_tags.h*, rispetto alle quali il template di processo può considerarsi parametrizzato;

- invoca la routine `pvm_setopt()` richiedendo l'uso di link diretti per le comunicazioni successive (vedi sez.3.4).

A questa prima fase segue il ciclo di computazione vera e propria dipendente dal tipo di processo considerato che commenteremo nelle sezioni successive. L'uscita dal ciclo può essere determinata esclusivamente dalla ricezione di una marca di terminazione (il campo *end_mark* del messaggio contenente il valore TRUE). Ricevuta tale marca il processo la propaga ai suoi destinatari quindi lascia PVM mediante la routine `pvm_exit()` e, infine, termina mediante il comando `exit()`.

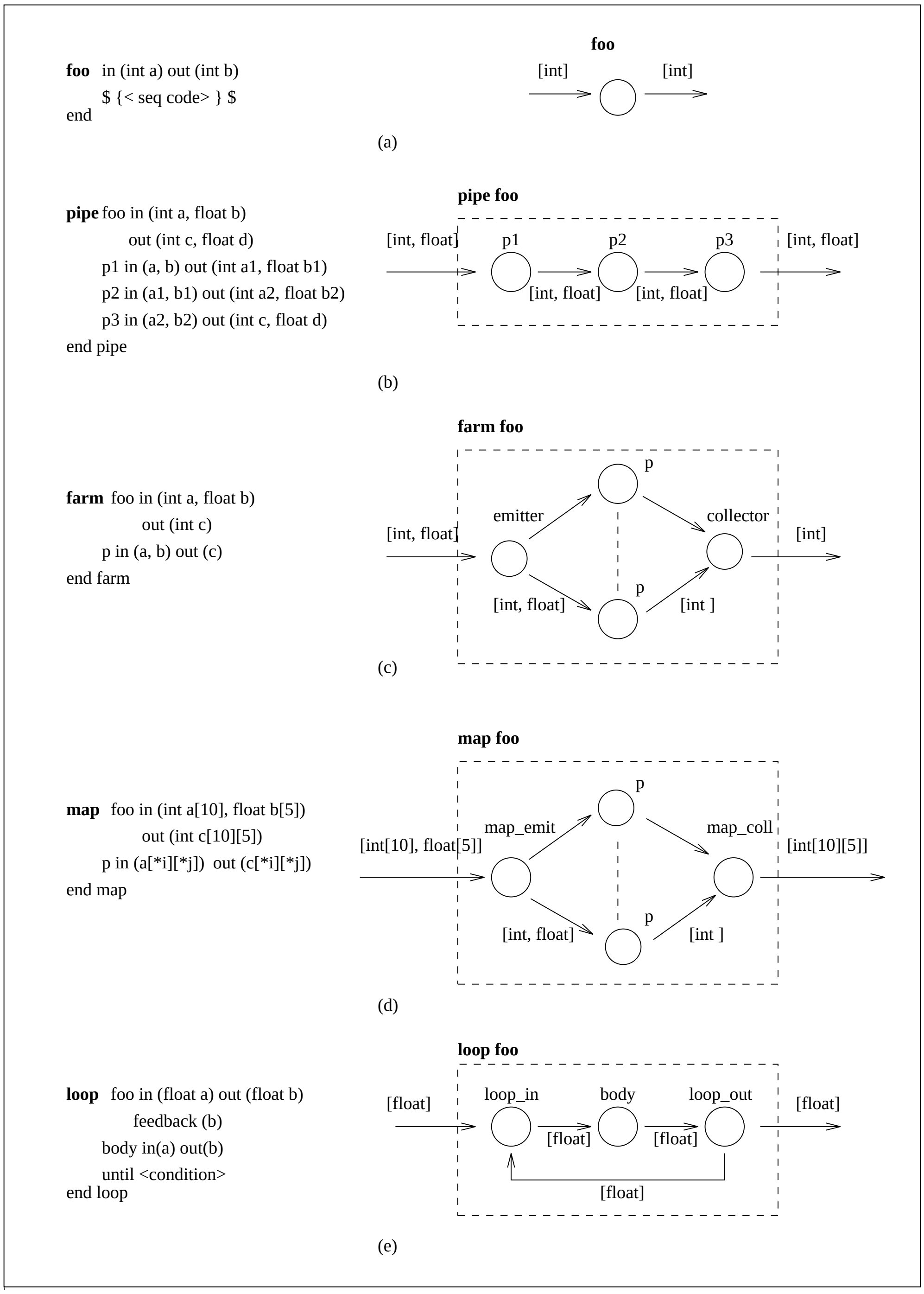

Figura 2. Definizione e struttura logica dei costrutti P^{3}L

4.2 Il costrutto *sequential*

Il costrutto **sequential** (fig.2.a), corrisponde evidentemente ad un unico processo (**seq**) che dispone di due soli canali di comunicazione: un canale di input per la ricezione dei task (intesi come elemento singolo del flusso di dati da elaborare) in ingresso e un canale di output per la spedizione dei risultati della computazione. Osserviamo che in tutti i processi i canali per la propagazione dei task sono etichettati mediante l'identificatore SEQ_TAG.

Le operazioni che il processo effettua all'interno del ciclo di computazione sono estremamente semplici ovvero:

- riceve da un processo mittente un task da elaborare;
- elabora il task, invocando la funzione corrispondente al codice sequenziale scritto dall'utente;
- invia ad un processo destinatario il risultato dell'elaborazione.

Le modalità di implementazione del protocollo di comunicazione sono essenzialmente quelle esaminate nel corso della descrizione del sistema PVM (sez.3.4). Ulteriori caratteristiche dell'implementazione saranno analizzate nel corso della descrizione delle funzionalità del back_end (sez.5.2).

4.3 Il costrutto *pipe*

Il template di mapping del costrutto **pipe** (fig.2.b) consiste nell'inserimento di canali di comunicazione tra gli stadi successivi interni al pipeline. Ciascuno stadio, non necessariamente sequenziale, sarà a sua volta implementato dai template di mapping del costrutto corrispondente secondo la regola di composizione gerarchica su cui è basato P^3L.

Osserviamo che un generico programma P^3L è sempre definito come un pipe ([Orl92]) in cui il primo e l'ultimo stadio sono dei particolari costrutti sequenziali che, rispettivamente, acquisiscono l'input del programma e ne gestiscono l'output. Ad essi corrispondono due template di processo (**seq_no_input** e **seq_no_output**) analoghi a quello che implementa il costrutto **sequential** ma con alcune particolarità. Il primo presenta una lista di parametri di ingresso vuota e dunque non necessita di alcun protocollo per la ricezione dei dati. Il processo sequenziale incluso, corrispondente a parte del codice sequenziale scritto dall'utente, genera, eventualmente leggendolo dall'input, uno *stream* di dati che costituiscono i parametri di ingresso dell'intera applicazione P^3L. Esso contiene, tra l'altro una speciale funzione di terminazione (`p3l_exit()`) in base alla quale, quando viene raggiunta la fine dello stream di input, viene definito ed inviato al processo successivo il messaggio di terminazione. Analogamente il processo *seq_no_output* presenta una lista vuota di parametri di uscita, ma il codice sequenziale incluso potrà eventualmente effettuare delle operazioni per la gestione dei risultati del programma, quale ad esempio la scrittura sullo standard output.

4.4 Costrutto *farm*

La struttura logica del costrutto **farm** (fig.2.c) si compone di tre attività principali, corrispondenti ciascuna ad un particolare template di processo:

- *distribuzione* dei task, da parte di un emettitore a cui essi pervengono, ai worker che li dovranno elaborare;
- *elaborazione* dei task da parte di ciascun worker e produzione dei risultati;
- *raccolta* dei risultati elaborati dai worker effettuata da un collettore.

Nell'implementazione del costrutto si è tenuto conto di due ulteriori funzionalità, ovvero:

- *bilanciamento del carico* tra i worker, effettuato dinamicamente dal processo emettitore;
- *riordinamento* dei risultati nel processo collettore; allo stato attuale esistono in realtà due diversi costrutti (cui corrispondono differenti processi) di tipo farm, *ordinante* e *non ordinante*; tra i due analizzeremo il primo poiché dal punto di vista implementativo il secondo risulta essere una sua semplificazione.

La fase di distribuzione dei task, implementata dal template di processo **emit**, è costituita da una prima fase di inizializzazione del farm in cui ad ogni worker viene inviato un task da elaborare. All'interno del successivo ciclo di computazione si alternano, la ricezione di un task proveniente dall'esterno e di un messaggio di disponibilità, consistente nel tid del processo mittente ed etichettato con un SEQ_SYNC_TAG, da parte di un worker che ha terminato la sua computazione. A queste comunicazioni segue l'invio del task pervenuto al worker libero. Una volta uscito dal ciclo, il processo emettitore si preoccuperà, prima di effettuare la propria terminazione, di attendere da tutti i worker il segnale di disponibilità onde evitare che possano rimanere bloccati tentando una impossibile comunicazione. Osserviamo che al task viene aggiunta una marca numerica (*tag*[3]), incrementata all'arrivo di ogni nuovo task, che permetterà al collettore di riordinare i risultati.

I worker del farm sono implementati mediante il template **seq_synch**, essenzialmente uguale al template **seq** che implementa il costrutto sequenziale. L'unica differenza tra i due template consiste nel fatto che il **seq_synch** invia un segnale di disponibilità all'emettitore, etichettato con SEQ_SYNC_TAG, non appena ha terminato la computazione di un task e l'invio del risultato al collettore.

L'ultima fase, quella di raccolta viene implementata dal template di processo **coll**. Per garantire l'ordinamento dei risultati, giunti al collettore essi vengono memorizzati in un

[3] Pur essendo indicati con lo stesso nome (*tag*), è bene fare attenzione a non confondere le costanti numeriche utilizzate esplicitamente nelle routine di invio e ricezione dei messaggi per identificare il tipo di canale logico tra processi e i valori numerici interni ai messaggi che ne costituiscono la marca di ordinamento.

buffer nella posizione relativa al numero d'ordine indicato dal loro tag. Saranno inviati all'esterno del costrutto solo dopo che tutti gli eventuali risultati con tag minore saranno stati spediti. È evidente che poiché il tag esaurisce qui la sua funzione non verrà propagato all'esterno insieme al risultato (eventualmente il risultato potrà avere degli altri tag relativi a costrutti più esterni).

Come abbiamo già detto esiste anche una implementazione del farm non ordinante che si compone di processi analoghi (**emit_no_tag, coll_no_tag, seq_synch**) a quelli descritti con l'unica differenza che non viene effettuata la gestione del tag (dunque non è più necessario, ad esempio, gestire il buffer per la memorizzazione dei risultati nel collettore). Essa viene utilizzata in casi in cui è inutile ordinare i risultati del costrutto, in particolare quando è un costrutto più esterno ad effettuarne l'ordinamento.

