Progetto HPCC/SEA

"Integrazione di task e data-parallelism: un nuovo linguaggio ospite per SkIE–CL *(rapporto finale)*

Claudio Gennaro

claudio@pisa.quadrics.com

Salvatore Orlando

orlando@unive.it

Raffaele Perego

r.perego@cnuce.cnr.it

Agosto 1999

L'obiettivo di questo lavoro di ricerca è di estendere l'ambiente di programmazione SkIE ed i relativi strumenti di programmazione visuale (VSkIE) per permettere l'esplicitazione di nuovi costrutti *data–parallel* all'interno di forme tipicamente *task–parallel* come *pipeline* e *processor farm*. Per questo scopo si è scelto di impiegare il linguaggio data-parallel High Performance Fortran (HPF) come nuovo linguaggio ospite di SkIE–CL.

Questo lavoro di ricerca rappresenta la prosecuzione del lavoro preliminare di fattibilità realizzato nel dicembre 1998. Per l'integrazione di HPF si è reso necessario risolvere molti problemi legati all'impiego di due distinti supporti run–time, quello di SkIE–CL e quello del particolare compilatore HPF impiegato.

Il Capitolo 1, rivolto ai programmatori, inizia illustrando le caratteristiche del compilatore HPF (ADAPTOR) scelto per l'esplicitazione dei task data-parallel. La sezione che segue mostra come utilizzare HPF all'interno dell'ambiente di sviluppo SkIEe le modifiche apportate alla sintassi di SkIE–CL. In particolare viene illustrato come introdurre uno o più stadi HPF all'interno di un programma SkIE–CL, come variare il loro grado di parallelismo e le restrizioni sull'uso di HPFin SkIE. Inoltre viene mostrato come adoperare HPF all'interno dell'ambiente di sviluppo visuale VSkIE.

Il Capitolo 2 illustra le problematiche relative all'integrazione di HPF nel linguaggio di coordinamento SkIE–CL. La soluzione proposta consiste nella realizzazione di un template SkIE scritto HPF.

Il capitolo 3 descrive infine un caso di studio, l'implementazione di una FFT tridimensionale, e le relative prestazioni ottenute su una macchina parallela reale.

Indice

1 Manuale d'uso **1**

1.1 Introduzione 1

1.2 Il compilatore `HPF ADAPTOR 6.0` 1

1.3 Requisiti per l'utilizzo di `HPF` in SkIE–CL 2

1.4 Uso di `HPF` in SkIE–CL 3

1.5 Note sulle variabili d'ambiente 5

1.6 Restrizioni sul'uso di `HPF` in SkIE–CL 5

1.7 L'ambiente di sviluppo visuale VSkIE 6

2 Problematiche e soluzioni **9**

2.1 Introduzione 9

2.2 Descrizione del problema 9

2.3 Loading dei task e relativi comunicatori 11

2.4 Startup del programma SkIE–CL 13

2.5 Struttura del template di uno stadio data-parallel `HPF` 17

3 Un esempio di applicazione **25**

A Modifiche apportate al supporto DALIB **43**

B Il CDROM allegato **45**

Capitolo 1

Manuale d'uso

1.1 Introduzione

Questa sezione illustra l'uso del liguaggio HPF all'interno dell'ambiente di sviluppo SkIE–CL. Dopo una breve introduzione al compilatore HPF ADAPTOR, viene illustrato la sintassi, i requisiti di sistema e le restrizioni d'uso del nuovo linguaggio ospite HPF.

1.2 Il compilatore HPF ADAPTOR 6.0

ADAPTOR (Automatic DAta Parallelism TranslatOR) è un compilatore HPF di pubblico dominio sviluppato in questi ultimi anni dall'istituto SCAI (GMD). Lo strumento trasforma programmi data–parallel scritti in HPF in programmi di tipo *message passing*. Al di là di alcune restrizioni, ADAPTOR supporta il linguaggio HPF 2.0 come pure alcune estensioni approvate all'interno dello standard. ADAPTOR è costutuito dai seguenti componenti:

- il traduttore sorgente–sorgente *fadapt*;
- la libreria *DALIB* (Distributed Array LIBrary) che costituisce il supporto run–time per la gestione dei descrittori e le distribuzioni degli array e fornisce l'implementazione delle routine di comunicazione;
- il coordinatore di compilazione *gmdhpf*.

La Figura 1.1 mostra come lavorano insieme i diversi componenti di ADAPTOR. Il coordinatore gmdhpf invoca il traduttore sorgente–sorgente che genera un programma SPMD di tipo *message passing* a partire del programma HPF. Dopodiché, invoca un compilatore nativo FORTRAN 77 o

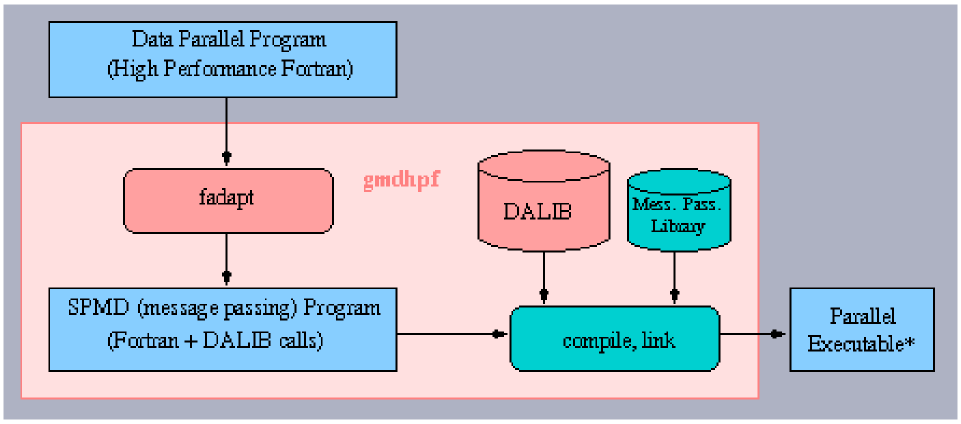

Figura 1.1: Struttura del compilatore ADAPTOR.

Fortran 90 per la compilazione del codice generato. Il codice oggetto generato viene linkato con supporto run–time DALIB.

Il traduttore sorgente–sorgente fadapt, la libreria DALIB e il coordinatore gmdhpf sono disponibili come programmi C e richiedono un compilatore standard C.

Per la compilazione del codice generato, linkaggio ed esecuzione sono necessari i seguenti componenti:

- un sottosistema message passing MPI o PVM, per alcuni sistemi (IBM SP, Intel Paragon, CM 5 and Meiko CS–2) può essere utilizzato il loro sistema di message passing nativo;
- un compilatore FORTRAN 77 (se si usa la sintassi standard degli array FORTRAN). Se il programma HPF contiene moduli, derived data types, puntatori, o utilizza qualche funzionalità speciale del Fortran 90, è necessario utilizzare un compilatore Fortran 90;
- un ambiente di sviluppo parallelo in cui possa essere caricato ed eseguito il programma generato.

1.3 Requisiti per l'utilizzo di HPF in SkIE–CL

Per utilizzare HPF come linguaggio ospiste di SkIE–CL sono necessari i seguenti componenti:

- il compilatore SkIE–CL aggiornato per supportare HPF;
- lo script di configurazione aggiornato per supportare HPF;

- il compilatore HPF ADAPTOR 6.0 correttamente installato con la versione del sistema operativo e MPI utilizzato da SkIE–CL;
- la libreria DALIB modificata per essere utilizzata con i programmi SkIE (vedi A).

La versione del compilatore SkIE–CL e HPF, i relativi script e la libreria DALIB modificata sono state prodotte nelle versioni per reti di workstation Linux e per il sistema operativo Solaris 2.6.

1.4 Uso di HPF in SkIE–CL

L'utilizzo di HPF all'interno di SkIE–CL è del tutto analogo a quello di un qualsiasi altro linguaggio ospite di SkIE–CL come C o FORTRAN. Esistono però delle lievi differenze che saranno illustrate in questo paragrafo.

Una prima differenza rispetto agli altri linguaggi ospite consiste nel fatto che un task scritto in HPF può costituire un stadio di un *pipe*, uno stadio di un *loop* o un *worker* di un *farm*. Poiché sia HPF sia il costrutto *map* hanno lo scopo di permettere la realizzazione di task data–parallel all'interno di SkIE–CL, si è ritenuto l'utilizzo di HPF in SkIE–CL "alternativo" a quello del *map*. Questo significa che una foglia HPF non può costituire un worker di una map.

Il codice di un task HPF deve essere racchiuso, come di consueto, all'interno dei separatori $hpf{ }hpf$. Per questioni che saranno spiegate più avanti la parte di codice che contiene definizioni di variabili distribuzioni ed eventuali inizializzazioni deve essere separata dalla sezione che viene eseguita ad ogni attivazione del sequenziale HPF. Questa separazione avviene attraverso il commento speciale !skie$ all'interno del codice HPF come mostrato in Figura 1.2. Il codice che precede il commento !skie$, chiamato *preambolo* (linee 4–8 dell'esempio), viene eseguito una volta sola alla prima attivazione del task. Il preambolo può contenere la definizione di variabili utilizzate all'interno del modulo e direttive HPF per la distribuzione degli array (definizioni e distribuzioni possono trovarsi in qualsiasi ordine fra di loro secondo la sintassi di HPF) seguite da un eventuale codice di inizializzazione le variabili definite nel modulo. Le variabili così definite (nel nostro esempio linee 4–5), ed eventualmente inizializzate (linea 8), "vivono" per tutta la durata del programma SkIE. Le distribuzioni possono essere riferite sia alle variabili definite internamente al modulo sia alle variabili dell'interfaccia del modulo HPF (linea 6). Al di là del commento separatore (linea 10) non possono esserci dichiarazioni di variabili ed eventuali inizializzazioni saranno eseguite per ogni esecuzione del modulo sequenziale. Se il separatore !skie$ non compare all'interno del

```
example_hpf in(double x[N][N]) out(double y[N][N])
$hpf{

integer z(10,10)
integer i, j
!hpf$ distribute (BLOCK,BLOCK) :: x, y

z = 0

!skie$ %%%%%%% IT IS JUST A SEPARATOR %%%%%%%%

forall (i=2:N-1, j=2:N-1)
    y(i,j) = (x(i,j) + x(i-1,j) + x(i+1,j)
   &     + x(i,j-1) + x(i-1,j-1) + x(i+1,j-1)
   &     + x(i,j+1) + x(i-1,j+1) + x(i+1,j+1) )/9
end forall

}hpf$

end
```

Figura 1.2: Esempio di utilizzo di HPF all'interno di SkIE–CL.

codice del modulo HPF, SkIE–CL genera un messaggio di *warning* avvisando che tutto il codice del modulo sarà interpretato come preambolo.

Allo stato attuale dell'integrazione durante la fase di compilazione del codice HPF, ADAPTOR produce dei messaggi di errore che si riferiscono al codice generato dal SkIE–CL. È fatto carico all'utente di aprire il sorgente HPF generato da SkIE–CL per verificare a quale linea del codice SkIE di partenza si riferiscono i messaggi.

