Appendice B

Ambienti per la programmazione parallela

La programmazione di applicazioni parallele è un compito molto complesso, in quanto è necessario gestire direttamente molti dettagli implementativi: la sincronizzazione e la comunicazione tra i vari processi che costituiscono il programma, il mapping dei processi ai processori disponibili, ecc ...

Considerando un tipo di parallelismo a granularità di tipo coarse grain (il tipo di parallelismo da considerare nel sistema da realizzare), la progettazione di un'applicazione parallela può essere divisa in due fasi [18]:

1. la definizione delle diverse componenti sequenziali;
2. la definizione delle relazioni esistenti tra questi elementi.

Per semplificare lo sviluppo di applicazioni parallele di questo tipo, negli ultimi anni sono stati sviluppati appositi ambienti visuali. Alcuni di essi vengono esaminati nei prossimi paragrafi.

B.1 Grafi di computazione

Il formalismo del **grafo di computazione** [17] è uno strumento molto utile per la definizione di un'applicazione parallela, ed è utilizzato in molti ambienti: un grafo di computazione è un grafo diretto, in cui i nodi rappresentano componenti sequenziali, mentre gli archi definiscono relazioni tra queste componenti. Una relazione tra due nodi può definire uno scambio di messaggi, o una sincronizzazione (ossia un vincolo di precedenza) tra questi due nodi.
I grafi di computazione:

- permettono lo sviluppo di applicazioni parallele, secondo la metodologia in due fasi prima descritta (la definizione delle componenti sequenziali, e la definizione delle relazioni tra queste componenti): i nodi del grafo di computazione rappresentano le componenti sequenziali, mentre gli archi vengono utilizzati per definire le relazioni;

- consentono di combinare con facilità componenti sequenziali precedentemente definite;

- permettono di astrarre la rappresentazione del programma dalla sua implementazione;

- favoriscono l'utilizzo della metodologia di *programmazione top-down*: durante lo sviluppo dell'applicazione è possibile sostituire un nodo con un grafo, come rappresentato nell'esempio di figura B.1. In questo esempio, il nodo B viene sostituito con il sottografo {B1, B2, B3, B4}. È importante notare come questa sostituzione non richieda alcuna modifica nei nodi A e C;

- consentono di individuare facilmente eventuali problemi o anomalie nella ripartizione delle funzionalità tra le varie componenti di un'applicazione parallela.

  Considerando ad esempio il grafo di computazione di figura B.2 (a), in cui per ogni nodo è riportata la stima del tempo di esecuzione, se i quattro moduli vengono eseguiti in parallelo su quattro processori, è richiesto un tempo di esecuzione totale pari a 11.

  Si può notare facilmente come l'esecuzione dei primi due nodi richieda un tempo di esecuzione molto superiore (9) a quello richiesto per l'esecuzione degli altri due (1). Questa "anomalia" può suggerire, se possibile, una ristrutturazione del programma, in modo da ottenere un insieme di procedure con tempi di esecuzione paragonabili, come mostrato in figura B.2 (b): in questo caso il carico di lavoro totale rimane invariato (pari a 20), ma il miglior bilanciamento permette di ridurre a 8 il tempo di esecuzione totale.

B.1.1 Esempi di utilizzo dei grafi di computazione

Vengono presentati due esempi, che dimostrano come i grafi di computazione possono essere un utile strumento per definire applicazioni di tipo parallelo.

Nel primo caso viene esaminato un programma parallelo progettato secondo il modello a memoria condivisa, mentre nel secondo esempio viene considerata un'applicazione basata sul modello a scambio di messaggi [18].

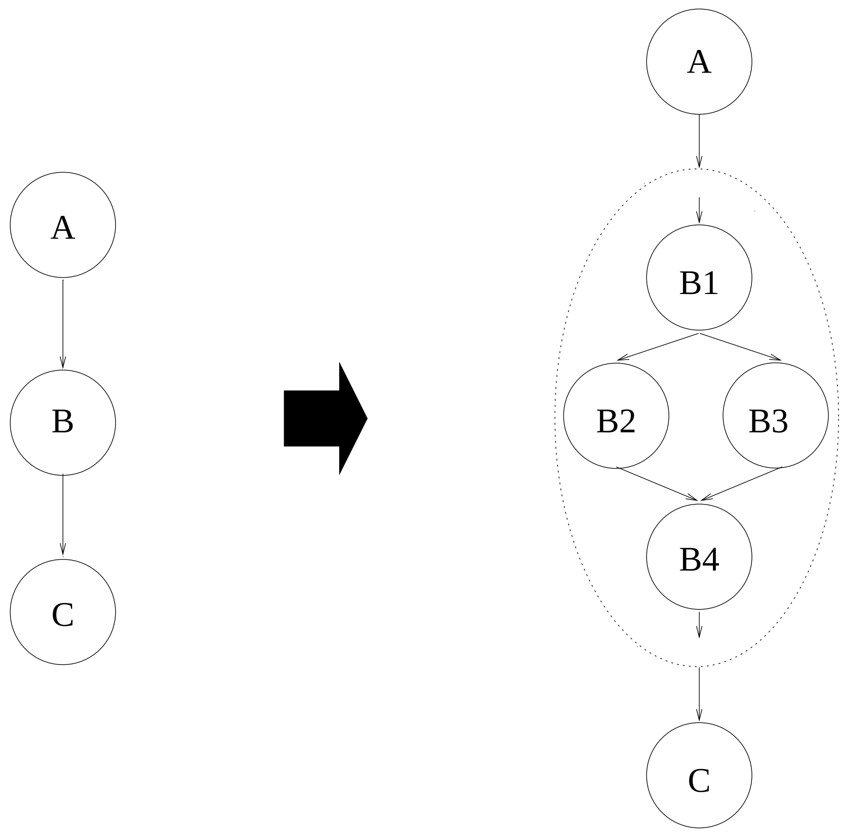

Figura B.1: Esempio di utilizzo dei grafi di computazione per la programmazione top-down