4.5 Costrutto *map*

Il template di implementazione del costrutto **map** è sotto certi aspetti simile a quello del farm. Possiamo, infatti, interpretare questo costrutto come un modulo che, per ciascun task in ingresso, produce dei sottotask indipendenti (*data decomposition*) che vengono eseguiti parallelamente e i cui risultati sono ricomposti in un singolo parametro di uscita (*data recomposition*).

I template di processo che compongono il *template di implementazione* del costrutto **map** sono analoghi a quelli del farm ma devono in più gestire il partizionamento dei dati. In realtà il processo emettitore invia l'intero task ai worker che, utilizzando opportuni parametri di partizionamento, ne elaborano soltanto un sottoinsieme. Questa politica è stata scelta in considerazione del fatto che sulla macchina astratta considerata, non solo i tempi di *start-up* ma anche quelli per l'accesso alla rete di comunicazione possono rivelarsi piuttosto alti. Dunque sembrerebbe preferibile effettuare un'unica comunicazione (*multicasting*), anche se abbastanza pesante in termini di dimensione del messaggio, piuttosto che ripetuti invii di strutture dati più piccole. Questa considerazione andrà successivamente validata o smentita dalla misurazione effettiva dei tempi di comunicazione e dalla definizione di un corrispondente modello analitico.

Allo stato attuale il template **map_emit**, che implementa la fase di distribuzione, si limita ad aggiungere un tag al task che gli è pervenuto e ad inviarlo utilizzando la primitiva di comunicazione `pvm_mcast()` a tutti i suoi worker.

Come abbiamo accennato, il partizionamento vero e proprio dei task avviene direttamente nel template di processo relativo ai worker (**map_work**). In particolare a ciascun worker viene assegnato un numero, *num_task*, eventualmente maggiore di 1 di partizioni da elaborare. Utilizzando come parametri attuali gli argomenti con cui è stato attivato, trasmessi tramite *argv*, e le informazioni incluse dal compilatore in un file header (*proc_name_tags.h*), il worker esegue iterativamente la computazione di tutte le partizioni a lui assegnate facendo ricorso a delle opportune procedure definite dal compilatore in base alla particolare istanza del processo e contenute in un'ulteriore file header (*proc_name_macros.h*). In una sezione successiva (sez.5.2) mostreremo più dettagliatamente quali sono i parametri necessari per il partizionamento e in che modo essi vengono trasmessi e utilizzati dai worker.

Per il momento ci limitiamo a descrivere la fase di elaborazione della partizione facendo riferimento alle funzionalità delle procedure utilizzate:

- viene estratta dal task in ingresso la partizione da elaborare e con essa viene creato un task locale in cui sono memorizzati anche gli indici che ne hanno permesso l'identificazione (*from_ext_to_int_partition*)[4];

- viene eseguita la computazione della partizione invocando il codice scritto dall'utente e memorizzando il risultato in un buffer avente un numero di posizioni pari al numero delle partizioni assegnate al worker (*call_seq_proc*);

- viene effettuata la copia degli indici di scomposizione/ricomposizione dal sottotask attuale al risultato ad esso corrispondente così da permettere al processo collettore la ricomposizione del risultato globale del task (*form_inp_to_out*);

- vengono aggiornati gli indici per l'elaborazione della partizione successiva (*inc_indexes*).

Terminata l'elaborazione delle partizioni assegnategli, il processo impacchetta, oltre alla marca di terminazione e agli eventuali tag, il numero dei task elaborati e tutti i risultati corrispondenti, per poi inivarli congiuntamente al processo collettore. La scelta di memorizzare i risultati parziali per poi spedirli in un unico messaggio, tipicamente di taglia maggiore, è stata effettuata in base alle considerazioni sui tempi di *start-up* e di accesso alla rete di comunicazione già discusse a proposito della definizione delle caratteristiche generali del costrutto map. Anche in questo caso la scelta dovrà essere validata mediante misurazioni reali di tempi e modelli analitici opportuni.

L'ultima funzionalità del costrutto map è implementata mediante il processo collettore (**map_coll**) che comincia il suo ciclo di computazione con la ricezione dei risultati di un sottotask, ovvero ponendosi in attesa di un messaggio etichettato mediante SEQ_TAG e proveniente da un generico processo[5]. In base all'informazione relativa al numero di partizioni elaborate, contenuta nel messaggio, il collettore spacchetta tutti i risultati ricevuti in un task locale; quindi, utilizzando gli indici ad essi associati, li ricompone mediante una opportuna funzione (*from_int_to_ext_partition*). La ricomposizione viene effettuata all'interno di un buffer i cui elementi sono composti da un campo per la memorizzazione del risultato di un intero task e uno per la registrazione del numero di partizioni del task relativo man mano ricevute. Il risultato finale viene inviato all'esterno del costrutto solo quando la ricomposizione è completata, ovvero quando il numero di partizioni ricevute è

[4] Al livello di implementazione gli indici di scomposizione vengono ricopiati nel task attuale come parametri aggiunti a quelli che costituiscono la partizione da elaborare e come tali vengono utilizzati dalla procedura di *estrazione* della partizione.

[5] È bene ricordare che anche quando un processo è disposto ad accettare un messaggio da un generico mittente, tramite *wildcard*, la garanzia che la comunicazione avviene solo con i processi interessati è data dal fatto che i mittenti specificano esplicitamente il destinatario.

uguale al numero di partizioni totale, e il rispetto dell'ordinamento dei task lo permette. La terminazione del processo collettore avviene in seguito alla ricezione di ciascuna delle marche di terminazione inviate dai worker e alla conseguente propagazione di un'analoga marca verso l'esterno.

Osserviamo che il numero di posizioni del buffer in cui vengono memorizzati i risultati dei task elaborati dai worker, dovrebbe essere determinato in base alla capacità o meno del sistema PVM di supportare più operazioni consecutive di multicasting non completate (ovvero in cui la ricezione non è stata ancora effettuata da tutti i processi destinatari). Infatti, senza fare alcuna assunzione sui tempi di elaborazione relativi dei worker, il numero massimo di task differenti le cui partizioni possono essere elaborate contemporaneamente dipende dalla capacità del processo emettitore di effettuare o meno un'operazione di multicasting prima che le precedenti siano completate. Nel caso in cui questo non sia possibile il numero di task differenti contemporaneamente in esecuzione sarà al più pari a due. Poiché non disponiamo di dati certi a tale proposito e supponendo ragionevolmente che, in ogni caso, il numero dei task in elaborazione risulti piuttosto basso, abbiamo definito il numero di posizioni del buffer nel processo collettore come funzione della quantità di spazio necessaria alla memorizzazione di ciascun risultato.

Ulteriori esperimenti che ci aiutino a definire con maggiore chiarezza il comportamento delle primitive di comunicazione in PVM ci permetteranno non solo di ottimizzare l'occupazione di memoria ma anche di evidenziare l'eventuale insorgere di errori in particolari situazioni critiche (ad esempio non è escluso che l'eventuale esecuzione di due, o più, operazioni di multicasting successive, qualora la prima non sia stata completata possa portare a malfunzionamenti del sistema).

4.6 Costrutto *loop*

Il costrutto **loop** è stato introdotto per permettere l'elaborazione iterativa dei task fino al soddisfacimento di una particolare condizione booleana. La particolarità nell'implementazione di tale costrutto è che esso deve garantire l'elaborazione parallela di task provenienti sia dall'esterno del costrutto, attraverso il canale di ingresso, che dall'interno, attraverso il canale di feedback.

Esso è stato implementato mediante la definizione di due template di processo che gestiscono rispettivamente l'ingresso (**loop_in**) e l'uscita (**loop_out**) del costrutto.

Il processo **loop_in** comincia la computazione con la ricezione di un task dal canale di ingresso ed il conseguente invio al costrutto annidato. Successivamente si pone in un ciclo in cui, alternativamente testa l'arrivo di messaggi su entrambi i canali; i task ricevuti saranno in ogni caso inviati al costrutto annidato ma mentre quelli provenienti dal canale di feedback saranno immediatamente rispediti allesterno, agli altri verrà prima associato un tag di ordinamento. In questo secondo caso il processo si preoccuperà anche di effettuare le necessarie operazioni di impaccchettamento e spacchettamento. Osserviamo infine che la priorità nelle comunicazioni viene comunque data al processo di feedback, infatti il processo loop_in prima di porsi in attesa su di un generico canale di ingresso, mediante *wildcard*, effettua una *receive non bloccante* sul canale di feedback.

La gestione dell'uscita dal costrutto è implementata nel template **loop_out**. Al momento della ricezione di un task, il processo invoca una funzione predefinita, in cui il compilatore avrà incluso il codice specificato dall'utente, per determinare se la condizione di terminazione è verificata. In dipendenza del risultato, positivo o meno, il task viene rispedito sul canale di feedback o, viceversa, memorizzato in un buffer per poi essere inviato all'esterno del costrutto in base al numero d'ordine fornito dal tag corrispondente.

Per il carattere iterativo della computazione effettuata, il problema più importante per l'implementazione del costrutto *loop* consiste nella gestione della terminazione. È evidente, infatti, che quando riceve la marca di terminazione il processo di ingresso al loop non può terminare, nè tantomeno effettuare la propagazione della marca, fino a quando non sarà terminata la computazione di tutti i task che attualmente circolano all'interno del costrutto. Poiché la soluzione adottata si ripercuote sull'implementazione di tutti i processi ne daremo una breve descrizione nel paragrafo successivo.

4.7 Gestione della terminazione

La gestione della terminazione avviene essenzialmente mediante l utilizzo di una marca numerica associata ai task da elaborare e propagata con essi utilizzando un apposito campo *end_mark* [6] presente nei messaggi con cui i task vengono scambiati tra i processi.

Schematizzando potremmo distinguere il procedimento adottato in tre fasi principali: inizializzazione della marca, propagazione, terminazione vera e propria del programma.

Nel processo *seq_no_input* è definita una funzione `p3l_exit()` la cui invocazione determina l'assegnamento del valore positivo al campo *end_mark* del messaggio attualmente in uscita dal processo. Tale funzione tipicamente viene invocata dal codice utente al momento in cui termina la lettura dei dati di ingresso e dunque la creazione dello stream di task da elaborare. Sul grafo di processi che costituisce l'implementazione di un generico programma P^3L viaggerà, quindi, una serie di messaggi con marca di terminazione non positiva contenenti ciascuno un task da elaborare (o il risultato più o meno parziale della sua elaborazione) più un messaggio terminale con marca di terminazione positiva.