Per scegliere il grado di parallelismo di un data–parallel HPF è sufficiente utilizzare lo stesso pragma utilizzato per il farm. Ad esempio se si vuole impiegare un grado di parallelismo 8 per l'esecuzione del data–parallel HPF chiamato Pluto basta inserire la seguente riga di codice:

```
pragma parallelism degree 8 in Pluto;
```

Di default il SkIE–CL sceglie un grado di parallelismo 2.

Per semplificare l'uso di HPF è stata introdotta la trasposizione automatica degli array ricevuti (spediti) attraverso i canali degli stream da (a) un data–parallel HPF. Se l'utente non vuole che venga eseguita nessuna trasposizione deve utilizzare il pragma fortran_array_in (per gli array ricevuti) e fortran_array_out (per gli array spediti) nel seguente modo:

```
pragma fortran_array_in in Pluto;
```

```
pragma fortran_array_out in Pluto;
```

1.5 Note sulle variabili d'ambiente

Lanciando lo script di configurazione di SkIE–CL vengono settate alcune variabili d'ambiente necessarie per gestire la fase di compilazione HPF. Anche se in generale non dovrebbe essere necessario modificarle è bene dare un sguardo al loro significato:

- `S_HPF`, indica il nome del compilatore HPF utilizzato. Di default è settato a `gmdhpf` e non deve essere modificato;

- `S_HPFFLAGS`, indica i flag passati al compilatore gmdhpf per la compilazione dei moduli HPF. Di default è settato a `-Wa"-strip 72 -free"`. Lo switch `-Wa` indica che le opzioni all'interno degli apici riguardano la fase di traduzione da HPF a FORTRAN 77 (`fadapt`). `-strip 72` indica che il codice FORTRAN 77 generato non deve superare le 72 colonne; come richiesto dal compilatore `f77`. `-free` indica a `fadapt` che il codice di partenza HPF è scritto "liberamente" senza rispettare l'incolonnamento tipico dei programmi FORTRAN. Se il codice HPF introdotto all'interno di un modulo SkIE rispetta lo standard di incolonnamento del FORTRAN è bene disabilitare l'opzione `-free`. Il flag `-Wf"-fno-second-underscore"` è necessario solo se usa come compilatore F77 il g77. `-fno-second-underscore` viene passato da gmdhpf al compilatore F77.

- `S_LIB_LIST`, indica in generale le librerie necessarie a SkIE–CL per la crezione del programma eseguibile. Oltre alle librerie standard è necessario indicare la libreria DALIB modificata (di solito posta in `/skie/lib/dalib.a`) e la libreria standard `-lf2c` (o `-lg2c` in alternativa).

1.6 Restrizioni sul'uso di HPF in SkIE–CL

In questo paragrafo sono riassunte tutte le restrizioni relative all'uso di HPF all'interno di SkIE:

- un task HPF può essere definito come uno stadio di un pipe, loop o worker di un farm. Non sono previsti al momento altri usi;

- gli stream utilizzati da un task HPF non possono essere espliciti;

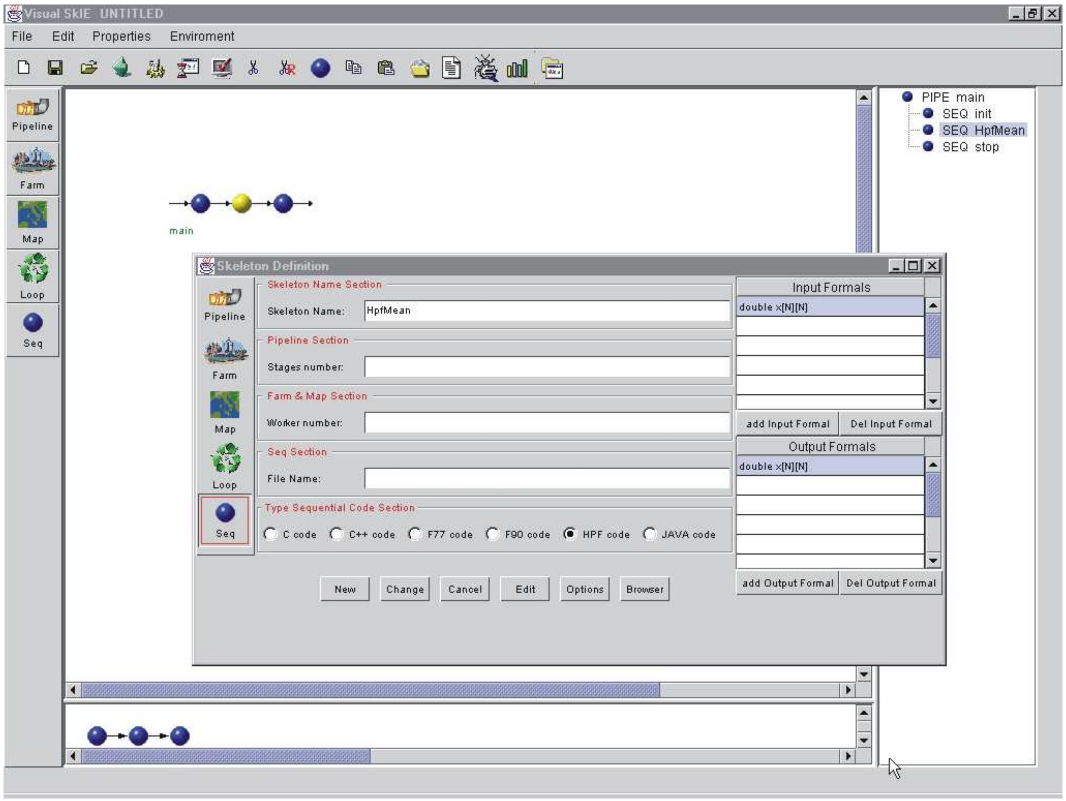

Figura 1.3: Introduzione di un task data–parallel HPF.

- una foglia HPF non può essere il primo stadio del pipe principale di un programma SkIE.

1.7 L'ambiente di sviluppo visuale VSkIE

Attraverso l'ambiente di sviluppo VSkIE è possibile creare applicazioni SkIE utilizzando una comoda intefaccia grafica [1]. La struttura dell'applicazione viene identificata attraverso un grafo composto da nodi connessi con archi. Poiché dal punto utente HPF appare come un linguaggio sequenziale l'estensione di VSkIE per supportare HPF è stata semplificata. Per introdurre un nuovo task data–parallel HPF bisogna operare come se s'introducesse un nuovo task sequenziale in un qualsiasi altro linguaggio ospite (vedi Figura 1.3). Dopo aver introdotto l'interfaccia di comunicazione `in/out` del task si inserisce il codice HPF (vedi Figura 1.4). A partire dal grafo dell'applicazione ed il codice all'interno dei task sequenziali VSkIE è in grado di generare i sorgenti in SkIE da compilare attraverso SkIE–CL.

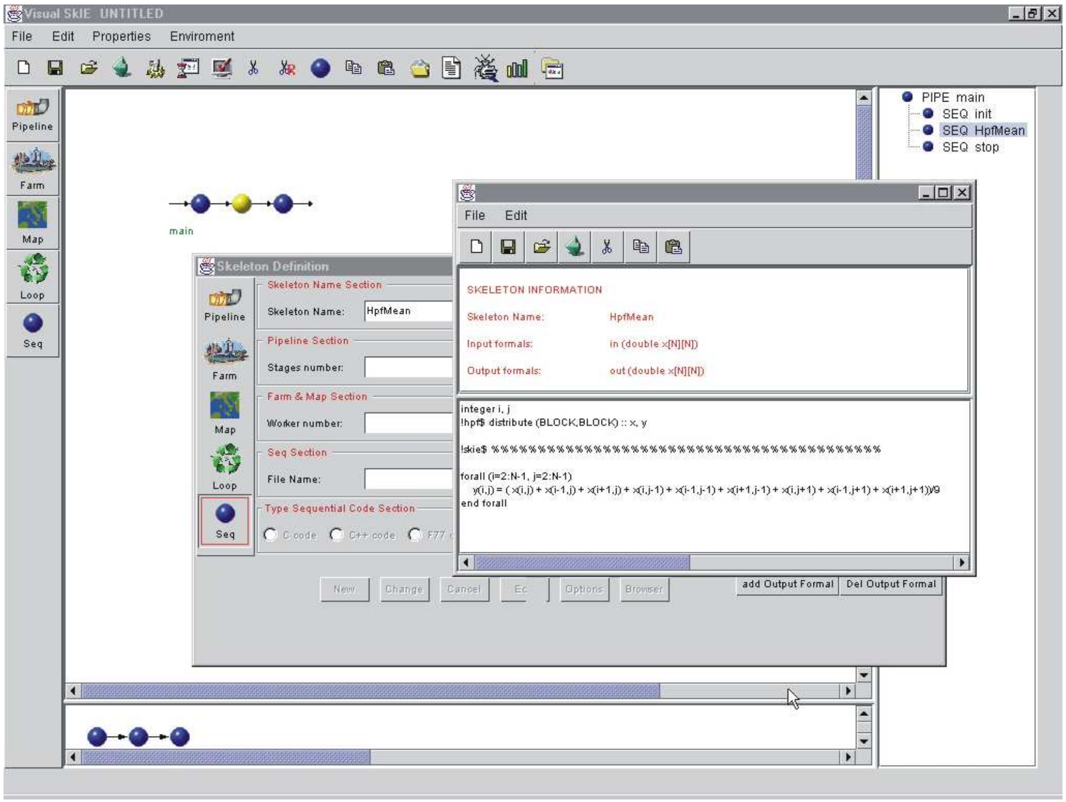

Figura 1.4: Introduzione del codice `HPF`.

Capitolo 2

Problematiche e soluzioni

2.1 Introduzione

In questa sezione vengono presentate le problematiche relative all'integrazione di HPF nel linguaggio di coordinamento SkIE–CL [6] e le soluzioni proposte. La presentazione verte su un semplice esempio, un pipeline di tre stadi dove il primo e il terzo sono semplici processi sequenziali scritti in C, mentre lo stadio intermedio è un processo data–parallel espresso in HPF [3, 4, 5] e compilato tramite ADAPTOR [2].

La presentazione parte dalle caratteristiche del supporto run–time di ADAPTOR, e introduce le modifiche necessarie per la sua integrazione nel supporto a template di SkIE–CL. I problemi considerati sono quelli dell'uso di MPI come strato di comunicazione tra processi, che dovrà essere impiegato sia dal supporto run–time di ADAPTOR per le comunicazioni relative ad HPF, sia da quello a template di SkIE–CL per le comunicazioni inter–task.

Inoltre, viene presentato il meccanismo impiegato per permettere ai task data–parallel HPF di invocare le comunicazioni inter–task, per le quali è stato proposto l'uso di particolari subroutine HPF EXTRINSIC di tipo SERIAL. In altri termini, se un dato costrutto SkIE–CL è basato su di un task HPF, i *template* corrispondenti, attraverso l'impiego del meccanismo delle subroutine EXTRINSIC, possono invocare semplici procedure C per le comunicazione inter–task e per la gestione degli stream SkIE–CL.