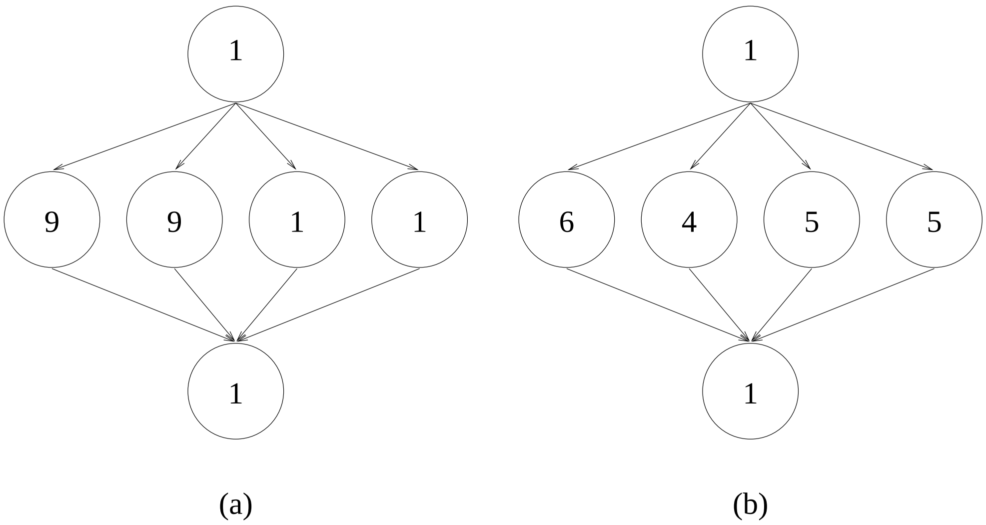

Figura B.2: Due grafi di computazione che rappresentano un'applicazione parallela, rispettivamente prima (a) e dopo (b) una ristrutturazione

Esempio di applicazione parallela basata sul modello a memoria condivisa

Si consideri un programma composto da cinque moduli (*M1*, *M2*, *M3*, *M4*, *M5*), che operano su cinque variabili condivise (x, y, z, v, w):

```
          M1                  M2                  M3
           .                   .                   .
(1)        .         (3)       .         (5)       .
           .                   .                   .

(a)     x:=1         (b)    y:=2         (c)   z:=x+y+3

           .                   .                   .
(2)        .         (4)       .         (6)       .
           .                   .                   .
```

```
            M4                        M5
             .                         .
(7)          .              (9)        .
             .                         .

(d)   v:=x+4                (e)  w:=x+z+v+5

             .                         .
(8)          .              (10)       .
             .                         .
```

Le istruzioni delle porzioni di codice non specificate (rappresentate dai numeri 1-10) non accedono alle cinque variabili condivise.

Per definire il comportamento del programma, anziché elencare tutte le possibili combinazioni descriventi l'ordine in cui le varie istruzioni possono essere eseguite, si può utilizzare un grafo di computazione, come indicato in figura B.3

I nodi del grafo rappresentano istruzioni, o porzioni di codice (il nodo i rappresenta l'istruzione/porzione di codice *(i)*). Gli archi, invece, rappresentano vincoli di precedenza: se esiste un arco $x \rightarrow y$, il codice *(x)* dovrà essere eseguito prima del codice *(y)* (ad esempio, con riferimento al caso considerato, l'istruzione *(e)* dovrà essere eseguita dopo le istruzioni *(a)*, *(c)* e *(d)*). Se invece non vi sono archi tra due nodi, le corrispondenti istruzioni/porzioni di codice possono essere eseguite in parallelo (ad esempio, sempre con riferimento al grafo di figura, le istruzioni *(a)* e *(b)* possono essere eseguite in parallelo).

Esempio di applicazione parallela basata sul modello a scambio di messaggi

Si consideri un programma costituito da moduli che invocano routine di una libreria di scambio di messaggi.

Quando un modulo deve inviare un messaggio ad un altro modulo, utilizza una procedura *send()*; al contrario, quando un modulo necessita di un dato da parte di un altro modulo, invoca una procedura *receive()*: l'esecuzione è sospesa finché il dato non viene ricevuto.

Il comportamento di un programma di questo tipo può essere rappresentato da un grafo di computazione, in cui i nodi descrivono i moduli, mentre gli archi rappresentano il flusso dei messaggi.

Nell'esempio di figura B.4 è rappresentata un'applicazione parallela costituita da tre moduli (A, B, C), che comunicano tra loro attraverso messaggi (nel grafo