Ciascun processo *intermedio* dell'implementazione, quando riceve un messaggio, come prima cosa ne esamina la marca di terminazione, poi, in base al suo valore e ad alcune particolari condizioni dipendenti dal tipo di processo, decide se effettuare la computazione del task o limitarsi a propagare la marca e/o terminare.

Quando la marca di terminazione positiva raggiunge l'ultimo processo del grafo, ovvero il processo *seq_no_output*, si è ormai realizzata la terminazione in cascata di tutti i processi

[6] In realtà se non considerassimo il costrutto loop sarebbe sufficiente una marca booleana per determinare la terminazione (marca a True) o meno (marca a False) del processo. Per questo motivo, e in accordo alla effettiva implementazione dei costrutti, considereremo i valori numerici come un'estensione del dominio dei booleani e parleremo di marca positiva (True) e negativa (False) per indicare rispettivamente i valori 1 e 0 eventualmente assunti dal campo end_mark.

del programma: dunque, prima di terminare a sua volta il processo *seq_no_output* invia un messaggio etichettato con END_PROG_TAG al processo *p3lrun* che determina la terminazione del sistema PVM e completa la terminazione del programma P^3L.

In fig.3 abbiamo schematizzato il flusso di uno stream di due task (*t1* e *t2* più il task fittizio finale *t'3*) con relativa marca di terminazione nell'ambito di un programma composto del solo costrutto farm (con *t'i* abbiamo indicato il risultato della elaborazione del corrispondente task *ti*).

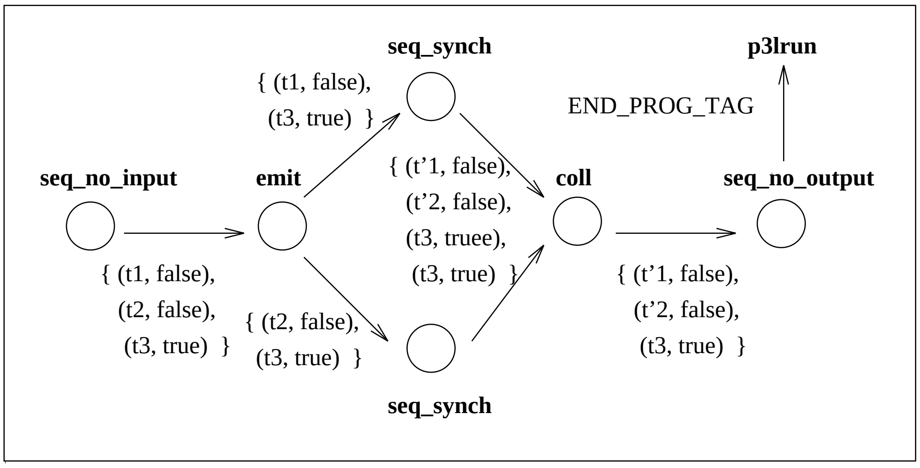

Figura 3. Esempio di propagazione di uno stream di task in un costrutto *farm*

Abbiamo già accennato che per permettere la gestione del costrutto *loop* è stato necessario utilizzare per la marca di terminazione dei valori interi; infatti, oltre ai valori 1 e 0 cui viene attribuito il tradizionale significato booleano, la marca di terminazione può assumere valori a partire da una costante numerica `Loop_end_mark` nota a tutti i processi del programma. Quando il processo di ingresso ad un costrutto loop, *loop_in*, riceve un messaggio con la marca di terminazione positiva assegna il valore `Loop_end_mark` al campo end_mark di un messaggio contenente un task fittizio e propaga[7] il messaggio ai processi che implementano il costrutto per segnalare l'inizio della fase di terminazione. È evidente infatti che in questo particolare tipo di costrutto la terminazione può avvenire soltanto dopo che l'elaborazione di tutti i task attualmente circolanti al suo interno è stata effettuata ed è quindi necessario che, quando si verifica, tale evento venga rilevato e comunicato al processo *loop_in*. Poiché l'unico processo in grado di rilevarlo è il processo di uscita dal costrutto, i processi che implementano la parte interna del costrutto si limitano a propagare un qualunque messaggio avente questa particolare marca di terminazione in modo

[7]Il termine *propagare*, spesso usato impropriamente per indicare l'invio di un messaggio al processo successivo, è stato scelto per sottolineare che tale messaggio una volta inviato verrà appunto propagato da un processo all'altro fino a raggiungere il processo finale del costrutto o del grafo di implementazione.

che esso possa raggiungere il processo *loop_out*. Tale processo, riconosciuto il messaggio di terminazione del loop, ne registra il tag di ordinamento e continua il suo normale ciclo di computazione fino all'esaurimento dei task presenti nel costrutto, ovvero fino all'uscita dal costrutto di tutti i task aventi tag minore di quello registrato. A questo punto invia un particolare messaggio etichettato con LOOP_END_TAG al processo *loop_in* che, a sua volta propaga un normale messaggio di terminazione con marca positiva e, finalmente, termina.

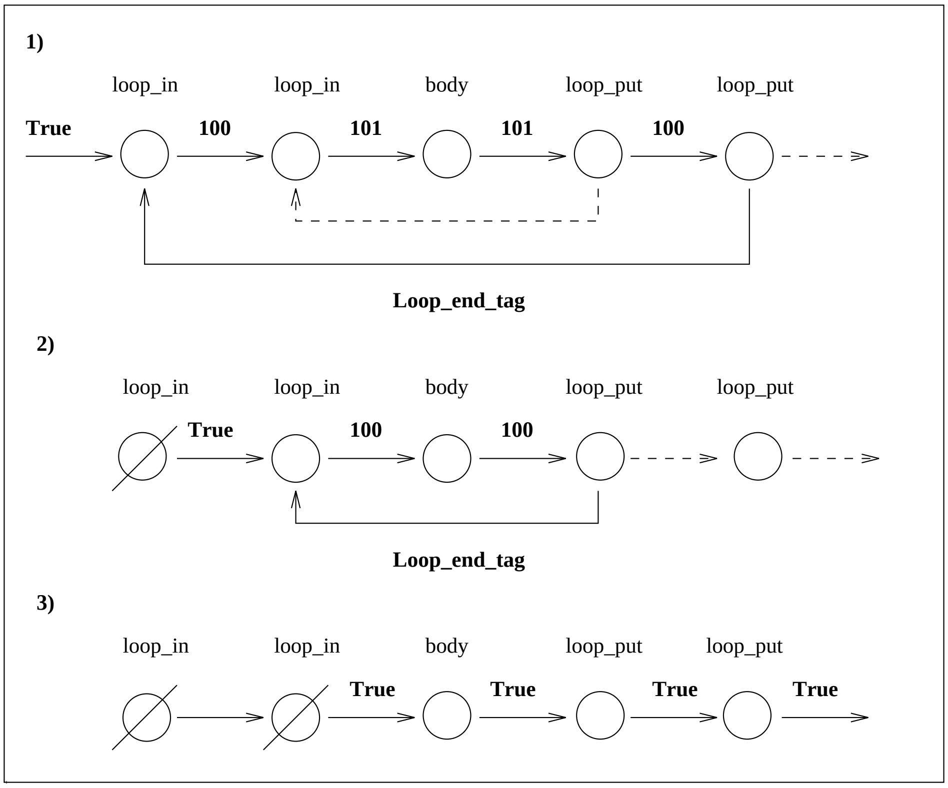

Figura 4. Esempio di propagazione della marca di terminazione in caso di costrutti loop annidati (`Loop_end_mark` = 100)

In realtà la gestione della terminazione è ulteriormente complicata dalla eventuale presenza di loop annidati. Per essere certi che ciascun costrutto loop abbia un comportamento analogo a quello appena descritto indipendentemente dal suo livello di annidamento è necessario che la terminazione, intesa come propagazione della marca di terminazione positiva e conseguente terminazione in cascata dei processi che la ricevono, avvenga a partire dal costrutto più esterno. Dunque il messaggio di terminazione del costrutto loop più esterno, avente marca di terminazione `Loop_end_mark`, deve poter oltrepassare i processi *loop_out* appartenenti ai costrutti annidati per poter essere "raccolto" solo dal processo *loop_out* corrispondente. Perché questo sia possibile i processi *loop_in* annidati effettuano una sorta di conteggio del livello di annidamento, incrementando il valore della marca di terminazione ogni qualvolta essa risulti maggiore o uguale a `Loop_end_mark`. Analogamente i processi *loop_out* effettuano un decremento del valore della marca di terminazione ogni qualvolta essa risulti maggiore di `Loop_end_mark`. Si ha quindi che nel caso in cui un processo *loop_out* riceve un messaggio con marca pari a `Loop_end_mark`, può riconoscersi come appartenente al costrutto loop più esterno e, in quanto destinatario del messaggio, effettuare le procedure di gestione della terminazione del costrutto sopra descritte. È evidente che questo tipo di comportamento si propagherà in cascata man mano che si verificherà la terminazione dei loop piu esterni ovvero che i successivi processi *loop_in* effettueranno la propagazione della marca di terminazione positiva. In fig.4 abbiamo schematizzato le tre fasi principali della terminazione di due costrutti loop annidati intese come successioni di comunicazioni per la propagazione della marca di terminazione del loop. Abbiamo utilizzato le linee tratteggiate per indicare i canali di comunicazione non utilizzati in quella particolare successione di comunicazioni e con la linea trasversale i processi terminati in dipendenza delle comunicazioni avvenute nella fase precedente.

Nella tabella mostrata in fig.5 abbiamo schematizzato il comportamento dei processi (esclusivamente in relazione al problema della terminazione) in funzione del valore della marca di terminazione del messaggio ricevuto. Ne facciamo quindi un breve commento, trascurando i processi di ingresso e di uscita dei quali abbiamo già discusso, per chiarire alcuni dettagli di implementazione dei processi e per sottolineare le differenze di comportamento tra i vari costrutti.

Per cominciare ricordiamo che il comportamento di un generico processo può essere essenzialmente di tre tipi: quando la marca è positiva il processo la propaga e termina; se la marca è negativa effettua l'elaborazione del task ricevuto e prosegue il suo normale ciclo di computazione; in ogni altro caso, quando cioé la marca sta ad indicare l'inizio della fase di terminazione di un costrutto loop ad un qualche livello di annidamento, prosegue il ciclo di computazione dopo aver propagato il task fittizio ricevuto.