2.2 Descrizione del problema

Per capire il problema si consideri il caso di un pipe a tre stadi, dove si impiega MPI per le comunicazioni inter–task, come appunto avviene nel supporto di SkIE–CL. In particolare:

- il 1º e il 3º stadio del `pipe` sono *task sequenziali* scritti in `C`, mentre il 2º è un task data–parallel scritto in `HPF`;
- ogni task (*sequenziale* o data–parallel) viene assegnato ad un set di processori *disgiunti*;
- i task comunicano tra di loro tramite chiamate ad MPI che impiegano il comunicatore globale `SKIE_COMM_WORLD`

Qui di seguito è mostrato il codice SkIE del nostro caso di studio:

```
const int N=64;
pipe main in() out()
  init   in()       out(stream of double x[N][N])
  HpfMean    in(x)      out(double y[N][N])
  stop   in(stream y) out()
end pipe

init in() out(stream of double x[N][N])
$c{
  int k,i,j;
  for (k=1; k<=20; k++)
  {
        for (i=0; i<N; i++)
         for (j=0; j<N; j++)
          x[i][j] =  rand()/RAND_MAX;
        SKIE_out();
  }
}c$
inc(math.h, stdlib.h)
end

stop in (stream of double y[N][N]) out()
$c{
  static int i=0;
  while (SKIE_in() != SKIE_EOS)
  {
        i = i+1;
        printf("matrice n. %d elaborata\n",i);
  }
}c$
inc(stdio.h)
end

pragma parallelism degree 4 in HpfMean;

HpfMean in(double x[N][N]) out(double y[N][N])
$hpf{

```

```
integer i, j
!hpf$ distribute (BLOCK,BLOCK) :: x, y

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

forall (i=2:N-1, j=2:N-1)
    y(i,j) = (
   &        x(i,j) + x(i-1,j) + x(i+1,j)
   &      + x(i,j-1) + x(i-1,j-1) + x(i+1,j-1)
   &      + x(i,j+1) + x(i-1,j+1) + x(i+1,j+1)
   &      )/9
end forall

}hpf$
end
```

Il 1^o stadio (`init`) produce uno stream di matrici casuali di `double`, lo stadio `HPF` (`HpfMean`) riceve le matrici e produce in uscita matrici della stessa dimensione, il 3^o stadio (`stop`) stampa semplicemente un messaggio per ogni matrice elaborata. In Figura 2.1 è illustrata la struttura la struttura del nostro caso di studio.

2.3 Loading dei task e relativi comunicatori

Il problema da risolvere è quello dell'uso di `MPI` come strato di comunicazione a basso livello per tutti i tipi di comunicazione, sia quelli del supporto "a template" di SkIE–CL per le comunicazioni inter–task, sia quelle relative ad `HPF` e implementate all'interno del supporto run–time di `ADAPTOR`.

L'idea alla base della soluzione proposta è la seguente: lo *splitting* del comunicatore globale di `MPI` (`MPI_COMM_WORLD`) in tanti *sottocomunicatori*, uno per ciascun gruppo disgiunto di processori virtuali, dove a ciascun task, sequenziale o data–parallel, è assegnato ad un gruppo distinto di processori. Per questo motivo, indicheremo con il termine *comunicatore locale* il sottocomunicatore (`new_comm`), che ciascun processore virtuale `MPI` vedrà come il comunicatore che comprende tutti i processori del proprio gruppo. È utile far notare che, nel caso in cui i task siano sequenziali, tali comunicatori locali comprenderanno un singolo processore. La Figura 2.1 illustra appunto la suddivisione del comunicatore globale in tre comunicatori locali.

L'esistenza di un comunicatore locale permette una semplice modifica del supporto run–time di `ADAPTOR`: tutte le comunicazioni necessarie per l'implementazione SPMD di programmi data–parallel, ovvero i programmi prodotti dal compilatore `ADAPTOR` a partire dal codice sorgente scritto in `HPF`, devono essere modificati in modo che facciano riferimento al particolare comunica-

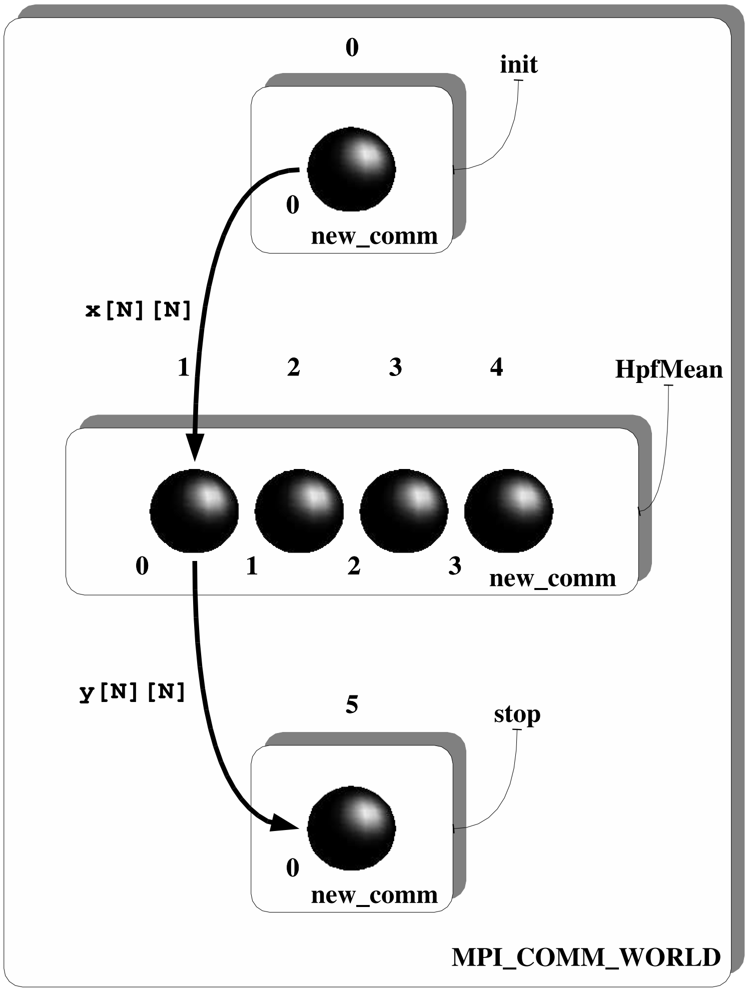

Figura 2.1: Struttura del pipe (caso di studio) con lo splitting del comunicatore globale.

tore locale (new_comm) invece di quello globale. Questo vale anche per le chiamate ad MPI che il supporto di ADAPTOR effettua per conoscere a tempo di esecuzione il numero di processori virtuali disponibili.

2.4 Startup del programma SkIE–CL

Per realizzare l'idea illustrata sopra, è necessario modificare la funzione main(), del programma generato dal compilatore, che carica i vari task (sequenziali e data–parallel) invocando l'uno o l'altro sulla base della conoscenza del mapping dei task sui processori. Prima di effettuare il caricamento, il main dovrà però effettuare lo splitting del comunicatore MPI_COMM_WORLD, in modo che il nuovo comunicatore locale ad ogni task (parallelo o sequenziale) diventi new_comm. Come si vedrà più avanti, per ogni data–parallel HPF il processore di rank locale 0 gestisce le comunicazioni con gli stadi adiacenti del programma SkIE. Chiameremo questo particolare processore Master del data–parallel HPF. Per semplificare l'integrazione dei task HPF e per modificare il meno possibile il *middle–end* di SkIE–CL, prima dello splitting di cui si è parlato, è stato scelto di raggruppare tutti i processori assegnati ai task sequenziali e i processori Master dei data–parallel HPF nel comunicatore standard usato da SkIE per le comunicazioni (SKIE_COMM_WORLD). In questo modo i processori dei data–parallel HPF diversi dal Master rimangono "invisibili" all'ambiente di comunicazione SKIE_COMM_WORLD, come mostrato in Figura 2.2. Il vantaggio è quello di non dover modificare gli algoritmi di assegnamento dei rank dei processi coinvolti nelle comunicazioni, utilizzando quelli generati da SkIE–CL quando al posto dei data–parallel HPF ci sono dei normali sequenziali.

Di seguito è mostrato il codice semplificato del main generato dal compilatore SkIE–CL a partire dal nostro caso di studio.

```
MPI_Comm new_comm;

int main(int argc, char ** argv)
{
  int me;
  int my_color;
  int virtual_ranks[3] = {0, 1, 5};
  MPI_Group grp_total, grp_skie;

  MPI_Init(&argc,&argv);

  MPI_Comm_group(MPI_COMM_WORLD, &grp_total);
  MPI_Group_incl(grp_total, 3, virtual_ranks, &grp_skie);
  MPI_Comm_create(MPI_COMM_WORLD, grp_skie, &SKIE_COMM_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD,&me);
```

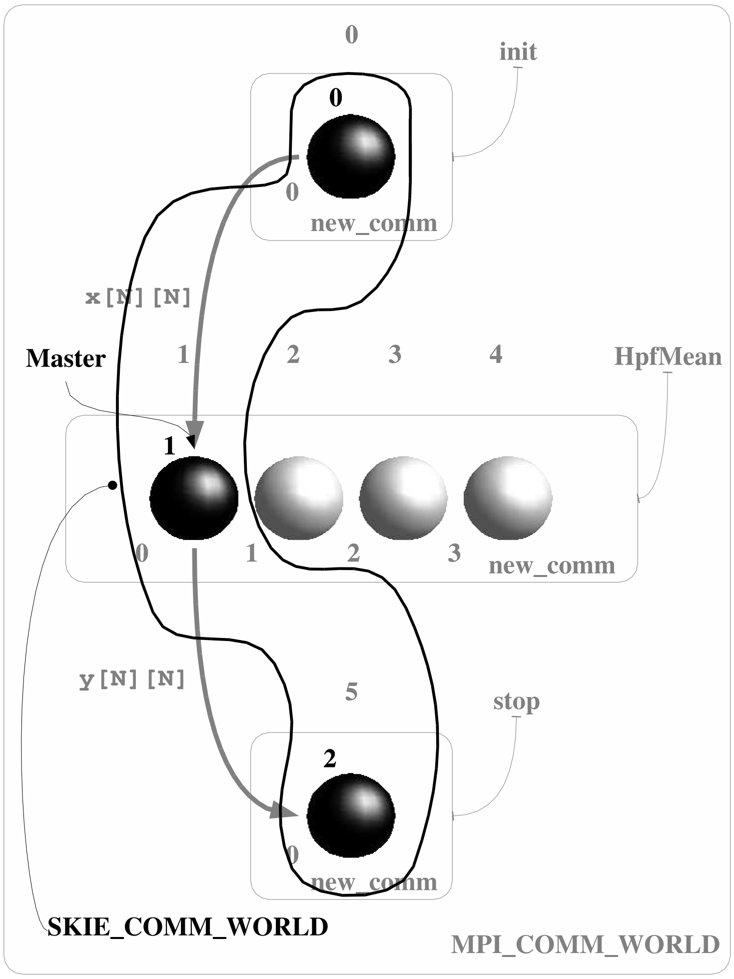

Figura 2.2: Raggruppamento dei rank del comunicatore `SKIE_COMM_WORLD`.

```
    my_color = 0;
    switch (me)
    {
      case 0:
        {
          my_color = 0;
          break;
        }
      case 1:
      case 2:
      case 3:
      case 4:
        {
          my_color = 1;
          break;
        }
      case 5:
        {
          my_color = 2;
          break;
        }
      default:
        signal_error(_S_TOO_NODE,USER_FILENAME);
    }

    MPI_Comm_split(MPI_COMM_WORLD, my_color, me, &new_comm);

    switch (my_color)
    {
      case 0:
        {
          ProcTemp_0(argv);
          break;
        }
      case 1:
        {
          /* Start an hpf task with parallel degree 4 */
          proctemp_1__();
          break;
        }
      case 2:
        {
          ProcTemp_2(argv);
          break;
        }
      default:
        signal_error(_S_TOO_NODE,USER_FILENAME);
    }
```

```
65
66     MPI_Finalize();
67
68     return (0);
69   }
70
```