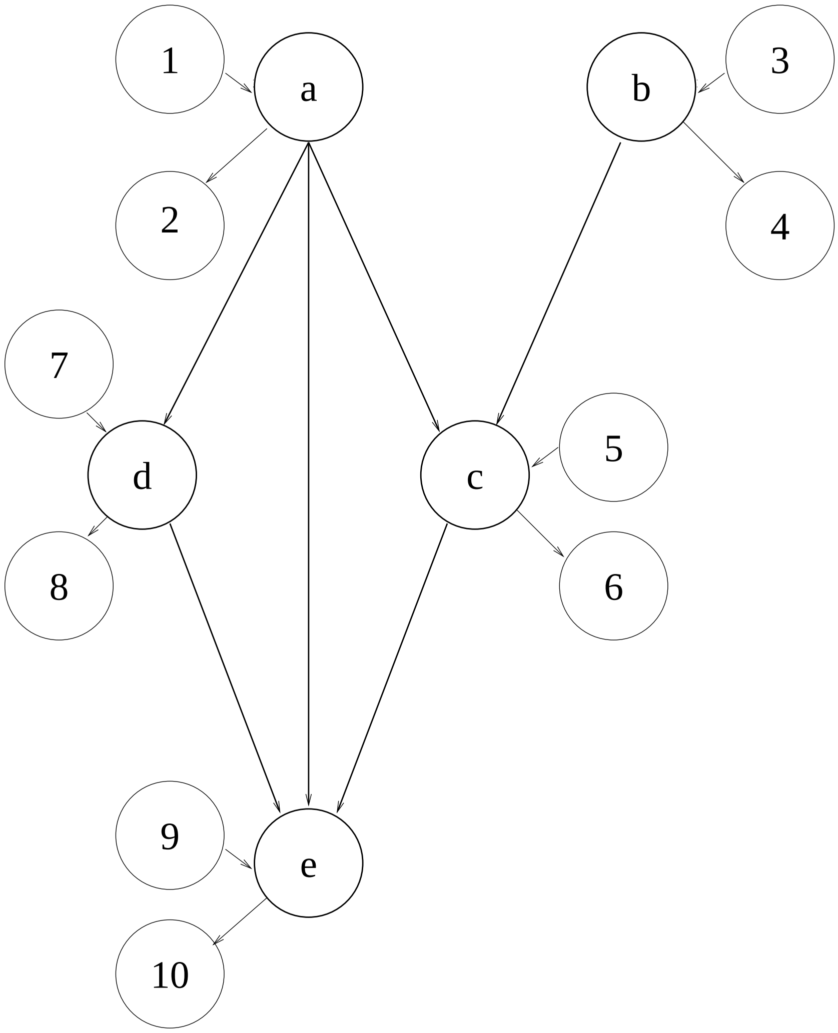

Figura B.3: Grafo di computazione con vincoli di precedenza tra i moduli costituenti il programma parallelo

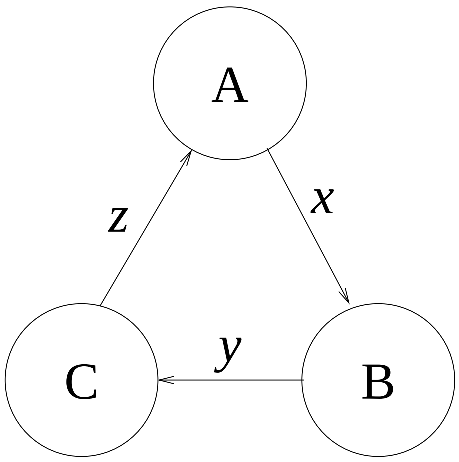

Figura B.4: Grafo di computazione descrivente le comunicazioni tra i moduli di un programma parallelo

gli archi sono etichettati con il valore dei dati che vengono scambiati). Ad esempio l'arco dal nodo A al nodo B, etichettato con il valore x, indica che il modulo A invia al modulo B un messaggio, contenente il dato x.

B.2 Analisi di alcuni ambienti per la programmazione parallela

Nei prossimi paragrafi vengono esaminati alcuni ambienti visuali per la programmazione parallela.
In particolare vengono considerati i sistemi:

- HeNCE
- CODE
- VPE
- Enterprise
- Paralex
- Meander

Per ognuno di questi ambienti vengono analizzate le specifiche caratteristiche e funzionalità, con particolare attenzione alle esigenze per il sistema DEMI.

B.3 HeNCE

HeNCE (Heterogeneous Network Computing Environment) [13] [14] [18] è un ambiente grafico per la programmazione di applicazioni parallele.
Utilizzando questo strumento, il programmatore, attraverso un'apposita interfaccia grafica, può:

- definire un'applicazione parallela;
- compilarla;
- eseguirla;
- farne il tracing.

I programmi sviluppati attraverso HeNCE possono essere eseguiti in un ambiente eterogeneo di workstation Unix, che utilizzano PVM [25] come sistema di comunicazione a scambio di messaggi.

B.3.1 Definizione di un'applicazione

In HeNCE un programma viene definito attraverso un grafo di computazione, in cui i nodi rappresentano procedure (scritte in linguaggio C o Fortran), mentre gli archi definiscono vincoli di precedenza: un arco da un nodo A ad un nodo B indica quindi che il codice associato al nodo A deve essere eseguito prima del codice associato al nodo B.

In figura B.5 è mostrato il tool grafico che viene utilizzato per la specifica del grafo di computazione (il grafo può comunque essere definito anche attraverso un'interfaccia di tipo testuale).
Nella definizione di un grafo di computazione si possono utilizzare sei tipi di costrutti:

- Nodi **subroutine**

  Ad un nodo subroutine è associata una procedura, una lista di parametri di input, e una lista di parametri di output. Quando un nodo subroutine deve essere eseguito, viene invocata la procedura ad esso associato.

  In HeNCE un nodo subroutine può comunicare con i nodi adiacenti solo all'inizio e alla fine della propria esecuzione: le liste dei parametri di input e di output definiscono quali dati devono essere scambiati in questa comunicazione. Non è invece ammesso l'utilizzo di variabili globali.

  Ad un nodo subroutine possono essere associate più implementazioni, una per ogni architettura presente nel sistema eterogeneo considerato.