In realtà la presenza di un task fittizio associato ai messaggi di terminazione del loop in alcuni casi risulta trasparente al processo che dovrebbe elaborarlo. Nel caso dei processi relativi al costrutto map, ad esempio, ogni task inviato ai worker (e successivamente al collettore) viene preceduto, come è d'obbligo nella nostra implementazione, da una informazione relativa al numero di sottotask da elaborare (o elaborati) che, in questo caso, viene posta a zero. Dunque mentre per gli altri processi di computazione vera e propria (*seq*, *seq_synch*) l'elaborazione del task fittizio viene evitata testando esplicitamente il valore della marca di terminazione, nel caso del processo *map_work* il comportamento corretto è già imposto dalla struttura del task fittizio. Analogamente nel caso del processo emettitore del costrutto farm il task fittizio viene elaborato, ovvero spacchettato/impacchettato, come ogni altro task in modo tale da assegnargli il giusto tag di ordinamento. In particolare, nel caso di messaggi con marca di terminazione di un loop, i processi emettitore e collettore del costrutto farm (*emit*, *coll*) si comportano come se si trattasse di messaggi con normali task da elaborare, mentre nel caso del costrutto map il task fittizio (preceduto dall'informazione

Seq_no_input

legge un elemento dello stream di ingresso:	invia un msg con end_mark = false
fine dello stream di ingresso -> P^{3}L_exit():	invia un msg con end_mark = true

Seq_no_output

True:	invia al processo p3lrun un msg etichettato con END_PROG_TAG e termina
False:	elabora

Seq - Seq_synch - Map_work

True:	propaga e termina
False:	elabora e propaga
≠ True o False:	propaga

Emit - Emit_no_tag

True:	propaga a tutti i worker (quando disponibili) e termina
≠ True:	elabora e propaga ad un worker (eventualmente un task fittizio)

Map_emit

True:	propaga a tutti i worker e termina
False:	elabora e propaga a **tutti** i worker
≠ True o False:	elabora e propaga a **un** worker un task fittizio con num_task = 0

Coll - Coll_no_tag - Map_coll

True:	propaga e termina dopo aver ricevuto un msg con end_mark positivo da **tutti** i worker
≠ True:	elabora e propaga

Loop_in

True:	propaga un msg con end_mark = Loop_end_mark
False:	elabora e propaga
≥ Loop_end_mark:	propaga un msg con end_mark incrementato di 1
riceve dal processo loop_out un msg con tag LOOP_END_TAG:	propaga un msg con end_mark = True e termina

Loop_out

True:	propaga e termina
False:	elabora e propaga
> Loop_end_mark:	propaga un msg con end_mark decrementato di 1
= Loop_end_mark:	continua il ciclo di computazione fino all'elaborazione del task avente tag precedente al tag del messaggio di terminazione del loop, quindi invia al processo loop_in un msg con tag LOOP_END_TAG

Legenda

propaga: invia al processo successivo un messaggio con end_mark avente lo stesso valore di quello associato al task di ingresso ricevuto dal processo precedente. Come task viene inviato il risultato della elaborazione, quando indicato esplicitamente o il task ricevuto in ingresso, negli altri casi.

elabora: effettua la computazione propria del costrutto sia essa l'elaborazione vera e propria del task in ingresso (seq) sia eventuali operazioni di impacchettamento o spacchettamento (processi emettitori/collettori).

Figura 5. Schema di comportamento dei processi in funzione del valore della marca di terminazione.

relativa al numero di sottotask posta a zero) viene propagato dall'emettitore (*map_emit*) ad un solo worker e non a tutti come accade nel normale caso di partizionamento del task. Analogamente nel processo collettore (*map_coll*) il messaggio contenente il task fittizio viene subito rispedito all'esterno del costrutto (ponendo il numero di partizioni del task ricevute pari al numero di partizioni da elaborare) senza attendere ulteriori comunicazioni.

Nel caso di messaggi di terminazione veri e propri, ovvero con marca positiva, è da rilevare che il processo *emit*, per effettuare una corretta gestione delle comunicazioni, prima di terminare attende la ricezione del segnale di disponibilità da parte di tutti i worker. Una ulteriore particolarità si ha poi nei processi collettori (*coll*, *map_coll*) che per poter cominciare le procedure di terminazione attendono di ricevere la marca positiva da tutti i worker.

Una considerazione a parte, infine, va fatta per i processi di implementazione del costrutto loop: come abbiamo ampiamente discusso tali processi devono farsi carico della gestione di costrutti annidati e dunque le operazioni necessarie per la gestione della marca di terminazione risultano maggiormente differenziate.

5 Funzionalità del compilatore: *back_end*

Tra le componenti del compilatore il **back_end**, che ne costituisce lo stadio finale, risulta pesantemente dipendente dall'interfaccia fornita dalla macchina astratta dunque, allo stato attuale, dalla classe di architetture target considerata[8].

Più precisamente, il *back_end* definisce la configurazione della macchina astratta, nel nostro caso quella virtualizzata da PVM, e sulla base delle strutture di mapping definite nella fase precedente della compilazione (*middle_end*), genera il codice eseguibile utilizzando i meccanismi di interfaccia che la macchina astratta fornisce.

Il codice di ciascun processo generato nella fase di mapping viene definito integrando il corrispondente *template di processo*, contenuto nella apposita libreria, con le informazioni, necessarie al supporto a tempo di esecuzione, proprie della particolare istanza di processo considerata. Per ogni processo, ad esempio dovranno essere determinate ed incluse nel codice eseguibile le definizioni dei tipi trasmessi sui canali di comunicazione e delle relative funzioni di impacchettamento e spacchettamento dei buffer, gli identificatori dei processi con cui le comunicazioni vengono effettuate, gli eventuali indici di partizionamento dei dati, A questo va poi aggiunta la generazione vera e propria degli eseguibili ottenuta mediante la compilazione del codice dei processi, scritti nel linguaggio host sequenziale.

Le funzionalità principali del *back_end*, ed in particolare della nostra implementazione, possono essere schematizzate nel modo seguente:

- generazione, per ciascun processo, di un file header contenente le definizioni dei tipi attuali trasmessi sui canali di comunicazione (**proc_name_types.h**);

[8]Attualmente ad ogni particolare classe di architetture target corrisponde una specifica macchina astratta in grado di virtualizzarla

pvm_config:

```
pisa-tts dx=/./pvm3/lib/HPPA/pvmd3 ep=/./Esempio/HPPA wd= /./Esempio
apollo dx=/./pvm3/lib/HPPA/pvmd3 ep=/./Esempio/HPPA wd= /./Esempio
database dx=/./pvm3/lib/SUN4/pvmd3 ep=/./Esempio/SUN4 wd= /./Esempio
```

Figura 6. Esempio di file *pvm_config*.

- generazione, per ciascun processo, di un file header contenente le definizioni delle macro o delle funzioni utilizzate, ad esempio per particolari copie dei dati, nel corrispondente template di processo e definite sulla base dei parametri attuali della particolare istanza considerata (**proc_name_macros.h**). Tra le altre ci interessa segnalare le macro di impacchettamento/spacchettamento dei parametri che costituiscono un passo fondamentale nella definizione del protocollo di comunicazione tra i processi;
- generazione del file di attivazione dei processi (**p3lrun**). In questa fase, sulla base delle informazioni ricavate nella definizione delle attivazioni dei processi, vengono anche definiti degli ulteriori file header (**proc_name_tags.h**) contenenti per ciascun processo le informazioni necessarie per identificare i partner di comunicazione;
- creazione, per ciascun processo, del corrispondente file sorgente, effettuata aggiungendo al codice del corrispondente template di processo le direttive per l'inclusione di un file contenente alcune definizioni di costanti o di tipi condivise dall'insieme dei processi (**p3l__typedefs.h**) e dei file header sopra descritti;
- configurazione della macchina virtuale e generazione dei file eseguibili, ottenuta definendo dei file contenenti le indicazioni per la compilazione del codice sequenziale su ciascuna macchina della configurazione (**make_host**) ed uno script shell (**compile**) che ne diriga l'esecuzione parallela.

Più che ai singoli passi del processo di compilazione in questa sede siamo interessati ad analizzare le principali problematiche incontrate nel corso dell'implementazione stessa, soprattutto in relazione al particolare tipo di macchina astratta considerata. In questa ottica le analizzeremo e ne presenteremo la soluzione nelle sezioni successive, senza necessariamente attenerci alle diverse fasi che implementano le funzionalità del *back_end* come sono state appena descrittte.

5.1 La configurazione della macchina astratta

La configurazione della macchina astratta consiste semplicemente nella creazione dell'*hostfile*, che abbiamo denominato **pvm_config**, (fig.6) con cui successivamente viene attivato il sistema PVM che la virtualizza.

```
p3lrun::

  main ()
  {
   ...........                               /* definizioni e inizializzazioni */
  system ("pvmd3 pvm_config &");                                  / * attiva Pvm */
  printf (" P^3LRUN: waiting starting pvm ...\ n");
  system ("sleep 1");
  while (conf_ok)
     { while ((ret = pvm_config (&nhost, &narch, &hostp) < 0 ));
       if (nhost = NUM_HOST)
          conf_ok = 0;
     }        /* itera fino alla configurazione completa della macchina virtuale */
                                                             /* attiva i processi */
  argv_[0] = "0";
  pvm_spawn (" seq_no_input0", argv_, 1, "pisa-tts", 1, &tids[0] );
  ...........
  argv_[0] = "0";
  pvm_spawn (" seqX", argv_, 1, "pisa-tts", 1, &tids[0] );
  ...........
  argv_[0] = "0";
  argv_[1] = "7500";
  argv_[2] = "1";
  pvm_spawn (" map_workY", argv_, 1, "pisa-tts", 1, &tids[0] );
  ...........
  argv_[0] = "0";
  pvm_spawn (" seq_no_outputW", argv_, 1, "pisa-tts", 1, &tids[0] );
  pvm_initsend (PvmDataDefault);   /* invia il vettore dei tid a tutti i processi */
  pvm_pkint (tids, NUMPROC, 1);
  pvm_mcast (tids, NUMPROC, TIDS_TAG);
  for (i := 0; i < NUMPROC; i ++)        /* attende il segnale di avvenuta ricezione
     pvm_recv (-1, START_ACK_TAG);                    dei tid da tutti i processi */
                    /* a partire da questo momento tutti i processi sono sicuramente
                                                  entrati nel loro ciclo di computazione */
```

Figura 7. Schema del file *p3lrun*.

Secondo le modalità discusse nella sezione 3.1, l'hostfile contiene i nomi degli host, forniti dal *middle_end*, che compongono la macchina astratta, e, per ciascuno di essi, l'indicazione della collocazione del processo pvmd e dei file eseguibili relativi alla sua architettura. Tanto i processi daemon che i file eseguibili sono infatti partizionati e/o replicati, in subdirectory opportunamente denominate, in dipendenza dell'architettura dell'host su cui dovranno essere eseguiti. Il tipo di architettura per ciascun host viene determinato invocando, tramite un comando di shell remota, la funzione `pvmgetarch` fornita da PVM.