La varibile `me` (linea 5) viene settata con il rank del processore corrente (linea 15). `my_color` è una variabile che definisce il "colore" del processore corrente cioè a quale sottocomunicatore `new_comm` appartiene. Sarà assegnato un colore diverso per ogni processore che esegue un task sequenziale e per ogni data–parallel `HPF`. In questo modo i processori appartenenti allo stesso data–parallel `HPF` hanno lo stesso colore. Il vettore `virtual_ranks` contiene i rank, rispetto al comunicatore globale `MPI_COMM_WORLD`, dei processi che saranno raggruppati nel comunicatore `SKIE_COMM_WORLD`. L'indice del vettore rappresenta il nuovo rank rispetto al comunicatore `SKIE_COMM_WORLD`. Quindi in generale se per il processore, con rank globale j, esiste un i tale per cui `virtual_ranks[i] == j`, allora esso assumerà il nuovo rank i nel comunicatore `SKIE_COMM_WORLD`. Viceversa, se il rank globale j non si trova in `virtual_ranks` allora non verrà incluso nel comunicare `SKIE_COMM_WORLD` (è il caso dei processori di un data–parallel `HPF` diversi dal Master). Le variabili `grp_total` e `grp_skie` (linea 8), sono i gruppi MPI corrispondenti rispettivamente a `MPI_COMM_WORLD` e `SKIE_COMM_WORLD`.

Dopo l'inizializzazione della macchina virtuale MPI (linea 10) vengono eseguite le seguenti operazioni:

- creazione di `grp_total` (linea 12);
- creazione di `grp_skie` (linea 13), utilizzando `virtual_ranks`;
- creazione di `SKIE_COMM_WORLD` (linea 14);
- inizializzazione di `me`.

Lo *switch* alla linea 18 assegna i colori ai processori del programma SkIE. In particolare si noti come i processori di rank globale 1, 2, 3 e 4 (cioè quelli coinvolti nell'esecuzione del task data–parallel) assumono lo stesso colore. Infine attraverso una chiamata a `MPI_Comm_split` il comunicatore globale viene splittato in tanti sottocomunicatori locali `new_comm`. Terminata la fase dello splitting i processori eseguono lo *switch* (linea 44) che li porterà ad eseguire i vari task del programma SkIE. Questa fase è del tutto simile a quella in cui non ci sono data–parallel `HPF` con la differenza che qui viene eseguita sul colore e non sul rank del processori.

2.5 Struttura del template di uno stadio data-parallel HPF

L'idea iniziale per l'integrazione di HPF in SkIE era quella di mantenere lo stile di progettazione dei template dei normali sequenziali degli altri linguaggi ospiti. Vale a dire un template C che si occupa di gestire le comunicazioni con gli altri task (attraverso i canali degli stream) del programma e che chiama la procedura associata al task sequenziale. Questo tipo di soluzione ha però delle limitazioni che la rendono inaccetabile per i nostri scopi. Lo scopo di HPF in SkIE–CL è quello di permettere l'esecuzione di applicazioni data–parallel che lavorano su grosse mole di dati. Per realizzare questo tipo di applicazioni è necessario disporre di un meccanismo che permetta ai processi data–parallel di istanziare, all'inizio dell'elaborazione, array (distribuiti) che rimangano in memoria per tutta l'elaborazione del programma. Per i sequenziali C standard questo è ottenibile definendo gli array come (in generale vale per qualunque variabile) `static` (in FORTRAN esiste una corrispondente direttiva SAVE). Questo tipo di direttiva non è però utilizzabile con i compilatori HPF attuali, vale a dire che un array distribuito non può essere definito di tipo SAVE. Se i dati sono "statici" (e distribuiti) si ha il vantaggio infatti di evitare la redistribuzione ad ogni attivazione del task HPF, ottendo forti miglioramenti in termini di prestazioni.

Per ovviare a questo problema è stato scelto di realizzare il template che gestisce i task data–parallel in HPF. Il template contiene il codice per la gestione dello stream e il codice del task HPF. In questo modo, a differenza di quanto avviene nel caso dei sequenziali scritti in altri linguaggi, tutte le variabili locali definite all'interno dei task HPF rimangono in memoria per tutta la durata del programma SkIE. SkIE–CL deve essere però messo in grado di distinguire la parte dichiarativa delle variabili (il preambolo) e la parte puramente algoritmica di ogni data–parallel HPF. Per non appesantire la sintassi di SkIE è stato scelto di utilizzare un semplice commento come separatore (!skie$) che non ha nessuna influenza sul codice HPF ma assume solo significato durante la fase di compilazione da parte di SkIE–CL.

Il codice relativo alla gestione dello stream è stato incapsulato in diverse funzioni C che vengono chiamate dal template HPF. La Figura 2.3 mostra le differenze di funzionamento fra i template standard e quelli per i data–parallel HPF. Le procedure di supporto C che gestiscono la comunicazione con il mondo "esterno" di un data–parallel HPF, sono agganciate al template HPF secondo lo schema dello standard HPF EXTRINSIC SERIAL. La semantica delle procedure EXTRINSIC SERIAL invocate nel template data-parallel per effettuare le comunicazioni inter-task assicura che

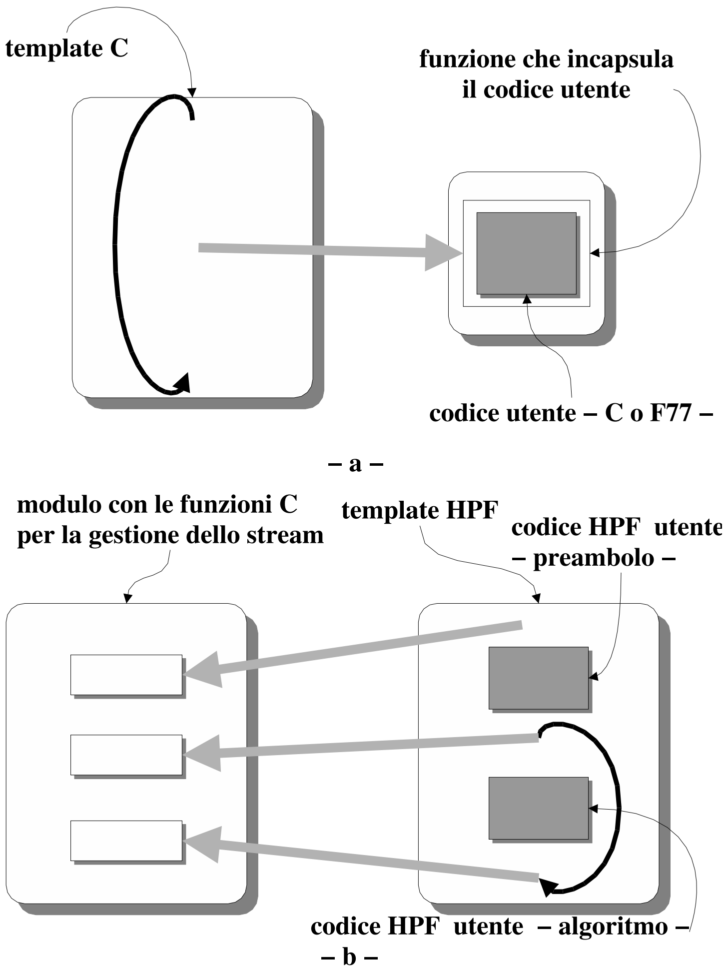

Figura 2.3: Struttura dei template; a) caso di sequenziale C o F77, b) caso di data–parallel `HPF`.

- solo un processore tra quelli allocati per l'esecuzione del task `HPF`, ovvero il Master, esegue le subroutine `EXTRINSIC SERIAL` per effettuare le comunicazioni. Tutti gli altri processori si sincronizzano con il Master, attendendo il ritorno dalla subroutine;

- una nuova "copia" di ciascun parametro della procedura invocata e' allocato sul processore Master;

- all'ingresso/uscita dalla subrotine le "copie" dei parametri vengono aggiornate con i valori delle opportune variabili. Il numero degli aggiornamenti può essere ridotto specificando se i vari parametri sono in input, in output o in input–output (`INTENT(IN,OUT,INOUT)`);

- se un parametro da passare alla subroutine è *replicato* su tutti i processori coinvolti (come succede per le variabili scalari in `HPF`), ed è esclusivamente in input, allora la copia da effettuare all'*ingresso* della subroutine stessa coinvolgerà soltanto il processore Master. Al contrario, se quel parametro è anche in output, la copia da effettuare all'*uscita* della subroutine richiede un broadcast da parte del processore Master per aggiornare le varie repliche della variabile coinvolta;

- se un parametro da passare alla subroutine è *distribuito*, sia nel caso che esso sia in input che in output sono necessarie comunicazioni per collezionare/distribuire porzioni di struttura dati allocate sui vari processori rispetto alla copia locale presente sul processore Master.