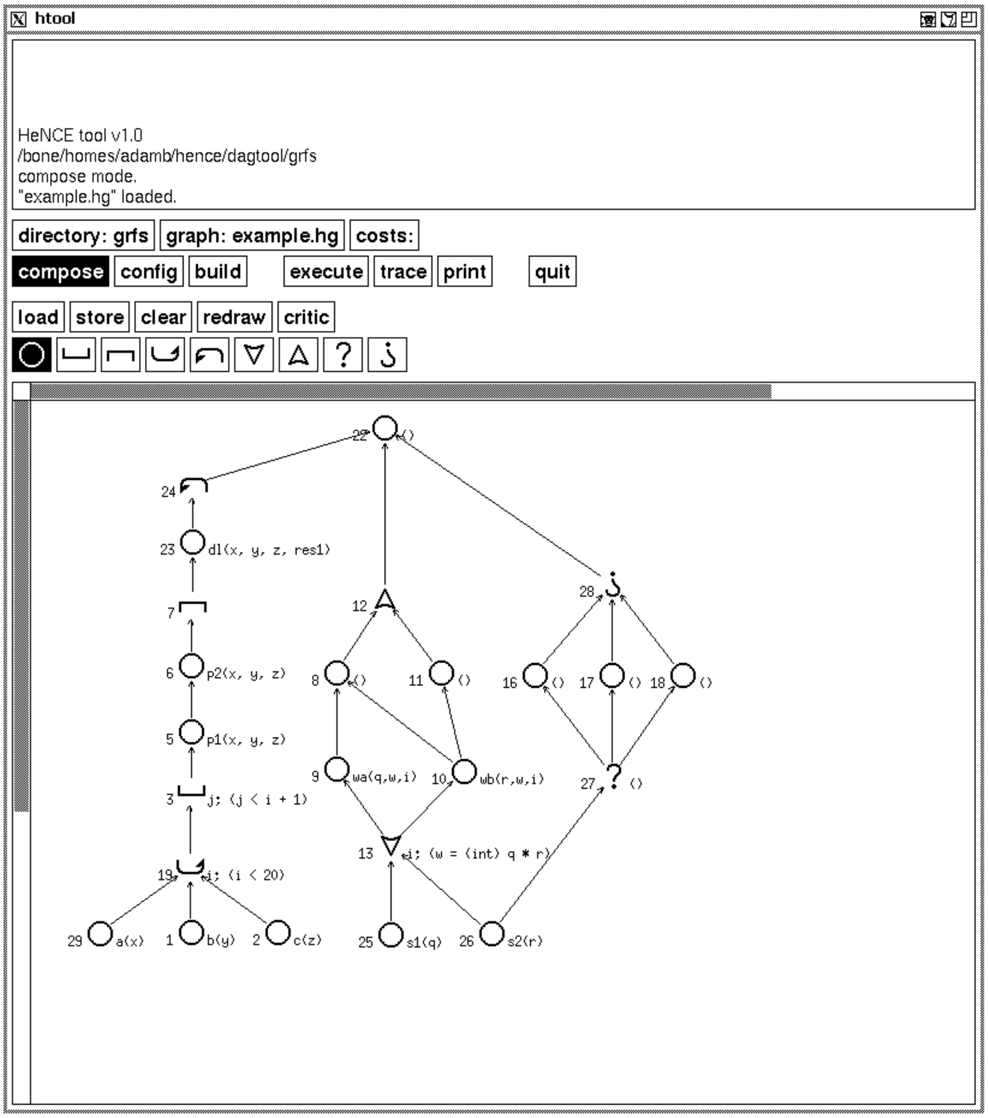

Figura B.5: Tool HeNCE per la definizione di un'applicazione parallela

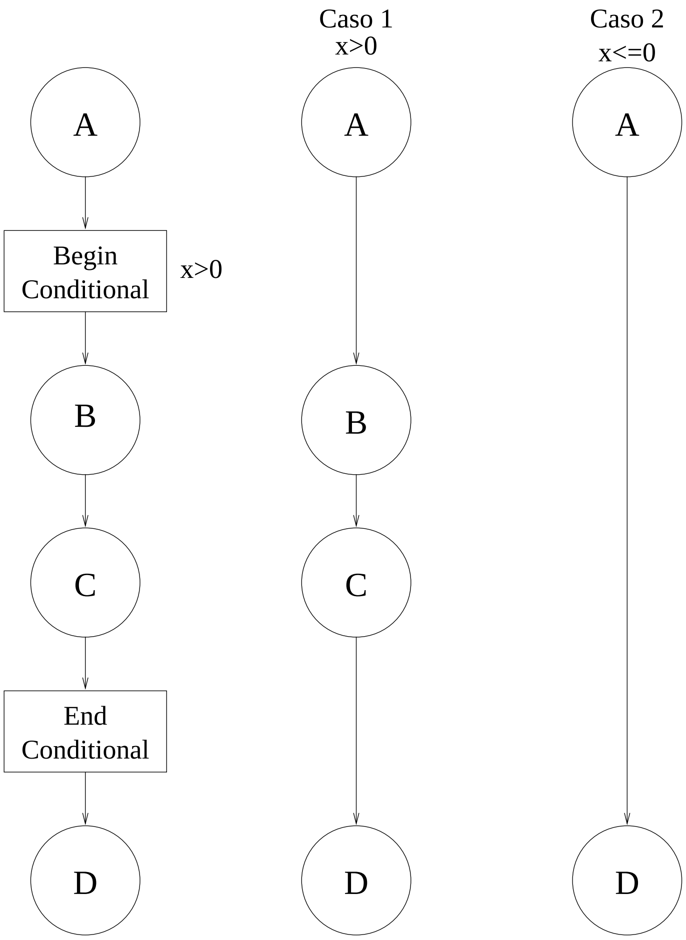

Figura B.6: Esempio di utilizzo del costrutto conditional

- Archi **single dependency**

  Un arco dependency collega due nodi subroutine, definendo un vincolo di precedenza tra questi due nodi.

- Costrutti **conditional**

  Usando il costrutto conditional, è possibile definire un sottografo che deve essere eseguito solo in certe condizioni: ad un nodo *Begin Conditional* viene associata un'espressione booleana (v. figura B.6); se questa è vera, allora viene eseguito il codice associato al sottografo compreso tra i nodi *Begin Conditional* e *End Conditional*. Se invece l'espressione booleana è falsa, tale codice non viene eseguito: l'esecuzione riprende dal nodo successivo al nodo *End Conditional*.