Ben più interessante e fortemente legata alla macchina astratta considerata, è la generazione del file **p3lrun** per l'attivazione automatica di PVM e dei processi che compongono l'applicazione. L'attivazione del sistema è implementata mediante l'invocazione del comando `pvmd` con un opportuno *hostfile*, a cui segue una pausa forzata e ripetuti controlli per verificare che tutti gli host siano effettivamente stati aggiunti alla macchina (fig.7).

La parte centrale del file *p3lrun* consiste nell'attivazione dei processi che implementano l'applicazione, effettuata mediante una serie di chiamate alla funzione `pvm_spawn()` con mapping esplicito. Il *tid* del processo attivato, restituito dalla `pvm_spawn()`, viene memorizzato in un vettore di elementi nella posizione corrispondente al numero di ordine di attivazione del processo. Nel seguito indicheremo questo numero come *tid logico* in quanto ci permetterà di individuare il tid effettivo del processo all'interno del vettore.

Completata l'attivazione dei processi e ricavati dunque i tid di tutti i task pvm, il vettore in cui questi sono memorizzati viene inviato in multicasting a tutti i processi che, ne segnalano l'accettazione inviando come risposta un messaggio sul canale identificato dal tag **START_ACK_TAG**. A questo punto il processo *p3lrun* si pone in attesa del messaggio di terminazione, etichettato con **END_PROG_TAG**, proveniente dall'ultimo processo in esecuzione (*seq_no_output*). Secondo quanto abbiamo ampiamente discusso nella sezione precedente, quando questo sarà giunto verrà effettuata l'uscita dal sistema mediante l'invocazione del comando `pvm_halt()` che determina la terminazione di tutti i processi ancora attivi ed in particolare dei *pvmd*.

Osserviamo che, in generale possono essere effettuate più attivazioni dello stesso codice eseguibile dando in tal modo origine a più istanze dello stesso processo. Nel seguito parleremo di **istanze di un processo** per indicare i processi ottenuti mediante attivazioni differenti dello stesso codice eseguibile. Nei casi più semplici, ad esempio nel caso dei worker *sequenziali* di un farm o di una map, le differenti istanze del processo attivate appartengono ad uno stesso costrutto e dunque condividono gli stessi partner di comunicazione (ad esempio l'emettitore quale mittente e il collettore quale destinatario).

In presenza di annidamento di costrutti, può accadere, invece, che istanze di uno stesso processo appartengano a costrutti differenti e dunque vadano differenziate in modo da fornire a ciascuna di esse le corrette informazioni per identificare i partner delle comunicazioni.

Ad esempio nel caso in cui un farm abbia come worker un costrutto map, le differenti istanze del costrutto map che costituiranno i worker del farm saranno composte a loro volta da istanze differenti dei medesimi processi. In fig.8 è mostrato il grafo dei processi corrispondente a questa particolare situazione. Per ciascun processo abbiamo indicato informalmente[9] il nome del file del codice eseguibile corrispondente e il numero di attivazione, ovvero il *tid logico* di ciascuna *istanza di processo*. Osserviamo che i processi emettitori delle due map (*3*, *8*) pur essendo attivazioni dello stesso codice eseguibile avranno come destinatari differenti gruppi di processi (rispettivamente {*5, 6, 7*} e {*10, 11, 12*}) e una analoga situazione varrà anche per i worker e i collettori corrispondenti. In pratica questo

[9]Il nome del codice eseguibile di ciascun processo è generato dal compilatore associando al nome del tipo del processo un particolare identificatore numerico, non necessariamente significativo nell'ambito del grafo generato ma dipendente dall'intero procedimento di compilazione. Il numero che affianca ciascun processo corrisponde al relativo numero di ordine di attivazione.

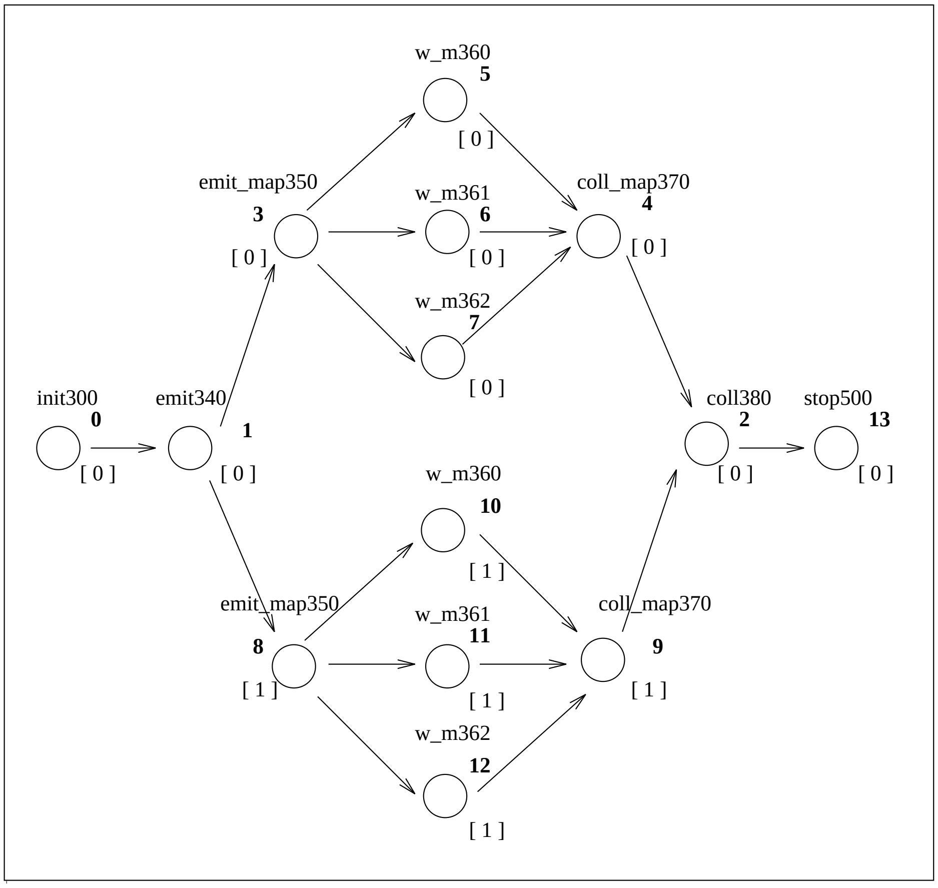

Figura 8. Esempio di grafo con istanze di processi aventi differente numero di costrutto.

significa che i due processi, pur eseguendo lo stesso codice, per effettuare le giuste comunicazioni dovranno utilizzare, e dunque individuare, dei parametri di comunicazione, i tid appunto, differenti.

Per questo motivo il compilatore associa ad ogni istanza di processo un numero d'ordine (**numero di costrutto**), che nell'ambito delle attivazioni di processi con uguale codice distingue i gruppi di istanze con partner di comunicazione differenti. In fig.8 il *numero di costrutto* è indicato tra parentesi quadre accanto a ciascun processo. All'atto della attivazione vera e propria questo numero viene passato, insieme alle eventuali informazioni di partizionamento dei dati (proprie dei processi worker e collettore del costrutto map), come argomento del processo tramite *argv*. Incidentalmente osserviamo che questa soluzione risulta necessaria in quanto tali parametri, ed in particolare il numero di costrutto, rappresentano un tipo di informazione esclusiva per la particolare istanza di processo attivata.

Come ultima cosa osserviamo che i parametri che descrivono la configurazione della macchina astratta (il numero degli host) e l'applicazione (il numero dei processi), utilizzati non solo dal processo *p3lrun* ma da tutti i processi dell'applicazione vengono inclusi dal compilatore sotto forma di costanti all'interno del file **p3l__typedefs.h** insieme a tutti i tag necessari per la definizione dei diversi canali di comunicazione (fig.9).

P^{3}L__typedefs.h

```
..........

#define NUMPROC         N
#define NUMHOST         M
#define TIDS_TAG      900
#define END_PROG_TAG   901
#define START_ACK_TAG  902
#define SEQ_TAG        903
#define SEQ_SYNCH_TAG  904
#define SEQ_MARK_TAG   905
#define LOOP_END_TAG   906
#define LOOP_END_MARK 1005
..........
```

Figura 9. Istruzioni fondamentali del file *p3l__typedefs.h.*

5.2 Il passaggio dei parametri di comunicazione

Il problema principale nell'implementazione delle comunicazioni tra i processi consiste essenzialmente nella determinazione degli identificatori dei processi, i *tid*, mediante i quali vanno riferiti i processi mittenti e destinatari. Abbiamo già visto che questi identificatori sono generati ed assegnati dal sistema PVM ai singoli processi all'atto della loro attivazione e dunque vengono notificati dal *p3lrun* a tutti i processi tramite l'invio del vettore in cui sono stati memorizzati, in ordine di attivazione. Per il singolo processo si tratta dunque di riconoscere tra tutti i *tid* di cui ha conoscenza quelli relativi ai suoi partner di comunicazione.

A tal fine il *back_end* costruisce per ciascun processo un file header (**proc_name_tags.h**) in cui sono contenute opportune informazioni ovvero:

- numero dei processi mittente;
- numero dei processi destinatari;
- una matrice contenente per ciascun *numero di costrutto* l'insieme dei *tid logici* dei processi mittente (**index_mitt**);

- una matrice contenente per ciascun *numero di costrutto* l'insiemedei *tid logici* dei processi destinatari (**index_dest**).

Utilizzando il numero di costrutto ricevuto come argomento di attivazione, ogni processo è in grado di riferire con il giusto parametro di identificazione i processi da cui ricevere o a cui inviare messaggi. Ad esempio, nel caso illustrato in fig.8, le informazioni relative ai partner di comunicazione per i processi emettitori della map (che, ricordiamo, sono soltanto attivazioni diverse dello stesso file eseguibile) saranno contenute in un file *map_emitK_tags.h* e avranno i valori mostrati in fig.10. In funzione del *numero di costrutto*, passato loro come primo argomento all'atto dell'attivazione, i due processi emettitori "*3*" e "*8*" saranno in grado di estrarre dai vettori dei mittenti e dei destinatari gli indici (*tid logici*) per individuare i tid reali dei propri partner di comunicazione all'interno del vettore inviato loro dal processo *p3lrun*.

map_emitK_tags.h

```
#define NUM_MITT 1
#define NUM_DEST 3
index_mitt [2] [NUM_MITT] = {{ 1 }, { 1 }}
index_dest [2] [NUM_DEST] = {{ 5, 6, 7 }, { 10, 11, 12 }}

Proc ''3'':  argv_[0] = [0]
           in_process = tids [index_mitt [0] [0]];
           out_process = tids [index_dest [0] [0]];
Proc ''8'':  argv_[0] = [1]
           in_process = tids [index_mitt [1] [0]];
           out_process = tids [index_dest [1] [0]];
```

Figura 10. Informazioni di comunicazione contenute nel file emit_map_tags.h e loro uso.