Consideriamo sempre il nostro caso di studio. Il template del task `HPF` corrispondente allo stadio intermedio del pipeline è il seguente.

```
      subroutine proctemp_1()

      INCLUDE 'hpf_constant'
      INCLUDE 'SupportHPF_1.h'

      DOUBLE PRECISION  x(64, 64)
      DOUBLE PRECISION  y(64, 64)
      LOGICAL StreamEnd = .false.
      integer temp
      integer ntask


  integer i, j
  !hpf$ distribute (BLOCK,BLOCK) :: x, y

      call hpftemp_inits_1()
      call hpftemp_first_recs_1()
```

```
18          do while (.not. StreamEnd)
19                  call hpftemp_init_acttag_1()
20                  call hpftemp_get_pbin_ntask_1(ntask)
21                  if (ntask >  0) then
22                          call hpftemp_get_input_1(x)
23
24      forall (i=2:N-1, j=2:N-1)
25          y(i,j) = (
26         &       x(i,j) + x(i-1,j) + x(i+1,j)
27         &     + x(i,j-1) + x(i-1,j-1) + x(i+1,j-1)
28         &     + x(i,j+1) + x(i-1,j+1) + x(i+1,j+1)
29         &     )/9
30      end forall
31
32                          call hpftemp_set_output_1(y)
33                          call hpftemp_set_out1_1()
34                  else
35                          call hpftemp_set_out2_1()
36                  end if
37                  call hpftemp_tail_loop_1()
38                  call hpftemp_test_loop_1(temp)
39                  if (temp .eq. 1) then
40                          StreamEnd = .false.
41                  else
42                          StreamEnd = .true.
43                  end if
44          end do
45          call hpftemp_tail_1()
46          end
```

Il codice del template è racchiuso in una subroutine `HPF` (`proctemp_1`) che è chiamata dalla funzione `main` del programma SkIE. Dopo alcune inclusioni e dichiarazioni di default (linee 3–10) inizia la parte di codice utente corrispondente al preambolo (linee 13–14). Le linee 16–22 e 32–46 realizzano la gestione dello stream chiamando, come detto, le procedure `C` di supporto definite in un modulo esterno. Tali procedure essendo, come detto, di tipo `EXTRINSIC SERIAL` sono di fatto eseguite solo dal processore Master. Il codice algoritmico dell'utente è contenuto nelle linee 24–30. Il codice delle subroutine per la gestione dello stream è costituito in larga parte da quello di un normale template per la gestione di un sequenziale scritto in un altro linguaggio ospite. Particolare attenzione deve essere rivolta alle chiamate di procedura `hpftemp_get_input_1` (linea 22) e `hpftemp_set_output_1` (linea 32). La prima restituisce la matrice `x` ricevuta via stream e la seconda spedisce la matrice `y` attraverso il canale di comunicazione. In generale la procedura `hpftemp_get_input_`i, associata al template `ProcTemp_`i`.hpf`, richiede tanti parametri (di tipo `INTENT (OUT)`) quanti sono i dati in input al task HPF. Analogamente la procedura `hpftemp_set_output_`i richiede tanti parametri (di tipo `INTENT (IN)`) quanti sono i dati in output al task HPF.

Il funzionamento del template HPF può essere compreso meglio se si osserva il codice generato a partire da esso con il traduttore `fadapt` di gmd-hpf. Qui di seguito è mostrato il risultato della traduzione del sorgente di `ProcTemp_1` dalle linee 22–32.

```
subroutine PROCTEMP_1 ()

        .
        .
        .

  call DALIB_array_make_dsp (TMP1_DSP,2,8,1,0,'TMP1')
  call DALIB_array_define (TMP1_DSP,1,64,1,64)
  call DALIB_array_allocate (TMP1_DSP,TMP1)
  call DALIB_array_access (TMP1_DSP,TMP1,TMP1_ZERO,TMP1_DIM1,TMP1_DIM2)
  if (DALIB_pid() .eq. 1) then
    call DALIB_push_host_context ()
    call HPFTEMP_GET_INPUT_1 (TMP1(TMP1_ZERO+TMP1_DIM1+1),TMP1_DSP)
    call DALIB_pop_host_context ()
end if
  call DALIB_broadcast (TMP1(TMP1_ZERO+TMP1_DIM1+1),64*64*8,1)
  call DALIB_array_lslice (X_DSP,2,1,64,X_START2,X_STOP2)
  do I2=X_START2,X_STOP2
    call DALIB_array_lslice (X_DSP,1,1,64,X_START1,X_STOP1)
    do I1=X_START1,X_STOP1
      X(X_ZERO+I2*X_DIM1+I1) = TMP1(TMP1_ZERO+I2*TMP1_DIM1+I1)
    end do
  end do
  call DALIB_array_deallocate (TMP1_DSP)
  call DALIB_array_free (TMP1_DSP,TMP1)

  call DALIB_section_create (ISEC_DSP1,X_DSP,1,
 $  2,N-1,1,1,2,N-1,1)
  call DALIB_overlap_update (ISEC_DSP1,1,1,1,1)
  call DALIB_section_free (ISEC_DSP1)
  call DALIB_array_lslice (Y_DSP,1,2,N-1,Y_START1,Y_STOP1)
  do I=Y_START1,Y_STOP1
    call DALIB_array_lslice (Y_DSP,2,2,N-1,Y_START2,Y_STOP2)
    do J=Y_START2,Y_STOP2
      Y(Y_ZERO+J*Y_DIM1+I) = (X(X_ZERO+J*X_DIM1+I)+X(X_ZERO+
 &J*X_DIM1+(I-1))+X(X_ZERO+J*X_DIM1+(I+1))+X(X_ZERO+(J-1)*X_DIM1+I)+
 &X(X_ZERO+(J-1)*X_DIM1+(I-1))+X(X_ZERO+(J-1)*X_DIM1+(I+1))+X(X_ZERO
 &+(J+1)*X_DIM1+I)+X(X_ZERO+(J+1)*X_DIM1+(I-1))+X(X_ZERO+(J+1)*X_DIM
 &1+(I+1)))/DBLE(9)
    end do
  end do

```

```
43
44      call DALIB_array_make_dsp (TMP3_DSP,2,8,1,0,'TMP3')
45      call DALIB_array_define (TMP3_DSP,1,64,1,64)
46      call DALIB_array_allocate (TMP3_DSP,TMP3)
47      call DALIB_array_access (TMP3_DSP,TMP3,TMP3_ZERO,TMP3_DIM1,TMP3_D
48     &IM2)
49      call DALIB_assign (TMP3_DSP,Y_DSP)
50      if (DALIB_pid() .eq. 1) then
51        call DALIB_push_host_context ()
52        call HPFTEMP_SET_OUTPUT_1 (TMP3(TMP3_ZERO+TMP3_DIM1+1),TMP3_DSP)
53        call DALIB_pop_host_context ()
54      end if
55      call DALIB_array_deallocate (TMP3_DSP)
56      call DALIB_array_free (TMP3_DSP,TMP3)
57
58                    .
59                    .
60                    .
61
62    end
63
```

Prima di chiamare la subroutine `hpftemp_get_input_1` viene creato l'array di appoggio `TMP1` (linee 7–10). Dopodiché il processore Master (la discriminazione viene ottenuta attraverso la lettura del rank del processore corrente con la `DALIB_pid`) chiama la `hpftemp_get_input_1` passandogli come parametro il puntatore a `TMP1`. Al termine della chiamata il contenuto della variabile `TMP1` sarà quello di `x` prelevato dallo stream in input. Il processore Master esegue poi il *broadcast* di `TMP1` a tutti gli altri processori (linea 16). Di seguito ogni processore copia la parte di `TMP1` di propria competenza nella sezione dell'array `x` (linee 17–23). La variabile `TMP1` viene poi deallocata (linea 24–25). Le linee 27–41 sono la traduzione in `F77` del `for all` della parte algoritmica del task `HPF`. Una volta prodotto il risultato dell'elaborazione sull'array `y`, in modo del tutto speculare al caso della ricezione di `x` viene spedito `y` sul canale dello stream attraverso la chiamata a `hpftemp_set_output_1`: dopo aver creato un nuovo array temporaneo `TMP3` (linee 44–48), viene chiamata la `DALIB_assign` che raccoglie l'array distribuito nell'array `TMP3` del processore Master (linea 49). Alla subroutine `hpftemp_set_output_1` viene passato il puntatore a `TMP3`. Infine `TMP3` viene deallocato (linee 55-56).

Le subroutine `hpftemp_get_input_1` e `hpftemp_set_output_1` non provvedono alla ricezione/spedizione dei dati ma eseguono solo una copia da/sui dati sui buffer utilizzati per la comunicazione vera e propria. In questa fase il compilatore genera dei cicli di di copia (se si tratta di array) correttamente in modo da gestire la trasposizione fra gli array `C` e `HPF`, e viceversa.

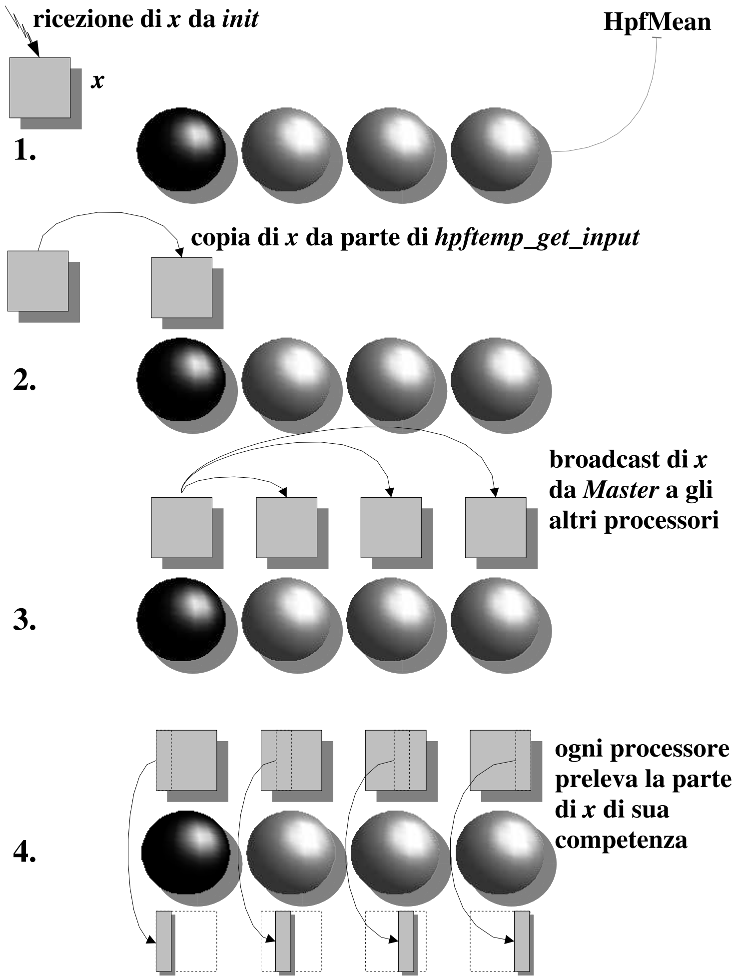

Figura 2.4: Fasi della ricezione dell'array `x` del caso di studio HpfMean.