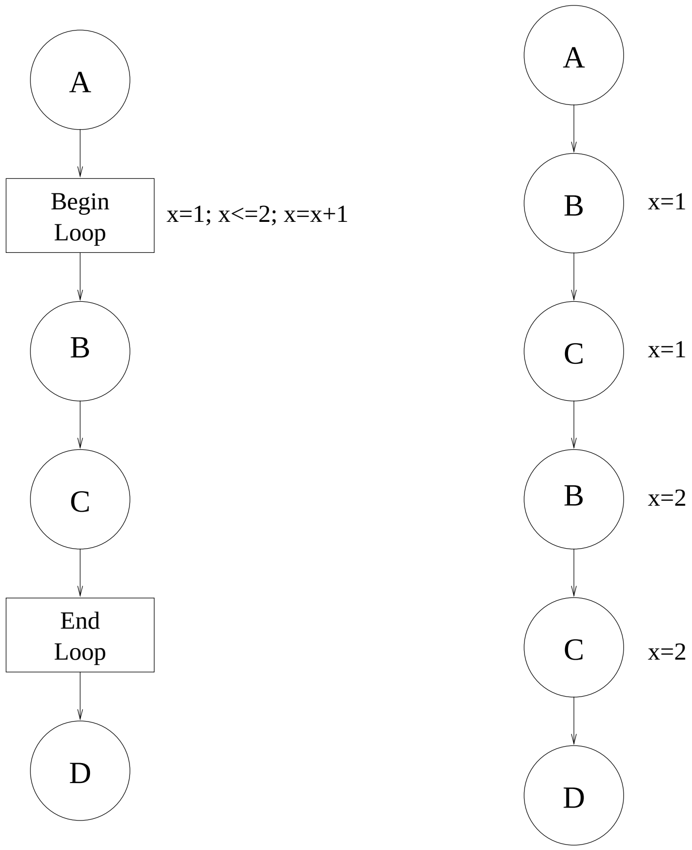

Figura B.7: Esempio di utilizzo del costrutto loop

- Costrutti **loop**

  Attraverso questo costrutto si esegue ripetutamente (un numero di volte pari al valore di un'espressione) il codice associato ai nodi di un sottografo, che costituisce il corpo del ciclo.

  La sintassi e la semantica sono del tutto analoghe a quelle del costrutto *for* del linguaggio C: ad un particolare nodo *Begin Loop* viene associato il nome di una variabile indice, il suo valore iniziale, l'istruzione che ne modifica il valore, e la condizione di terminazione. Il codice associato al sottografo delimitato dai nodi *Begin Loop* e *End Loop* viene eseguito finché la condizione di terminazione viene soddisfatta (dopo ogni iterazione viene eseguita l'istruzione che modifica il valore della variabile indice).

  Nell'esempio di figura B.7, il corpo del ciclo deve essere eseguito due volte.

- Costrutti **fan**

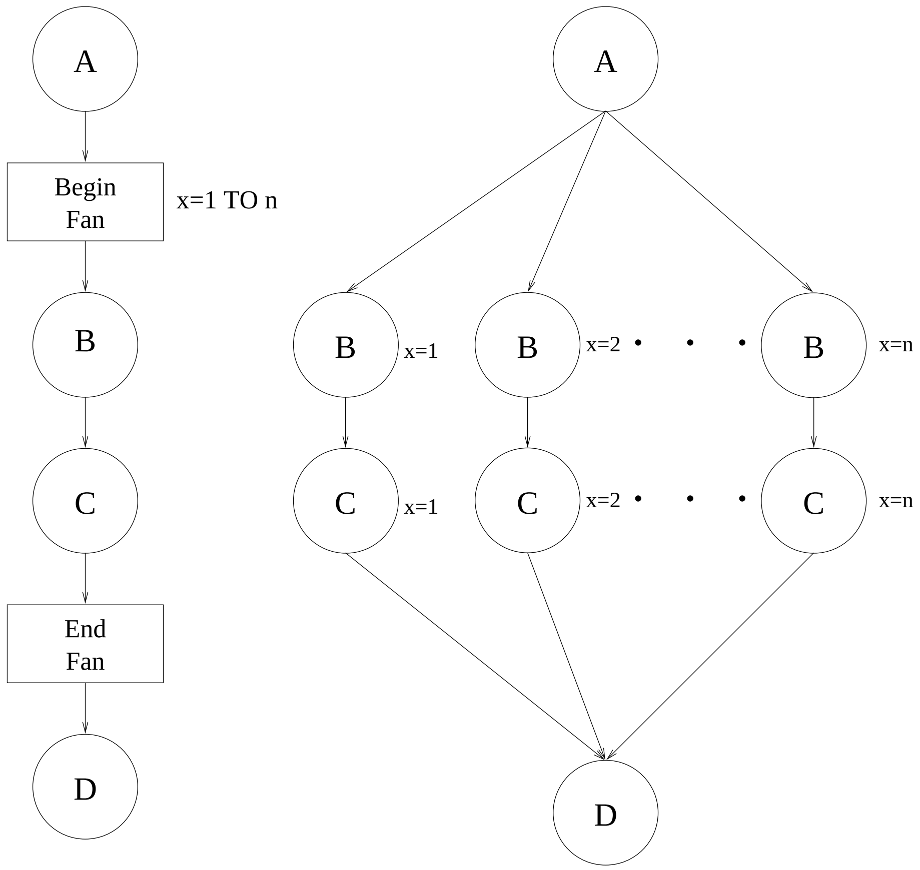

Figura B.8: Esempio di utilizzo del costrutto fan

Il costrutto fan permette di replicare un sottografo, e di eseguirne in parallelo il codice associato.

Ad un nodo *Begin Fan* (v. figura B.8) viene associata un'espressione del tipo:

```
var = exp1 TO exp2
```

dove *var* è il nome di una variabile, e *exp1* e *exp2* sono due espressioni numeriche. Il sottografo, delimitato dai nodi *Begin Fan* e *End Fan*, viene replicato un numero di volte pari a *exp2 - exp1 + 1*.