Nei casi in cui il processo invia (riceve) messaggi a (da) diversi destinatari (mittenti), ad esempio nel caso del *loop_out* che invia i task elaborati o all'esterno del costrutto o nel canale di feed_back, l'ordine dei relativi tid logici viene considerato fissato per convenzione una volta per tutte. Tale ordine viene seguito dal back_end per l'inizializzazione del vettore dei mittenti (destinatari) così che il template di processo che ne fa uso sia in grado di identificare la giusta posizione dei tid logici desiderati.

Ancora nell'ambito della comunicazione tra processi, una caratteristica propria della macchina astratta considerata consiste nell'esigenza di definire delle funzioni di impacchettamento e spacchettamento per ciascun messaggio inviato o ricevuto da un processo. Poiché la definizione di queste funzioni dipende dal tipo del messaggio, esse devono essere definite dal *back_end*, utilizzando le informazioni di comunicazione ricevute dal *middle_end*. In particolare vengono definite tre tipi di macro per ogni funzionalità, corrispondenti alle componenti tipiche di un generico messaggio ovvero:

- `pk_mark` e `up_mark`, per l'impacchettamento/spacchettamento della marca di terminazione;
- `pk_tags` e `up_tags`, per l'impacchettamento/spacchettamento dei tags di ordinamento;
- `pk_param` e `up_param`, per l'impacchettamento/spacchettamento del messaggio vero e proprio, in generale il task da elaborare o i risultati derivati dall'elaborazione.

Purtroppo anche nei processi che non effettuano alcuna elaborazione sui task ma semplicemente ne gestiscono la raccolta o la distribuzione, è necessario, allo stato attuale, lo spacchettamento e il successivo impacchettamento dell'intero messaggio anche se finalizzato semplicemento al controllo della marca di terminazione ed, eventualmente, all'aggiunta di un tag di ordinamento. Questo inconveniente è determinato dal vincolo imposto dal sistema PVM, per cui lo spacchettamento di un messaggio deve essere effettuato esattamente nello stesso modo in cui ne è stato effettuato l'impacchettamento e non è possibile accedere, in modo parziale, ad un buffer senza pregiudicarne il riutilizzo.

In questi casi, per evitare almeno un'inutile copia dei dati, lo spacchettamento viene effettuato direttamente sulla struttura dati che dovrà essere spedita realizzando al contempo un opportuno procedimento di gestione dei tag di ordinamento. Nel caso di processi di distribuzione (emettitori e *loop_in*), infatti, si dovrà mantenere libera la posizione iniziale del vettore dei tag in uscita per permettere l'inserimento di un nuovo tag relativo al costrutto cui il processo considerato appartiene. Analogamente nei processi di raccolta (collettori e *loop_out*) si dovrà eliminare all'atto dell'impacchettamento il tag relativo al costrutto più interno cui il processo considerato appartiene. Poichè questo tag è comunque necessario al processo per effettuare una corretta gestione dei task, l'intero vettore dei tag associato al task in ingresso viene spacchettato su una struttura ausiliaria, **list_tags**. Successivamente, mediante la funzione ausiliaria `tags_in_out(.)`, definita dal compilatore ed inserita nel file **proc_name_macros.h**, i tag esterni al costrutto (ovvero tutti tranne il primo) vengono copiati nell'area corrispondente al task in uscita per essere impacchettati con esso.

A questo va aggiunto che nei processi relativi al costrutto loop è stato necessario definire due diversi insiemi di funzioni di impacchettamento/spacchettamento corrispondenti ai tipi in uscita/entrata sui diversi canali presenti. In particolare è stata definita anche una ulteriore funzione `tags_in_feed(.)` per effettuare la copia dell'intero vettore dei tag sull'area di impacchettamento relativa ai task in uscita sul canale di feed_back.

Osserviamo che, in realtà, sarebbe possibile utilizzare il tag delle primitive di invio e ricezione per implementare la marca di fine terminazione ed eventualmente il tag del costrutto più interno nella composizione gerarchica dei processi. Questo permetterebbe di evitare del tutto l'inutile impacchettamento/spacchettamento di parametri non utilizzati dal processo ma limiterebbe di molto la generalità delle funzioni e strutture fino a questo momento definite in previsione di loro eventuali modifiche o espansioni.

Un'ultima considerazione riguarda il fatto che le operazioni di impacchettamento e spacchettamento dei dati inglobano anche funzioni di riordinamento dei parametri. Ovvero utilizzando le informazioni di *passing* fornite dal `middle-end` l'impacchettamento dei parametri viene effettuato direttamente secondo l'ordine definito da eventuali cambiamenti di posizione o operazioni di *by_passing* dei parametri di ingresso/uscita, senza ricorrere ad ulteriori funzioni di gestione e dunque evitando inutili copie dei dati.

5.3 Il passaggio dei parametri di partizionamento dei dati

Il partizionamento dei dati riguarda esclusivamente i processi appartenenti al costrutto *map* che modella una particolare forma di parallelismo del tipo *data-parallel*. Abbiamo visto che nel template di implementazione definito in relazione alla nostra macchina astratta, il processo emettitore invia l'intero task da elaborare a tutti i suoi worker i quali, utilizzando gli opportuni parametri, effettueranno ciascuno l'elaborazione esclusivamente sulla partizione che è stata loro assegnata. Analogamente i risultati parziali elaborati dovranno essere ricomposti dal collettore che li riceve, insieme agli indici di ricomposizione, dai worker corrispondenti.

Ciascun worker elabora un numero di sottotask pari al valore *num_task* fornitogli come argomento al momento dell'attivazione. Il singolo sottotask viene estratto dal task ricevuto in ingresso utilizzando degli indici che ne definiscono la posizione all'interno di ciascuna dimensione di partizionamento. Gli indici relativi al primo task da elaborare sono memorizzati all'interno del vettore *dim_index_partition* e successivamente vengono aggiornati utilizzando la funzione *inc_indexes*. Poichè ciascun worker elabora una partizione differente di sottotask, per ognuno sarà memorizzato un insieme di indici differenti all'interno del vettore *dim_index_partition*. Ogni singolo worker sarà in grado di riconoscere le proprie informazioni mediante l'indice di posizione che gli viene passato al momento dell'attivazione. Per una corretta gestione degli indici, la funzione *inc_indexes* utilizza inoltre, come limite superiore di incremento, le dimensioni del partizionamento che, essendo comuni a tutti i worker, vengono memorizzate come primo elemento del vettore *dim_index_partition*.

In definitiva, quindi, le informazioni fondamentali per la decomposizione del task vengono inserite dal compilatore nel file *proc_name_tags.h* relativo al worker della map e consistono essenzialmente in:

- numero di dimensioni su cui avviene il partizionamento (*dim_dim_part*);
- un vettore (*dim_index_partition*) contenente a sua volta delle liste: nella prima sono indicate, per ciascun indice di partizionamento, le dimensioni del partizionamento relativo a quell'indice e, nelle successive, gli indici di partizionamento iniziale per ciascun worker.

Inoltre a ciascuna istanza dei worker al momento dell'attivazione vengono passati, rispettivamente come secondo e terzo argomento, i seguenti parametri:

- il numero di task da elaborare (*num_task*);

- l'indice mediante il quale estrarre dal vettore *dim_index_partition* gli indici che individuano il sottotask con cui iniziare l'elaborazione (*start_task_index*).

In fig.11 abbiamo mostrato in maniera informale il valore di queste strutture dati e il modo in cui vengono estratti i parametri fondamentali all'interno di ciascun worker nel caso in cui l'elaborazione, effettuata su due matrici di interi di dimensione 150*150 partizionate l'una per riga e l'altra per colonna, venga ripartita tra tre worker.

In questo esempio le dimensioni su cui viene effettuato il partizionamento sono due, corrispondenti rispettivamente alle righe e alle colonne delle due matrici, ciascuna avente dimensione 150 (come riportato nella prima coppia di elementi nel vettore *dim_index_partition*). I sottotask da elaborare sono le singole moltiplicazioni riga-colonna e dunque un totale di 22500 operazioni, partizionate in tre gruppi di 7500 sottotask. Come abbiamo già discusso, utilizzando l'argomento *start_task_index* ciascun worker è in grado di individuare, nel vettore *dim_index_partition*, la coppia di indici di partenza del partizionamento, ovvero il proprio task iniziale (*start_task*). In particolare, poiché il partizionamento viene effettuato a partire dagli indici più bassi e dunque, nel nostro caso, partizionando le righe della prima matrice, il primo worker comincerà la sua elaborazione con la prima riga e la prima colonna, il secondo con la riga 50 e la prima colonna, il terzo con la riga 100 e la prima colonna.

Ricordiamo che nella sezione sez.4.5 abbiamo descritto le funzioni utilizzate dai worker per la corretta elaborazione dei sottotask. Ora dovrebbe risultare chiaro che gli indici che permettono di individuare la partizione da elaborare sono forniti inizialmente dal vettore *start_task* e successivamente ottenuti incrementandone i valori mediante la funzione *inc_indexes*. È da notare che l'incremento viene effettuato di volta in volta sull'indice più esterno nel vettore fino a quando questo non raggiunge il limite definito dalla dimensione del partizionamento. Una volta effettuata l'elaborazione, ripetuta un numero di volte pari a *num_task*, la funzione *from_inp_to_out* provvede ad associare al risultato di ciascun sottotask gli indici di partizionamento corrispondenti.

A questo proposito ricordiamo che il worker invia il risultato dell'elaborazione di tutti i sottotask a lui assegnati in un unico messaggio in cui viene incluso anche il numero dei sottotask che servirà al processo collettore sia per lo spacchettamento del task che per la ricomposizione del task globale.

Da parte sua il processo collettore, disponendo di un buffer predisposto per contenere interamente più task finali, man mano che i risultati parziali gli pervengono dai worker, li ricompone utilizzando gli indici ad essi associati e li memorizza in una posizione dipendente dal tag associato al task cui essi appartengono. Quando il task è stato interamente ricomposto e i vincoli di ordinamento risultano rispettati (sez.4.5) esso viene finalmente spedito all'esterno del costrutto.

Dunque anche il processo collettore necessita di opportune informazioni, ovvero:

- il numero di dimensioni su cui avviene il partizionamento (*dim_dim_part*);

```
Map_workK_tags.h
  int dim_dim_part = 2;
  int dim_index_partition [ ] [ 2 ] = {{150, 150}, {0, 0}, {50, 0}, {100, 0}};


Map_work1:
  argv_[1] = num_task = 7500
  argv_[2] = start_task_index = 1

Map_work2:
  argv_[1] = num_task = 7500
  argv_[2] = start_task_index = 2

Map_work3:
  argv_[1] = num_task = 7500
  argv_[2] = start_task_index = 3


Map_worki (i = 1, 2, 3 ):
  dim_part = dim_index_partition [ 0 ]
  start_task = dim_index_partition [ start_task_index ]
```

Figura 11. Informazioni di partizionamento contenute nel file emit_map_tags.h.