Capitolo 3

Un esempio di applicazione

In questo capitolo viene mostrato un esempio di una applicazione in SkIE−CL e HPF. Di seguito è riportato il codice SkIE di un pipeline a tre stadi che esegue la trasformata di FFT tridimensionale.

```
const int SIZE = 64;
const int LOG = 6;

pipe main in() out()

  matrix_generator   in() out(stream of double
                                xr[SIZE][SIZE][SIZE],
                              stream of double
                                xi[SIZE][SIZE][SIZE])

  fft3d    in(xr,xi) out(double
                          yr[SIZE][SIZE][SIZE],
                         double
                          yi[SIZE][SIZE][SIZE])

  matrix_printer   in(yr, yi) out()

end pipe


matrix_generator in() out(stream of double xr[SIZE][SIZE][SIZE],
                          stream of double xi[SIZE][SIZE][SIZE])
$c{
        matrixmaker(xr,xi);
        SKIE_out();
}c$
src(funzioni.c)
end

matrix_printer  in(double yr[SIZE][SIZE][SIZE],
```

```
                   double yi[SIZE][SIZE][SIZE]) out()
$c{
        FILE *filereal, *fileimm;

        filereal = fopen("fft3d.out.real","w");
        fileimm  = fopen("fft3d.out.imm","w");

        output_su_file_matrice(filereal, yr);
        output_su_file_matrice(fileimm, yi);

        fclose(filereal);
        fclose(fileimm);
}c$
src(funzioni.c)
inc(stdio.h)
end

pragma parallelism degree 8 in fft3d;

fft3d    in(double xr[SIZE][SIZE][SIZE], double xi[SIZE][SIZE][SIZE])
         out(double yr[SIZE][SIZE][SIZE], double yi[SIZE][SIZE][SIZE])
$hpf{
        INTERFACE
        PURE SUBROUTINE fft_slice(a,ai,SIZE,isgn)
          INTEGER,INTENT(IN)                 :: SIZE,isgn
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: a
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: ai
        END SUBROUTINE fft_slice
        END INTERFACE

        integer i1, i2, i3

        integer tickstart, tickstop, tickrate
        real time, mflops

!hpf$   distribute xr(BLOCK,*,*)
!hpf$   distribute xi(BLOCK,*,*)

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


        call system_clock (tickstart, tickrate)


!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)
```

```

         end do
        end do

        forall (i1=1:SIZE)
                xr(i1,:,:) = transpose(xr(i1,:,:))
                xi(i1,:,:) = transpose(xi(i1,:,:))
        end forall


!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)

         end do
        end do

        forall (i2=1:SIZE)
                xr(:,i2,:) = transpose(xr(:,i2,:))
                xi(:,i2,:) = transpose(xi(:,i2,:))
        end forall


!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)

         end do
        end do

        call system_clock (tickstop)

        yr = xr
        yi = xi

        time   = float (tickstop - tickstart) / float(tickrate)
        print *, 'time   = ', time, ' seconds'

}hpf$
src(fft_slice.hpf, ipow.hpf, ilog2.hpf)
end
```

Il primo stadio del pipeline, `matrix_generator`, crea, attraverso la funzione `matrixmaker`, due array tridimensionali, `xi` e `xr`, di dimensioni $SIZE \times SIZE \times SIZE$ dove $SIZE$ è un costante definita all'inizio programma che

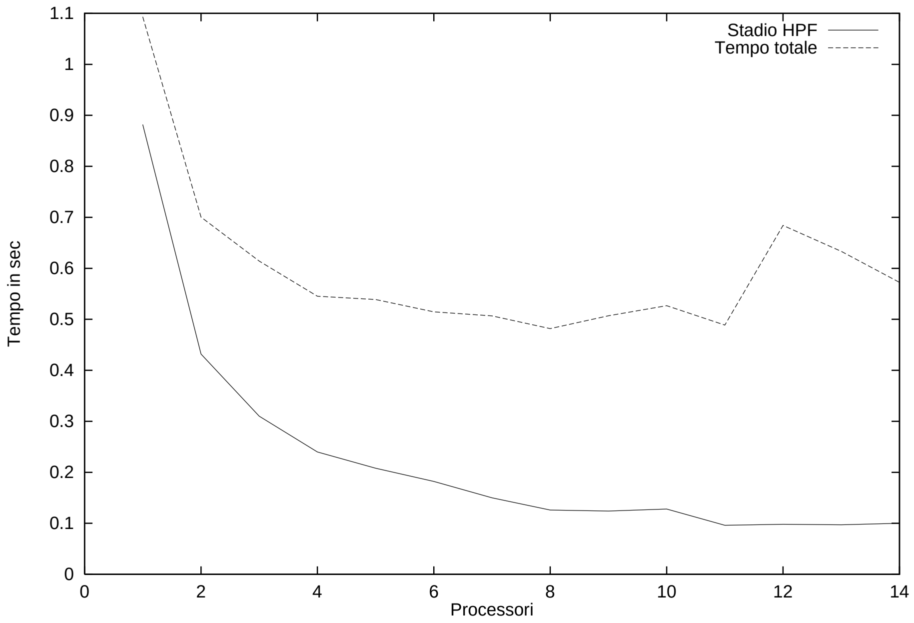

Figura 3.1: Tempi di esecuzione del programma fft3d, dimensioni del problema $32 \times 32 \times 32$.

specifica le dimensioni del problema. `xi` e `xr`, che rappresentano rispettivamente la parte immaginaria e reale del cubo da trasformare con la FFT3D, vegono spediti allo stadio `HPF`, chiamato `fft3d`, che esegue la appunto la FFT3D. L'algoritmo fa uso della trasformata monodimensionale (la subrotine fft_slice) applicandola tre volte al cubo di ingresso secondo le tre direzione spaziali x, y e z. Gli array `xi` e `xr` sono distribuiti inizialmente in blocchi secondo la direzione x. Il primo ciclo `do` esegue la trasformata secondo la direzione z. Ogni processore esegue cioè la trasformata su ogni vettore dell'asse z che compone il cubo senza coinvolgere operazioni di comunicazione, in forza della distribuzione utilizzata. Il ciclo `forall` che segue esegue la trasposizione degli array di ingresso secondo i piani x, z del cubo su cui è stata eseguita la prima trasformazione. Il secondo ciclo `do`, identico al primo, esegue di nuovo la trasformata lineare secondo l'asse z, ma poiché il cubo è stato trasposto secondo i piani y, z questa seconda trasformazione avviene in realtà secondo l'asse delle y del cubo di partenza. In modo analogo procedendo ad un seconda trasposizione (secondo il piano x, z in questo caso) ed un terzo ciclo di trasformazione si ottiene l'ultima trasformazione secondo la direzione x. Il cubo trasformato viene infine passato al terzo stadio del pipeline, `matrix_printer` che esegue il salvataggio del cubo trasformato su due distinti file uno per la parte reale e uno per la parte immaginaria.

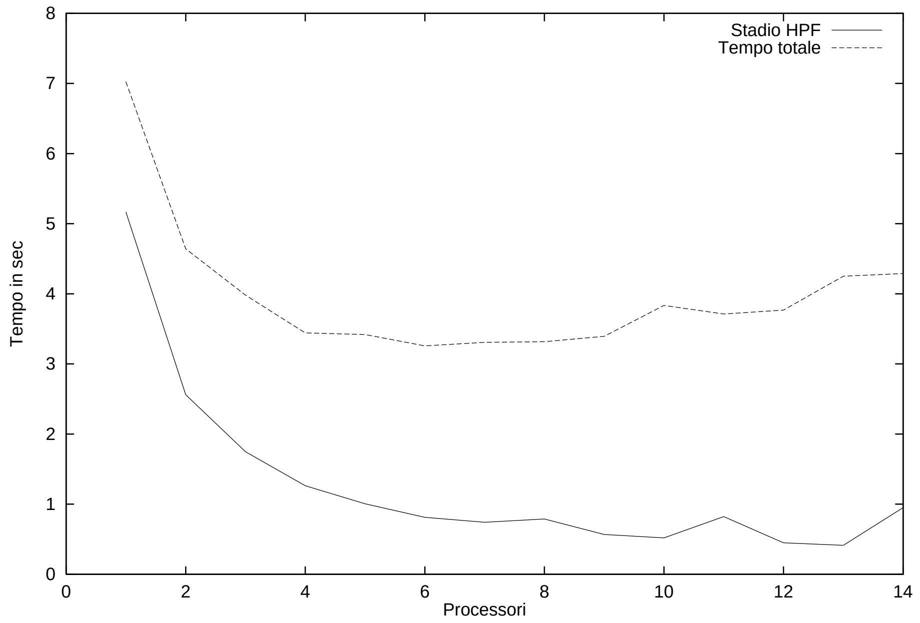

Figura 3.2: Tempi di esecuzione del programma fft3d, dimensioni del problema $64 \times 64 \times 64$.

Nelle Figure 3.1 e 3.2 sono mostrati tempi di calcolo dell'applicazione eseguita sulla macchina QM1, rispettivamente per i casi $32 \times 32 \times 32$ e $64 \times 64 \times 64$, al variare del numero di nodi utilizzati per lo stadio HPF. I tempi di esecuzione sono stati depurati dai tempi di salvataggio dei dati su disco. Ogni grafico mostra sia il tempo di esecuzione totale del programma sia il tempo di esecuzione del solo stadio HPF. La differenza fra il tempo totale del programma SkIE e il task HPF è composta in piccola parte dal tempo di esecuzione dello stadio `matrix_generator`[1] e in gran parte dai tempi di comunicazione fra i vari stadi della pipeline e le ridistribuzioni dovute al template HPF di cui si è parlato nel capitolo precedente. Questo è confermato sia dal fatto che il tempo totale tende ad aumentare al di là di un certo numero di processori (soprattutto nel caso $64 \times 64 \times 64$) sia dall'osservazione che il fenomeno tende ad essere meno importante con dimensioni inferiori del problema.

La perdita di prestazione dovuto all'uso dell'ambiente SkIE–CL rispetto ad una soluzione completamente sviluppata in HPF è comuque limitata. La Figura 3.3 mostra il confronto fra la stessa applicazione fft3d completamente scritta in HPF e quella scritta in SkIE+HPF. Considerato che nella versione scritta totalmente in HPF viene risparmiato il tempo dovuto alla comunica-

[1] Da misurazioni effettuate risulta infatti che il tempo dello stadio `matrix_generator` è di circa 0.12 sec per il caso $32 \times 32 \times 32$ e 1 sec per il caso $64 \times 64 \times 64$.

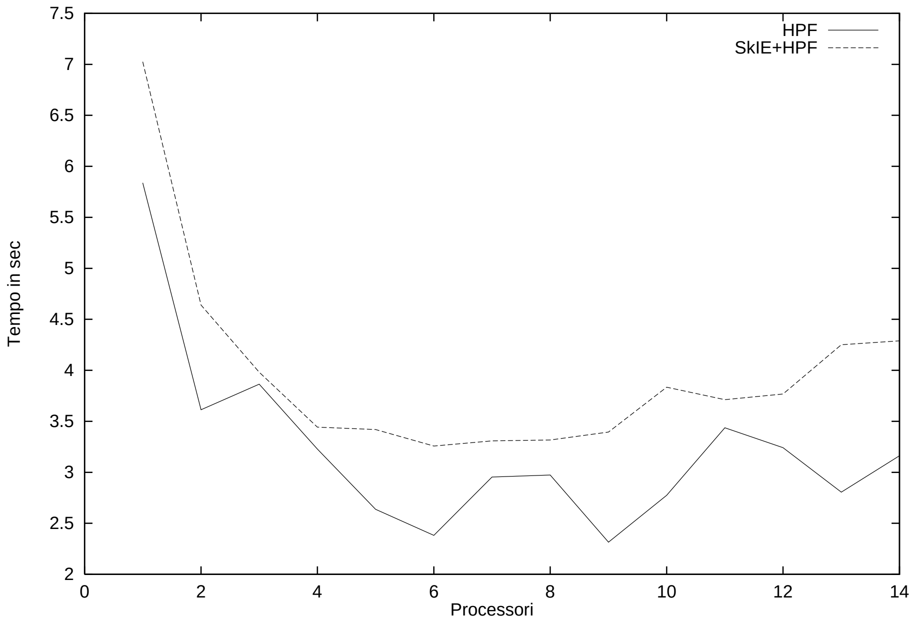

Figura 3.3: Confronto dei tempi di esecuzione del programma fft3d sviluppato in SkIE+HPF puro HPF, dimensioni del problema $64 \times 64 \times 64$.

zione fra lo stadio che crea i dati e lo stadio HPF che li elabora, le prestazioni del template SkIE scritto in HPF risultano soddisfacenti.