- Costrutti **pipe**

  Ad un particolare nodo *Begin Pipe* (v. figura B.9) viene associata un'espressione del tipo:

```
var = exp1 TO exp2
```

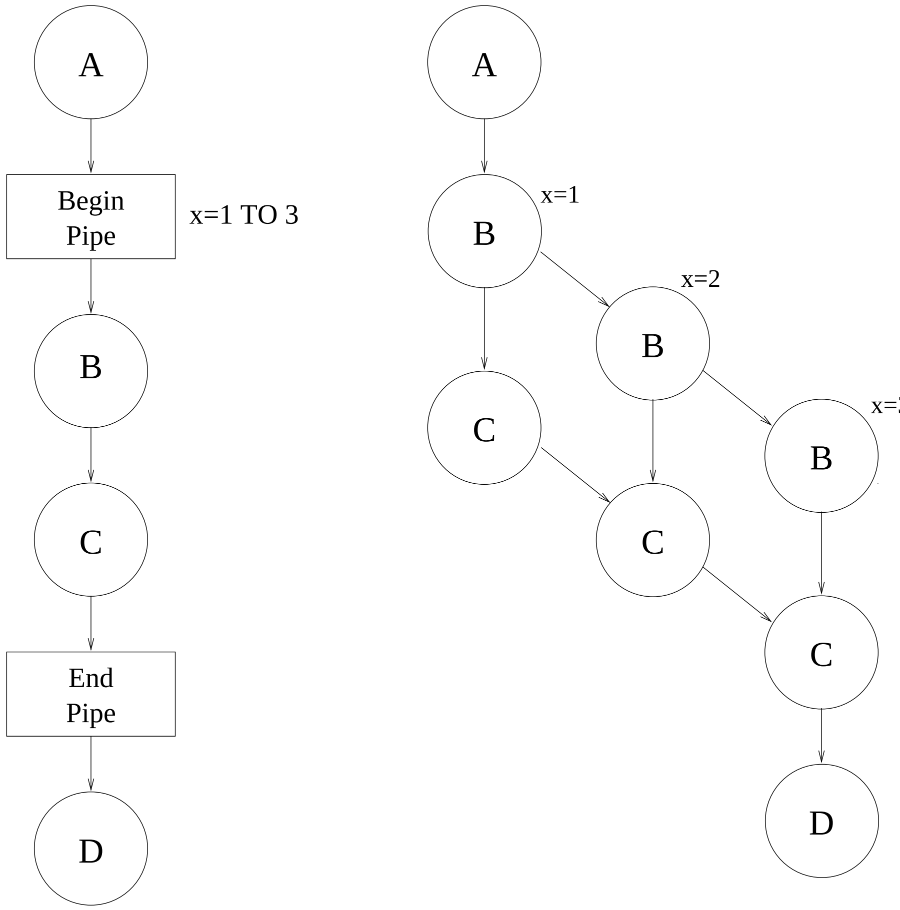

Figura B.9: Esempio di utilizzo del costrutto pipe

dove *var* è il nome di una variabile, e *exp1* e *exp2* sono due espressioni numeriche. Il sottografo delimitato dai nodi *Begin Pipe* e *End Pipe* viene replicato durante l'esecuzione un numero di volte pari a *exp2 - exp1 + 1*.

Con riferimento all'esempio di figura B.9, per ogni valore assunto dalla variabile x, vengono create e attivate nuove istanze dei nodi B e C, e vengono distrutte quelle precedenti.

B.3.2 Compilazione

Una volta che il grafo di computazione per l'applicazione parallela è stato definito, ed è stato specificato il codice associato ai vari nodi subroutine, il sistema "integra" tale codice con le opportune chiamate alle routine PVM.

Si procede poi con la compilazione, per ogni architettura presente nel sistema considerato.

B.3.3 Esecuzione

Dopo la compilazione, il programma HeNCE può essere eseguito sulla macchina virtuale definita dall'utente.

L'esecuzione dell'applicazione ha inizio dalla procedura associata al nodo iniziale del grafo, e termina quando sono state eseguite tutte le procedure associate ai nodi raggiungibili da questo. Una procedura, che corrisponde ad un certo nodo del grafo di computazione, può essere eseguita solo quando la procedura associata al nodo "predecessore" ha terminato la sua esecuzione.

Il programmatore ha la possibilità di definire la cosiddetta *matrice dei costi*, in cui specifica il tempo stimato per l'esecuzione delle varie procedure sui vari calcolatori costituenti la macchina virtuale: l'elemento *(m,p)* di questa matrice è un numero che rappresenta la stima del tempo di esecuzione per la procedura p sulla macchina m.

La matrice dei costi viene usata dall'algoritmo di mapping, che deve associare le varie procedure ai processori disponibili: indicando con $load_m$ il load del processore m, e con $cost_{m,p}$ l'elemento *(m,p)* della matrice dei costi, la procedura p viene "associata" al processore m per cui è minimo il valore della somma $load_m + cost_{m,p}$.

In questo algoritmo viene considerato solo il load dovuto a programmi HeNCE: non si considerano eventuali altri job di utenti generici.

B.3.4 Tracing

L'interfaccia HeNCE comprende un modulo per la visualizzazione dell'esecuzione di un programma. In figura B.10 è mostrato un esempio.

Attraverso questo strumento è possibile verificare come le varie procedure sono state "mappate" ai processori disponibili, si può controllare lo stato dei vari componenti della macchina virtuale (compresa l'attività dei processori), ecc...