- un vettore contenente, per ciascun indice di partizionamento le relative dimensioni (*dim_part*) (osserviamo che in realtà corrisponde al primo elemento del vettore *dim_index_partition*);
- gli indici di ricomposizione di ciascun task.

Mentre le prime due informazioni sono contenute nel corrispondente file *proc_name_tags.h*, gli indici di ricomposizione vengono ottenuti combinando opportunamente gli indici di scomposizione associati a ciascun task e contenuti con esso nel corrispondente messaggio di ingresso. È grazie a questi indici che il processo collettore è in grado di ricostruire il task globale, man mano che i risultati parziali provengono dai vari worker. In particolare i valori delle dimensioni del partizionamento permettono al collettore di calcolare il numero di task elaborati e poter quindi riconoscere quando la ricomposizione di un task è stata completata semplicemente verificando se la somma del numero di sottotask pervenuti è pari al totale calcolato.

5.4 La generazione del codice eseguibile

Il passo finale nella generazione del codice eseguibile per la macchina astratta consiste nella compilazione dei file sorgenti generati dal *back_end* nei passi precedenti e scritti nel linguaggio host sequenziale, che nel caso in esame è il C. Su ciascuna macchina host viene compilato il sottoinsieme dei processi che dovrà essere eseguito su di essa. In realtà per motivi di ottimizzazione i processi sono partizionati tra i vari host in funzione del tipo di architettura. Ovvero il file sorgente, corrispondente a più processi che saranno eseguiti su macchine diverse ma aventi lo stesso tipo di architettura, sarà compilato su di una sola macchina. A tale scopo i file eseguibili prodotti dalla compilazione saranno collocati in una subdirectory della directory in cui si svolge la compilazione, identificata dal codice letterale della particolare architettura considerata (Tabella 1).

Per effettuare in maniera automatica la compilazione facciamo ricorso ad un ulteriore strumento software, **Make** ([Fel79]), che permette di specificare per ciascun file eseguibile gli ulteriori file da cui dipende eventualmente la sua compilazione e, in relazione a tale dipendenza, i comandi da effettuare per garantirne la validità. In tal modo è possibile specificare per ciascun processo il tipo di compilazione da effettuare (ad esempio il particolare compilatore da utilizzare in funzione del tipo di architettura su cui la compilazione sarà eseguita), la collocazione dei file sorgente, le librerie da includere, la collocazione finale degli eseguibili. A tale scopo il *back_end* genera dunque dei *makefile*, uno per ogni host, su cui sarà eseguito il comando `make`. Poiché la compilazione guidata da ciascuno dei makefile dovrà avvenire sulla particolare macchina per cui essi sono stati definiti (e su cui andranno eseguiti i processi compilati), in analogia con l'approccio parallelo dell'intero lavoro sarà possibile effettuarla parallelamente sulle diverse macchine considerate. Per far ciò il *back_end*, genera uno *shell script* denominato **compile**, contenente i comandi di compilazione (`make`) da effettuare eventualmente all'interno di invocazioni di shell remote. È evidente che perché possano essere eseguiti in parallelo è necessario che essi siano effettuati in *background*, dunque l'istruzione finale nello *shell script* sarà un comando di `wait` che assicurerà il ritorno alla shell attuale soltanto dopo che tutte le compilazioni saranno state effettuate. Solo allora sarà possibile eseguire il file *p3lrun* che gestirà l'esecuzione parallela dell'applicazione P^3L in tal modo compilata.

6 Uso del compilatore P^3L

Per utilizzare il compilatore P^3L nella versione che utilizza il supporto PVM è necessaria una breve fase di configurazione del proprio ambiente di programmazione. Faremo riferimento alla configurazione data per l'utilizzo dell'ambiente di programmazione nella rete di workstation del Dipartimento di Informatica dell'Università di Pisa, ma le modifiche da apportare, nel caso si utilizzi un sistema diverso, sono abbastanza ovvie se tale sistema consiste ancora in una rete eterogenea di macchine `Unix`.

6.1 Accesso agli host

Innanzi tutto l'utente deve avere diritto di accesso alle macchine che fanno parte della macchina virtuale PVM. La lista di queste macchine è contenuta nel file di configurazione **/.../p3l/PVMP3L/TIME/config_pvm.pl**. L'accesso deve potere avvenire senza richiesta di login e password da parte del sistema, quindi, deve essere creato un file **/.rhosts** contenente i nomi degli host dai quali ogni richiesta di login remota viene accettata da un'altra macchina senza necessità di verifica dei diritti. Un esempio di tale file per l'utente "susanna" è mostrato in fig.12. Osserviamo che per ogni host è riportato anche il nome simbolico completo per esigenze di sistema.

```
************************  /.rhosts **********************
  pisa-tts susanna
  apollo susanna
  simula susanna
  database susanna
  inti susanna
  pisa-tts.di.unipi.it susanna
  database.di.unipi.it susanna
  inti.di.unipi.it susanna
  simula.di.unipi.it susanna
**********************************************************
```

Figura 12. Esempio di file *.rhosts*

6.2 Configurazione dell'ambiente di shell

Poiché il compilatore P^{3}L è presente in varie versioni, deve essere indicata la versione che si intende utilizzare settando opportune variabili di ambiente. Questo può essere fatto manualmente con i comandi:

```
setenv P3L_INSTALL /.../p3l/PVMP3L
setenv PVM true
unsetenv MEIKO
```

oppure automaticamente inserendo tali comandi nel file `/.login`. Il compilatore trova automaticamente le librerie di PVM ma, per non avere dipendenza con la particolare configurazione del sistema, la locazione della directory principale di PVM deve essere indicata con una variabile di ambiente. Questo permette, inoltre, di potere utilizzare alcuni strumenti di PVM come la *console*, la *console grafica* e il *debugger*. Ciò può essere fatto semplicemente inserendo nel file `/.cshrc` e/o `/.login` le righe mostrate in fig.13 (e contenute insieme alle istruzioni viste sopra nel file **/.../p3l/PVMP3L/config_env**) facendo attenzione che le righe riguardanti PVM siano inserite nella parte non interattiva del file, cioè la parte che viene eseguita anche in caso di login remota. Il fatto di settare tali variabili anche nelle login remote viene utilizzato solo in fase di debugging, non nella normale esecuzione di programmi, e può essere verificato facilmente con il comando:

```
remsh nome_host_remoto env
```

Nei file ~/.cshrc e ~/.login non devono essere presenti, nella parte non interattiva, comandi di I/O in quanto essi impediscono la creazione remota dei processi *pvmd* di PVM.

Per finire facciamo un'ultima osservazione riguardante il particolare ambiente di shell in cui il compilatore P^3L viene invocato. Poiché, come abbiamo già visto (sez.5.4), i file eseguibili prodotti dal compilatore vengono memorizzati a partire dalla directory in cui la compilazione viene invocata, è necessario che il compilatore sia in grado di individuare tale cammino per poter adeguatamente configurare la macchina astratta (sez.5.1). Questa particolare informazione viene estratta dalla variabile PWD che deve essere presente nell'ambiente di shell al momento dell'invocazione del compilatore. In alcuni casi l'ambiente di shell è configurato in maniera tale che la variabile PWD, ad ogni cambio di directory, assuma automaticamente il valore della directory corrente. È necessario dunque verificare che essa sia presente nel proprio ambiente di shell e, in caso contrario, settarla manualmente associandole il valore del *path assoluto sotto NFS* della directory in cui saranno posti i file eseguibili.

```
########## PVM
if (!$?PVM_ROOT) then
   setenv PVM_ROOT /project/P4/pvm3
   setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
   set path=($path $PVM_ROOT/lib)
   set path=($path $PVM_ROOT/lib/$PVM_ARCH)
endif
#################### END_PVM

########## XPVM
if (!$?PVM_ROOT) then
   setenv XPVM_ROOT /project/P4/pvm3/xpvm
   set path=( $path $XPVM_ROOT/src/$PVM_ARCH )
endif

setenv TCL_LIBRARY /usr/local/lib/tcl
setenv TK_LIBRARY /usr/local/lib/tk

if ( !$?MANPATH ) setenv MANPATH
setenv MANPATH /home/p4/project/pvm3/man:$MANPATH
#################### END_XPVM
```

Figura 13. Istruzioni da inserire nei file *.cshrc* e/o *.login*

6.3 Compilazione del programma P^3L

Dopo avere configurato l'ambiente, il programma P^3L viene compilato con il comando:

`p3l nome_file_programma`.

Per le opzioni del comando si rimanda a [Bac94]; ci limitiamo a dire che, nella versione descritta, la configurazione generata per default effettua l'esecuzione del programma unicamente sull'host che ha eseguito la compilazione mentre specificando l'opzione

`p3l -f nome_file_specifica nome_file_programma`

il programma viene eseguito su tutti gli host che compongono la macchina virtuale. La prima volta che un programma viene compilato con l'opzione `-f`, il compilatore chiede all'utente di specificare il numero di worker di ogni map e farm. Queste informazioni vengono memorizzate in *nome_file_specifica* che, quindi, è il nome che l'utente intende utilizzare per tale file. Se si utilizza un file preesistente il compilatore utilizza le specifiche in esso contenute. Al termine della compilazione, il programma P^{3}L risulta trasformato nel codice di un programma PVM. Per compilare tale programma è sufficiente eseguire il comando

`compile`

che esegue la compilazione in parallelo su tutti gli host della macchina virtuale. L'opzione di profiling non è ancora presente nella versione attuale. E' però possibile utilizzare questa funzionalità inserendo la riga `#define PROFILING` all' inizio del file `p3l__typedefs.h` prima di invocare lo script `compile`. In questo modo si possono avere i tempi medi di esecuzione di ogni iterazione per ogni processo del programma.

6.4 Esecuzione del programma P^{3}L

Il programma P^{3}L viene eseguito con il comando

`p3lrun`

che automaticamente esegue lo *start_up* della macchina virtuale e genera tutti i processi che compongono il programma parallelo.

Analogamente al termine dell'esecuzione del programma viene eseguito lo *shut_down* della macchina virtuale. Sullo schermo apparirà un messaggio di terminazione indicante il tempo di esecuzione del programma calcolato a partire dal momento in cui il processo *p3lrun* riceve l'ack di partenza da tutti i processi attivati a quello in cui riceve l'ack di terminazione dal processo *seq_no_output*.