Sfruttare il task parallelism I vantaggi dell'ambiente di sviluppo SkIE rispetto all'uso di HPF puro può essere osservato nel caso di stream di lunghezza maggiore di uno. Come si vedrà le prestazioni del programma SkIE–CL saranno infatti superiori in quanto il tempo di calcolo dello stadio di generazione del cubo i-esimo da elaborare si sovrappone con il tempo necessario alla trasformazione del cubo $(i-1)$-esimo. Per sfruttare meglio il vantaggio del pipeling è conveniente spezzare lo stadio HPF nelle sue componenti di elaborazione cioè le trasformate secondo le tre direzioni spaziali, come qui di seguito riportato, è illustrato in Figura 3.4

```
const int SIZE = 32;
const int LOG = 5;

pipe main in() out()
  matrix_generator   in()       out(stream of double
                                       xr1[SIZE][SIZE][SIZE],
                                    stream of double
                                       xi1[SIZE][SIZE][SIZE])

```

```
  fft3d1    in(xr1,xi1)      out(double
                                   tr2[SIZE][SIZE][SIZE],
                                 double
                                   ti2[SIZE][SIZE][SIZE])

  fft3d2    in(tr2,ti2)      out(double
                                   tr3[SIZE][SIZE][SIZE],
                                 double
                                   ti3[SIZE][SIZE][SIZE])

  fft3d3    in(tr3,ti3)      out(double
                                   yr[SIZE][SIZE][SIZE],
                                 double
                                   yi[SIZE][SIZE][SIZE])

  matrix_printer   in(yr, yi) out()

end pipe


matrix_generator in() out(stream of double xr1[SIZE][SIZE][SIZE],
                          stream of double xi1[SIZE][SIZE][SIZE])
$c{
        int i;

        for(i=0;i<100;i++)
        {
                matrixmaker(xr1,xi1);
                SKIE_out();
        }
}c$
src(funzioni.c)
end

matrix_printer  in(double yr[SIZE][SIZE][SIZE],
                   double yi[SIZE][SIZE][SIZE]) out()
$c{

        FILE *filereal, *fileimm;

        filereal = fopen("fft3d.out.real","w");
        fileimm  = fopen("fft3d.out.imm","w");

        output_su_file_matrice(filereal, yr);
        output_su_file_matrice(fileimm, yi);

        fclose(filereal);
        fclose(fileimm);

```

```
}c$
inc(stdio.h)
end

pragma parallelism degree 5 in fft3d1;
pragma parallelism degree 5 in fft3d2;
pragma parallelism degree 4 in fft3d3;

fft3d1    in(double xr1[SIZE][SIZE][SIZE], double xi1[SIZE][SIZE][SIZE])
          out(double yr2[SIZE][SIZE][SIZE], double yi2[SIZE][SIZE][SIZE])
$hpf{
        INTERFACE
        PURE SUBROUTINE fft_slice(a,ai,SIZE,isgn)
          INTEGER,INTENT(IN)                :: SIZE,isgn
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: a
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: ai
        END SUBROUTINE fft_slice
        END INTERFACE

        integer i1, i2, i3

!hpf$   distribute xr1(BLOCK,*,*)
!hpf$   distribute xi1(BLOCK,*,*)

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr1(i1,i2,:),xi1(i1,i2,:),SIZE,1)

         end do
        end do

        forall (i1=1:SIZE)
                xr1(i1,:,:) = transpose(xr1(i1,:,:))
                xi1(i1,:,:) = transpose(xi1(i1,:,:))
        end forall

        yr2 = xr1
        yi2 = xi1


}hpf$
src(fft_slice.hpf, ipow.hpf, ilog2.hpf)
end

```

```
fft3d2    in(double xr2[SIZE][SIZE][SIZE], double xi2[SIZE][SIZE][SIZE])
          out(double yr3[SIZE][SIZE][SIZE], double yi3[SIZE][SIZE][SIZE])
$hpf{
        INTERFACE
        PURE SUBROUTINE fft_slice(a,ai,SIZE,isgn)
          INTEGER,INTENT(IN)                 :: SIZE,isgn
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: a
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: ai
        END SUBROUTINE fft_slice
        END INTERFACE

        integer i1, i2, i3


!hpf$   distribute xr2(BLOCK,*,*)
!hpf$   distribute xi2(BLOCK,*,*)

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr2(i1,i2,:),xi2(i1,i2,:),SIZE,1)

         end do
        end do

        forall (i2=1:SIZE)
                xr2(:,i2,:) = transpose(xr2(:,i2,:))
                xi2(:,i2,:) = transpose(xi2(:,i2,:))
        end forall

        yr3 = xr2
        yi3 = xi2


}hpf$
src(fft_slice.hpf, ipow.hpf, ilog2.hpf)
end



fft3d3    in(double xr3[SIZE][SIZE][SIZE], double xi3[SIZE][SIZE][SIZE])
          out(double yr[SIZE][SIZE][SIZE], double yi[SIZE][SIZE][SIZE])
$hpf{
        INTERFACE
        PURE SUBROUTINE fft_slice(a,ai,SIZE,isgn)
```

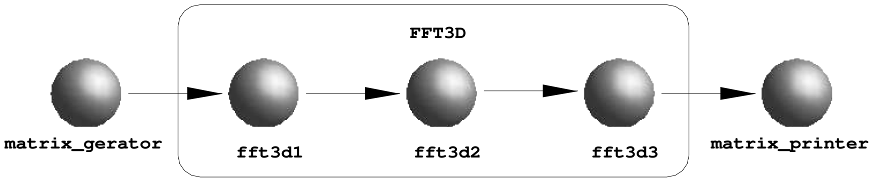

Figura 3.4: Schema del programma FFT3D con pipeline.

```
          INTEGER,INTENT(IN)                :: SIZE,isgn
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: a
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: ai
        END SUBROUTINE fft_slice
        END INTERFACE

        integer i1, i2, i3

!hpf$   distribute xr3(BLOCK,*,*)
!hpf$   distribute xi3(BLOCK,*,*)

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

              call fft_slice(xr3(i1,i2,:),xi3(i1,i2,:),SIZE,1)

         end do
        end do

        yr = xr3
        yi = xi3


}hpf$
src(fft_slice.hpf, ipow.hpf, ilog2.hpf)
end
```

Lo stadio `matrix_generator` genera uno stream di 100 cubi da trasformare (in questo caso i cubi generati per sempilicità sono tutti uguali). I tre stadi successivi eseguono le tre trasformate. In particolare, `fft3d1` esegue la trasformata secondo la direzione z e la prima serie di trasposizioni, `fft3d2` esegue la trasformata secondo la direzione y e la seconda serie di trasposizioni e infine `fft3d3` la trasformata secondo la direzione x.

Un'altra tecnica di ottimizzazione è quella di strasformare lo stadio `fft3d` del programma SkIE–CL iniziale in uno stadio di tipo farm. Segue il programma così modificato, il cui schema è illustrato in Figura 3.5

```
const int SIZE = 32;
const int LOG = 5;

pipe main in() out()

  matrix_generator   in()       out(stream of double
                                      xr[SIZE][SIZE][SIZE],
                                    stream of double
                                      xi[SIZE][SIZE][SIZE])

  farmhpf    in(xr,xi)      out(double
                                 yr[SIZE][SIZE][SIZE],
                                double
                                 yi[SIZE][SIZE][SIZE])

  matrix_printer   in(yr, yi) out()

end pipe


matrix_generator in() out(stream of double xr[SIZE][SIZE][SIZE],
                          stream of double xi[SIZE][SIZE][SIZE])
$c{
        int i;

        for(i=0;i<100;i++)
        {
                matrixmaker(xr,xi);
                SKIE_out();
        }
}c$
src(funzioni.c)
end

matrix_printer  in(double yr[SIZE][SIZE][SIZE],
                double yi[SIZE][SIZE][SIZE]) out()
$c{
        FILE *filereal, *fileimm;

        filereal = fopen("fft3d.out.real","w");
        fileimm  = fopen("fft3d.out.imm","w");

        output_su_file_matrice(filereal, yr);
        output_su_file_matrice(fileimm, yi);

        fclose(filereal);
```

```
        fclose(fileimm);
}c$
inc(stdio.h)
end

pragma parallelism degree 6 in farmhpf;

farm farmhpf in(double xr[SIZE][SIZE][SIZE], double xi[SIZE][SIZE][SIZE])
             out(double yr[SIZE][SIZE][SIZE], double yi[SIZE][SIZE][SIZE])

 fft3d in (xr, xi) out (yr, yi)

end farm

pragma parallelism degree 2 in fft3d;

fft3d    in(double xr[SIZE][SIZE][SIZE], double xi[SIZE][SIZE][SIZE])
         out(double yr[SIZE][SIZE][SIZE], double yi[SIZE][SIZE][SIZE])
$hpf{
        INTERFACE
        PURE SUBROUTINE fft_slice(a,ai,SIZE,isgn)
          INTEGER,INTENT(IN)                 :: SIZE,isgn
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: a
          double precision,DIMENSION(SIZE),INTENT(INOUT) :: ai
        END SUBROUTINE fft_slice
        END INTERFACE

        integer i1, i2, i3

!hpf$   distribute xr(BLOCK,*,*)
!hpf$   distribute xi(BLOCK,*,*)

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)

         end do
        end do

        forall (i1=1:SIZE)
                xr(i1,:,:) = transpose(xr(i1,:,:))
                xi(i1,:,:) = transpose(xi(i1,:,:))
        end forall

```

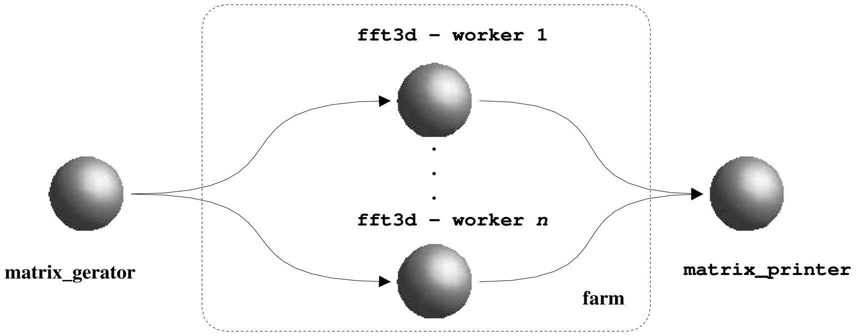

Figura 3.5: Schema del programma FFT3D con farm.

```

!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)

         end do
        end do

        forall (i2=1:SIZE)
                xr(:,i2,:) = transpose(xr(:,i2,:))
                xi(:,i2,:) = transpose(xi(:,i2,:))
        end forall


!hpf$   independent
        do i1=1,SIZE
         do i2=1,SIZE

                call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)

         end do
        end do

        yr = xr
        yi = xi

}hpf$
src(fft_slice.hpf, ipow.hpf, ilog2.hpf)
end
```

Lo stadio `farmhpf` è un farm di costituito da un certo numero (in questo caso 6) worker `HPF` ognuno dei quali con un grado di parallelismo 2.