B.4 CODE

CODE [18] [39] [41] è un ambiente visuale per lo sviluppo di applicazioni parallele, eseguibili su un'unica architettura.
Usando questo ambiente è possibile:

- definire applicazioni parallele;

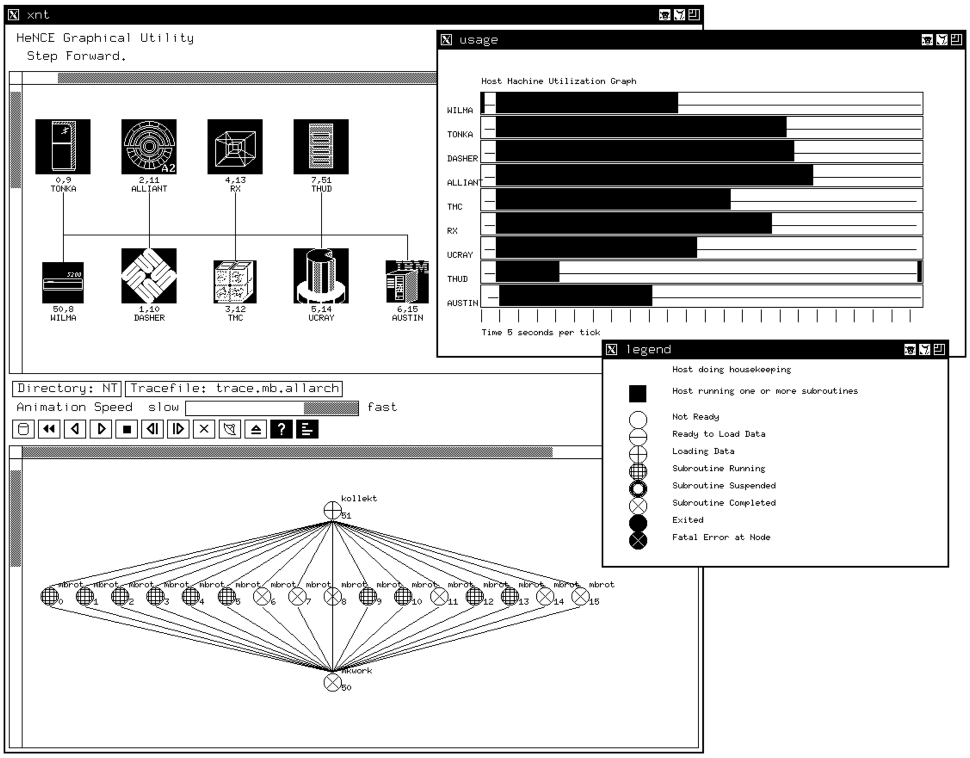

Figura B.10: Esempio di tracing di un'applicazione con HeNCE

- compilarle;
- eseguirle.

Non sono invece disponibili funzionalità di tracing o debugging.
In CODE viene usato PVM [25] come sistema di message passing.

B.4.1 Definizione di un'applicazione

Un'applicazione parallela viene definita in CODE attraverso un grafo, in cui i nodi rappresentano unità computazionali o variabili condivise, mentre gli archi rappresentano “flussi” di informazioni tra i nodi. Ogni arco collega due porte, ognuna delle quali è associata ad un nodo del grafo.
Esistono diversi tipi di nodi con i quali si può costruire il grafo:

- Nodi **Unit Computation** (**UC**)

  Un nodo di questo tipo rappresenta un'elaborazione sequenziale: ad esso è associata una procedura (scritta in linguaggio C).

  Ad un nodo UC sono poi associati dati locali, *regole di firing*, e *regole di routing*. Le regole di firing definiscono le condizioni che devono essere soddisfatte affinché la procedura possa essere eseguita. Le regole di routing, invece, definiscono in che modo il risultato della computazione viene inviato ai vari nodi.

  In figura B.11 è mostrato un esempio di specifica delle regole di firing e routing per un nodo UC: in questo caso le regole di firing stabiliscono che l'esecuzione deve iniziare solo quando, sulle porte A e B sono disponibili dei valori, che vengono assegnati rispettivamente alle variabili X e Y. La regola di routing, invece, definisce che, se l'espressione booleana $X > 1$ *and* $Y < 2$ è vera, allora il valore $X{+}Y$ deve essere assegnato alla porta C.

- Nodi **Name sharing**

  Un nodo di questo tipo rappresenta un insieme di variabili che possono essere condivise tra le procedure definite nel programma. Questo nodo viene poi collegato con archi ai nodi che utilizzano questi dati comuni (CODE gestisce direttamente la mutua esclusione per l'accesso a queste variabili).

- Nodi **Call** e nodi **Input/Output interface**

  Un nodo call viene utilizzato quando un grafo deve “chiamare” un altro grafo, in modo da collegare le corrispondenti computazioni.

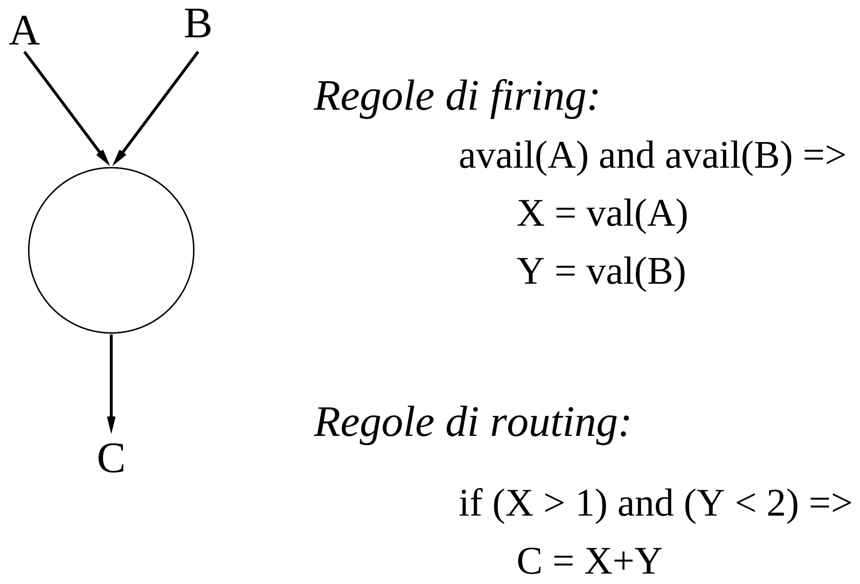

Figura B.11: Esempio di nodo Unit Computation, con le associate regole di firing e di routing