Per quel che riguarda lo *standard output* di ogni processo, esso viene redirezionato nel file **/tmp/pvml.uid** dove *uid* è il codice di identificazione dell'utente; è possibile visualizzare tale file, anche durante l'esecuzione del programma, con il comando

`tail -f /tmp/pvml.uid`

6.5 Debugging dei programmi

L'ambiente di programmazione P^{3}L non è ancora stato integrato con un debugger, quindi correggere i programmi può essere impegnativo se si utilizzano esclusivamente metodi come la visualizzazione di messaggi di diagnostica. In particolare, nell'ambiente PVM, lo stdout di un processo non viene visualizzato finché tale processo non termina la sua esecuzione. Questo rende abbastanza difficile trovare, ad esempio, un errore di indirizzamento o un ciclo infinito. Un metodo può essere quello di fare terminare un processo subito dopo l'emissione di un messaggio. A tale scopo è possibile utilizzare la macro *panic_(stringa)* che visualizza la stringa di caratteri e termina il programma.

Nel caso in cui si familiarizzi leggermente con l'ambiente PVM, è possibile utilizzare gli strumenti di debugging di PVM. Con essi si ha a disposizione una console per il controllo interattivo della macchina virtuale, il debugger e una console grafica (*xpvm*) che permette di avere una visualizzazione delle interazioni tra i processi con la possibilità di avere finestre di debugging e tracing. La console *pvm* e la console grafica *xpvm* devono essere eseguite con il parametro *pvm_config* in modo da indicare loro dove trovare i file eseguibili dei vari processi:

```
pvm pvm_config
```

oppure

```
xpvm pvm_config
```

A questo punto deve essere eseguita la `spawn` di *p3lrun* con l'opzione `-aoff` che evita la creazione e la terminazione della macchina virtuale. Per i dettagli relativi a pvm si rimanda a [GBD+94] facendo presente che il manuale di PVM è anche disponibile *on_line* se si sono settate le variabili di ambiente viste precedentemente; per i dettagli su xpvm si faccia riferimento all'help on_line di tale strumento e ai file di documentazione contenuti nella directory principale di xpvm (`/project/P4/pvm3/xpvm`).

Un aspetto che deve essere tenuto in considerazione, programmando in un ambiente di macchine eterogenee, è quello riguardante le conversioni dei tipi dei parametri delle liste di importazione ed esportazione dei costrutti P^3L. Le varie macchine che compongono la macchina virtuale PVM possono, infatti, avere formati diversi per la gestione degli stessi tipi e diverse tecniche di allineamento. I formati sono automaticamente convertiti dal supporto di PVM ma questo viene fatto solamente se il supporto si accorge che il formato di un certo tipo è diverso per due macchine. Questo significa che se il programmatore P^3L "nasconde" la differenza fra due rappresentazioni i dati non possono essere convertiti correttamente. Supponiamo, ad esempio, che un costrutto P^3L debba elaborare una struttura complicata e il programmatore decida di esportarla trasformandola in un array di caratteri con una operazione di *casting* e di eseguire poi l'operazione inversa nel costrutto che importa la struttura. Poiché per il supporto di comunicazione di PVM i due processi si scambiano un semplice array di caratteri, la conversione dei tipi viene eseguita solo per il tipo carattere e quindi non è garantito che il riportare con un casting l'array ricevuto alla struttura originaria dia il risultato voluto, se le macchine che eseguono i due processi sono di tipo diverso. Sono, quindi, da evitare le gestioni troppo disinvolte dei tipi dei costrutti del linguaggio così come le supposizioni sulle rappresentazioni interne dei dati.

Un'ulteriore osservazione va fatta, infine, per ricordare che la versione attuale del compilatore P^3L per PVM si presenta ancora sotto alcuni aspetti incompleta.

Dunque, poiché non è ancora stata implementata correttamente la gestione dell'ordinamento dei parametri nell'invocazione dei worker della map e del farm, è necessario che l'ordine dei parametri nella definizione del costrutto sia lo stesso che nell'invocazione del worker. Ad esempio il codice:

```
map mand in(int vec_a[10], int vec_b[10], int k) out(int mat[10][10])
    pixel in(k, vec_a[*i], vec_b[*j]) out(mat[*i][*j])}
end map
```

dovrà essere sostituito con il codice:

```
map mand in(int vec_a[10], int vec_b[10], int k) out(int mat[10][10])
    pixel in(vec_a[*i], vec_b[*j], k) out(mat[*i][*j])}
end map
```

Analogamente non è possibile allo stato attuale compilare un costrutto *map* con un unico *worker*.

È evidente che soltanto una fase di test prolungata permetterà di verificare la correttezza del compilatore in ogni sua parte. Nell'attesa che gli eventuali problemi riscontrati siano risolti essi verranno riportati in un ulteriore specifico documento.

7 Conclusioni e sviluppi futuri

Come abbiamo più volte ribadito, il lavoro qui documentato è ancora in fase di completamento e dunque ci sembra ragionevole attribuire uguale importanza tanto agli obiettivi raggiunti quanto alle direzioni di sviluppo individuate. Dalle riflessioni fatte fino a questo momento, è possibile concludere che l'implementazione dell'ambiente di programmazione P^3L per l'ambiente PVM rappresenta un passo importante nella portabilità delle applicazioni parallele in quanto tale ambiente è disponibile sia per reti di calcolatori eterogenei che, con le opportune modifiche, per diversi multicomputer. Una prima implementazione del compilatore del linguaggio per la nuova architettura target, per quanto poco raffinata, è stata completata. Dunque si è raggiunto il duplice scopo di:

- focalizzare e risolvere alcune problematiche relative alla generazione del codice eseguibile, quali l'implementazione dei protocolli di comunicazione tra i processi sulla macchina astratta (ovvero la generazione dei parametri di comunicazione e la definizione delle opportune funzioni di impacchettamento e spacchettamento) o della distribuzione dei parametri di partizionamento nel costrutto *map*;
- realizzare dei template implementativi funzionanti per i costrutti del linguaggio su cui basare gli studi successivi.

Affinchè l'implementazione realizzata possa essere in grado di garantire buoni livelli di efficienza è necessario approfondire lo studio dei rapporti che legano tra loro la struttura delle applicazioni, le loro possibili implementazioni e il tipo di architettura target considerato. In particolare è necessario determinare:

- dei *modelli analitici di performance* per i template di implementazione che abbiamo definito, che permettano al compilatore di determinare i parametri ottimi per definire la migliore implementazione possibile e a noi di verificare la validità delle scelte implementative effettuate;

- *misure reali dei tempi di computazione* (anche in relazione alla multiprogrammazione) e, soprattutto, di comunicazione in funzione del numero dei processi attivati, della grandezza dei messaggi scambiati, del tipo di comunicazione, locale (tra processi allocati sullo stesso host) o meno, effettuata. Il problema maggiore nella determinazione di questi valori risiede nel traffico della rete attualmente utilizzata (una sottorete della rete disponibile presso il Dipartimento di Informatica dell'Università di Pisa) che incide fortemente sulla attendibilità delle misurazioni. Delle misurazioni più attendibili saranno presto effettuate su una macchina SP2 disponibile presso l'Istituto CNUCE di Pisa.

Sulla base di questi modelli e di valori di tempo attendibili sarà possibile determinare successivamente:

- eventuali modifiche dei *template di mapping*, che tengano conto di caratteristiche proprie dell'architettura target quali ad esempio l'alto rapporto tra i tempi di comunicazione e i tempi di computazione oppure il limitato numero di risorse computazionali. Tra l'altro si dovrà valutare se e in quali casi la distribuzione di funzionalità in più processi costituisca un elemento di degradazione dell'efficienza della implementazione e non sia dunque preferibile, in alcuni casi, ridefinire i template utilizzando un minore numero di processi (ad esempio, nel farm sostituire emettitore e collettore con un solo processo distributore);

- eventuali modifiche dei *template dei processi*, soprattutto in relazione alle scelte riguardanti il partizionamento dei dati (per il costrutto map) e la temporanea memorizzazione dei task da elaborare (nei processi emettitore) o dei risultati (nei processi collettore) da inviare all'esterno del costrutto;

- l'integrazione delle funzioni di *mapping* dei costrutti sulla macchina astratta e di partizionamento dei processi sui nodi della macchina fisica. La semplice funzione di partizionamento attualmente adottata si basa sulla uniformità delle risorse computazionali e della rete di interconnessione adottata. Evidentemente non tiene conto dei tempi delle comunicazioni locali introdotti dalla multiprogrammazione nè del traffico delle comunicazioni tra particolari tipi di processi. D'altra parte la multiprogrammazione stessa dovrebbe essere evitata introducendo degli opportuni algoritmi di riduzione del grafo dei processi, che permettano di definire una rete di processi in cui il carico risulti bilanciato e il numero di processi sia non superiore al numero delle risorse disponibili. L'analisi e le eventuali ridefinizioni dei template di mapping, inquadrati in quest'ottica più ampia, costituiscono uno strumento da cui l'integrazione auspicata proviene direttamente.

A questi passi fondamentali che, ai fini di una corretta implementazione della metodologia di programmazione considerata costituiscono il necessario completamento del nostro lavoro, si può aggiungere l'implementazione degli altri costrutti data parallel del linguaggio e l'eventuale definizione di nuovi costrutti che esprimano forme di parallelismo in grado di sfruttare a pieno le caratteristiche della nuova architettura considerata.

Bibliografia

[Bac94] B. Bacci. *P^3L: Manuale Utente*, January 1994. Rapporto Interno gruppo P4.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[DDO+92] M. Danelutto, R. DiMeglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the suppport of Massively Parallel Programs. *Future Generation Computer Systems J.*, 8:205–220, 1992.

[Fel79] S.I. Feldman. Make - A Program for Maintaining Computer Programs. *Software, Practice and Experience*, 9:255–265, 1979.

[GBD+94] G.A. Geist, A. Beguelin, J. Dongarra, J. Weicheng, R. Manchek, and V Sunderam. *PVM 3: User's Guide and Reference Manual*, September 1994.

[Hey89] A.J.G. Hey. Experiments in MIMD Parallelism . In *Proc. of Int. Conf. PARLE '89*, pages 28–42, Eindhove, The Netherlands, June 1989.

[Kun88] H.T. Kung. Computational models for parallel computers. In C.A.R. Hoare Series, editor, *Scientific applications of multiprocessor*, volume NATO ASI Series, F55, pages 1–17. Prentice-Hall International, 1988.

[Orl92] S. Orlando. The P^3L language manual. Technical Report HPL-PSC-92-52, Hewlett Packard Laboratories, Pisa Science Center (Italy), 1992.

[Pel93] S. Pelagatti. *A methodology for the development and the support of massively parallel programs*. PhD thesis, Dipartimento di Informatica, Università di Pisa - Italy, March 1993.

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, and R Manchek. The PVM Concurrent Computing System: Evolution, Experiences and Trends. *Journal of Parallel and Distributed Computing*, 20, 1994.

[Sun90] V.S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.