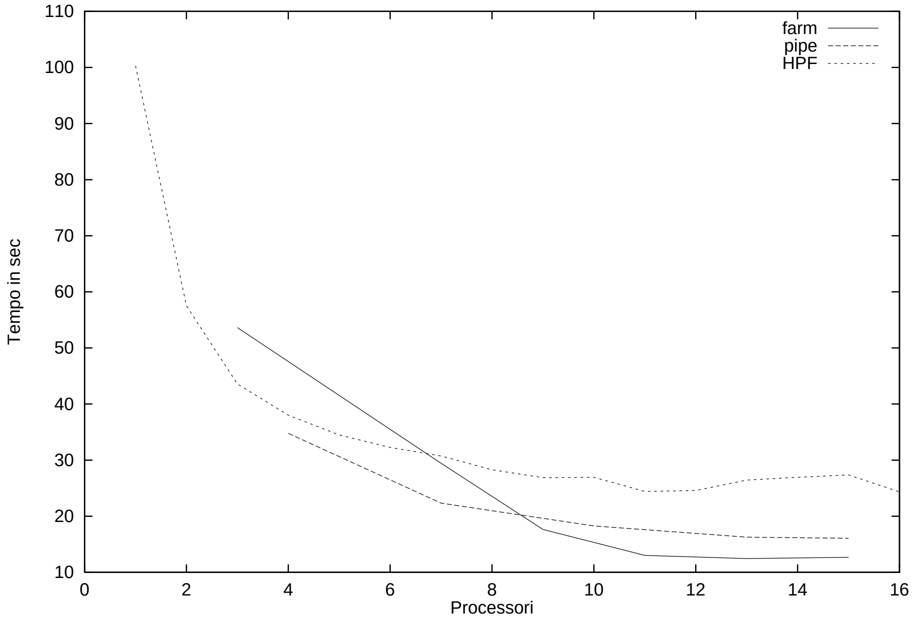

Figura 3.6: Confronto fra tempi di esecuzione del programma fft3d realizzato con `HPF` puro SkIE–CL con pipeline e SkIE–CL con farm. Stream di 100 dati e dimensioni del problema $32 \times 32 \times 32$.

La Figure 3.6 mostra il confronto fra i tempi di esecuzione delle due soluzioni proposte rispetto a quello dell'implementazione in `HPF` puro. Poiché anche in questo caso non è stato considerato il tempo dello stadio finale `matrix_printer`, nel resto della trattazione il processore impiegato per la sua esecuzione verrà trascurato. Lo stream considerato è di lunghezza 100. Il numero di processori nel caso con pipeling è dato da numero di processori per i tre stadi `HPF` più uno per lo stadio `matrix_generator`. Se indichiamo con p_1, p_2 e p_3 il mumero di processori rispettivamente per gli stadi `fft3d1`, `fft3d2` e `fft3d3` del pipeline, le configurazioni che sono state prese in considerazione sono le seguenti:

- $p_1 = 1$, $p_2 = 1$, $p_3 = 1$ e un processore per `matrix_generator`, numero totale di processori 4;

- $p_1 = 2$, $p_2 = 2$, $p_3 = 2$ e un processore per `matrix_generator`, numero totale di processori 7;

- $p_1 = 3$, $p_2 = 3$, $p_3 = 3$ e un processore per `matrix_generator`, numero totale di processori 10;

- $p_1 = 4$, $p_2 = 4$, $p_3 = 4$ e un processore per `matrix_generator`, numero totale di processori 13;

- $p_1 = 5$, $p_2 = 5$, $p_3 = 4$ e un processore per `matrix_generator`, numero totale di processori 15.

Nella soluzione con farm il grado di parallelismo dei worker (vale a dire dei task data–parallel `HPF`) è stato mantenuto fisso uguale a 2; mentre è stato fatto variare il loro numero [2]. Sia p_w il numero di worker e p_{hpf} il grado di parallelismo dei worker `HPF` le configurazioni considerate in Figura 3.6 sono allora le seguenti:

- $p_w = 1$, $p_{hpf} = 2$ e un processore per `matrix_generator`, numero totale di processori 3;

- $p_w = 2$, $p_{hpf} = 2$, un processore per l'emettitore, un processore per il collettore e un processore per `matrix_generator`, numero totale di processori 7;

- $p_w = 3$, $p_{hpf} = 2$, un processore per l'emettitore, un processore per il collettore e un processore per `matrix_generator`, numero totale di processori 9;

- $p_w = 4$, $p_{hpf} = 2$, un processore per l'emettitore, un processore per il collettore e un processore per `matrix_generator`, numero totale di processori 11;

- $p_w = 5$, $p_{hpf} = 2$, un processore per l'emettitore, un processore per il collettore e un processore per `matrix_generator`, numero totale di processori 13;

- $p_w = 6$, $p_{hpf} = 2$, un processore per l'emettitore, un processore per il collettore e un processore per `matrix_generator`, numero totale di processori 15.

Come si può notare la soluzione in SkIE–CL scala decisamente meglio e in particolare la soluzione con il costrutto farm per cui si ha un tempo totale di elaborazione di 12.4 sec contro i 16.3 del caso con pipeline e 27.4 del caso `HPF` puro.

[2] Da prove sperimentali effettuate risulta che tale scelta è ottimale

Bibliografia

[1] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: an heterogeneous HPC environment. Submitted for publication, 1999.

[2] T. Brandes. ADAPTOR Programmer's Guide Version 5.0. Internal Report Adaptor 3, GMD-SCAI, Sankt Augustin, Germany, April 97.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Jan. 1997. Version 2.0.

[5] C.H. Koebel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[6] Marco Vanneschi. PQE2000: HPC Tools for industrial applications. *IEEE Concurrency*, 6(4), October-December 1998.

Appendice A

Modifiche apportate al supporto DALIB

Tutte le modifiche apportate alla libreria DALIB (6.1) sono relative alla versione MPI situata nella directory `$PHOME/dalib/MPI`.

Tutte le occorrenze della costante `MPI_COMM_WORLD` sono state sostituite con la variabile `new_comm` all'interno dei sorgenti della libreria. In questo modo, come spiegato sopra, le procedure DALIB operano all'interno del comunicatore `new_comm` senza interferire con l'ambiente di comunicazione dei template di SkIE–CL. I file che contengono le chiamate MPI da modificare in modo da utilizzare il comunicatore `new_comm` sono `init.m4` e `pt2pt.m4`.

Un'altra importante modifica da apportare alla libreria è quella di eliminare la chiamata alla funzione `eval_arg` all'interno della funzione `dalib_machine_enroll`. Lo scopo di `eval_arg` è quello di inizializzare il supporto DALIB in base ai parametri passati attraverso la riga di comando del programma generato dal compilatore HPF. Queste inizializzazioni si riferiscono ad esempio al numero di processori, test di misura dei tempi di esecuzione, etc. Dato che, nel nostro caso, il programma principale viene generato dal compilatore SkIE–CL le inizializzazioni della funzione `eval_arg` non sono più necessarie. Inoltre devono essere necessariamente rimosse poiché in caso contrario tenterebbero di leggere gli argomenti passati al programma SkIE–CL, il cui formato è chiaramente diverso da quello utilizzato da Adaptor. La chiamata alla funzione `eval_arg` è stata sostituita con una parte delle inizializzazioni che essa opera e che sono comunque necessarie al supporto. La parte di codice che contiene la chiamata ad `eval_arg` è stata modificata come segue:

```
    pcb.p = size;

    if (pcb.i == 1)
```

```
4
5         { /* only first process gets the parameters */
6
7           /*eval_arg (*NP, *NP);*/
8           dalib_set_top1 (*NP);
9
10          pcb.trace_flag = 0;    /* default start without trace */
11          pcb.call_flag = 0;     /* default start without call trace */
12          pcb.redist_flag = 0;   /* default start without redist trace */
13          pcb.comm_flag = 0;     /* default start without comm statistics */
14          pcb.time_flag = 0;     /* default start without time statistics */
15
16        }
17
18
19         strcpy (dalib_program_name, "skie.out.skie");
```

La linea 8 chiama la funzione `dalib_set_top1` necessaria per l'inizializzazione delle informazioni riguardanti la topologia virtuale dei processori. In questo caso si è scelto di utilizzare una semplice topologia unidimensionale. Le linee dalla 10 alla 14 settano i flag del programma in modo da non generare nessuna informazione di traccia e statistiche di esecuzione. La linea 19 setta il nome del programma; questa informazione veniva originariamente estratta dalla riga di comando.

Quando un programma HPF parte vengono eseguite alcune inizializzazioni fra le quali quelle del supporto di comunicazione MPI. In particolare viene chiamata la funzione `mpi_init`. Questa chiamata (eseguita dalla funzione `dalib_machine_enroll` la cui definizione si trova nel file init.m4) deve essere eliminata, poiché è già eseguita dal codice generato da SkIE–CL.

Infine per evitare problemi di linkaggio è necessario eliminare tutte le chiamate alle funzioni `getarg_` e `iargc_` presenti nel file arguments.m4.

Appendice B

Il CDROM allegato

Il CDROM allegato contiene le seguenti directory:

```
Adaptor

DalibModificata

Docs

Esempi

MPI

ProveDiFattibilita

SkieHpf

Sources
```

In particolare:

- La directory `Adaptor` contiene i sorgenti del compilatore `HPF` utilizzato;
- `DalibModificata` contiene i sorgenti della libreria DALIB modificate per supportare SkIE con `HPF`;
- `Docs` contiene questo report e il report precedente entrambi in formato postscript;
- Nella directory `Esempi` sono presenti i sorgenti degli esempi mostrati in questo report. In particolare la subdirectory `Pipe3` contiene i sorgenti del semplice pipeline del Capitolo 2.2. `FFT3D` contiene i sorgenti

della FFT3D nella prima versione con tre stadi. `FFT3D_pipe` contiene i sorgenti della versione con pipelining e `FFT3D_farm` la versione realizzata con il costrutto farm. Infine `FFT3D_hpf` contiene la versione in `HPF` puro. Si esegua il comando `'source nomescript'` (dove `nomescript` è il nome dello script di configurazione di SkIE–CL per l'architettura utilizzata) prima di eseguire la compilazione. Per le tre versioni della FFT3D è necessario modificare lo script di configurazione eliminando lo switch `-free` nella variabile di ambiente `S_HPFFLAGS`, come segue:

```
setenv S_HPFFLAGS '-Wa"-strip 72" -Wf"-fno-second-underscore"'
```

- Nella directory `MPI` è contenuto l'ambiente di sviluppo message–passing MPICH;
- `ProveDiFattibilita` contiene alcune prove preliminari allo sviluppo di SkIE e `HPF`;
- `SkieHpf` è la directory con i binari di SkIE–CL. La subdirectory `bin` contiene gli eseguibili del compilatore SkIE–CL nelle due versioni per le architetture Linux e Sun/QM1; La subdirectory `lib` contiene la libreria DALIB modificata e `scripts` gli scripts di configurazione.
- Infine `Sources` contiene i sorgenti del generatore di codice di SkIE con `HPF`.