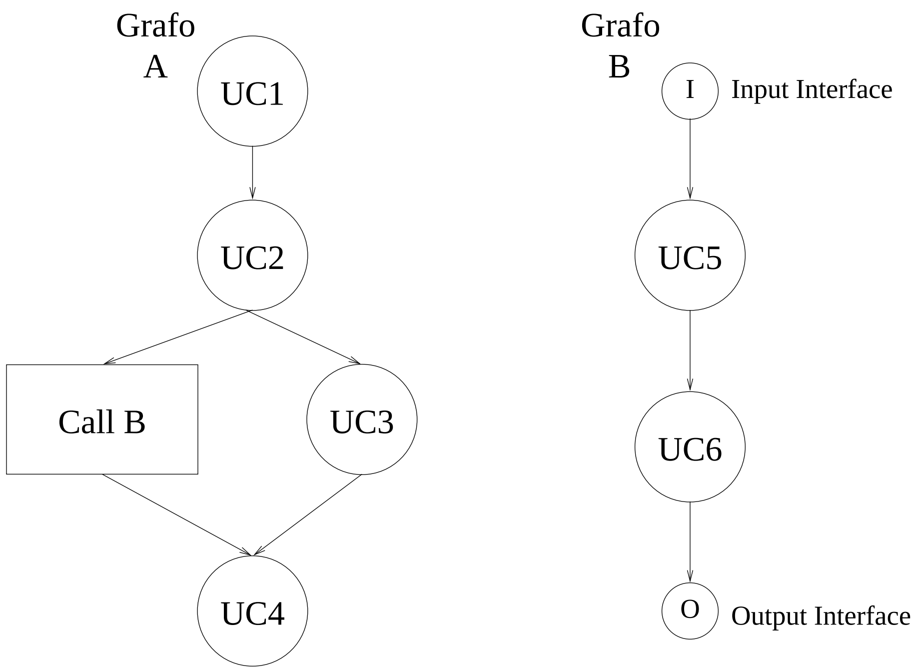

Figura B.12: Esempio di utilizzo dei nodi Call e dei nodi Interface

Nell'esempio di figura B.12 il grafo A contiene un nodo call, che "chiama" il grafo B.

Nel grafo chiamato sono presenti un nodo input interface e un nodo output interface (i nodi I e O del grafo B), che gestiscono il passaggio dei parametri tra grafo chiamante e grafo chiamato.

Quando un grafo "invoca" un altro grafo, l'esecuzione nel grafo chiamante prosegue con il nodo UC successivo, senza dover attendere che l'elaborazione delle procedure associate al grafo chiamato sia stata completata. In questo modo è possibile avere in esecuzione più procedure allo stesso tempo.

- Nodi **Creation**

  Un nodo creation è un particolare nodo di tipo input interface, tale che un parametro associato ad un nodo di questo tipo è visibile (ma non modificabile) da tutti i nodi del grafo chiamato (un parametro associato ad un nodo input interface, invece, è visibile e modificabile solo dal nodo UC che lo riceve).

Nel modello CODE un'applicazione parallela può quindi essere rappresentata da più grafi; tra questi ne è presente uno particolare, chiamato *main*, che contiene il nodo UC associato alla procedura che deve essere eseguita per prima. Il grafo main può poi contenere "chiamate" agli altri grafi.

B.4.2 Compilazione ed esecuzione

Dopo che il grafo è stato definito, CODE integra il codice specificato dall'utente con le chiamate alle apposite procedure della libreria PVM, che si occupano di gestire la comunicazione tra i processi.

Il programma viene poi compilato sul calcolatore considerato, e mandato in esecuzione.

B.5 VPE

VPE (Visual Programming Environment) [40] è un ambiente visuale per la programmazione di applicazioni parallele, basate sul modello a scambio di messaggi.

VPE può essere utilizzato in un ambiente eterogeneo, costituito da un insieme di workstation che utilizzano il sistema a scambio di messaggi PVM [25] o MPI [54].

Usando VPE il programmatore ha la possibilità di:

- definire un'applicazione parallela;
- compilarla;
- eseguirla.

Per le future implementazioni di VPE è prevista la realizzazione di una funzionalità di tracing, attraverso la quale l'utente avrà la possibilità di controllare direttamente lo stato di esecuzione dell'applicazione.

B.5.1 Definizione di un'applicazione

Per specificare un'applicazione parallela, può essere usato un apposito tool grafico: l'applicazione viene definita attraverso un insieme di grafi.

I nodi di questi grafi rappresentano processi, mentre gli archi definiscono "canali" attraverso cui passano i messaggi inviati da un processo ad un altro. Ogni arco connette due porte, ognuna associata ad un nodo del grafo.

Per costruire il grafo che definisce l'applicazione, si possono utilizzare quattro tipi di costrutti:

- Nodi **compute** (**comp**)

  Un nodo compute rappresenta un processo.

  Ad un nodo di questo tipo è associato codice (scritto in linguaggio C o Fortran) che può contenere chiamate a routine VPE per lo scambio di messaggi. Il programmatore può indicare uno specifico computer (o un tipo di computer) per l'esecuzione di questo codice.

  Ogni nodo compute può essere replicato: al momento dell'esecuzione saranno attivate diverse istanze, ognuna operante su propri dati.

- **Porte di input e di output**

  Ad ogni nodo compute possono essere associate porte di input e/o porte di output, usate per lo scambio di messaggi con gli altri processi: quando si deve inviare/ricevere un messaggio, viene invocata un'apposita routine VPE, in cui si specifica il nome della porta di output/input da usare.

- Nodi **call** e **interface**

  Un'applicazione parallela può essere definita attraverso più grafi collegati tra loro. Il nodo call viene utilizzato per rappresentare la "chiamata" ad un grafo, da parte di un altro grafo.

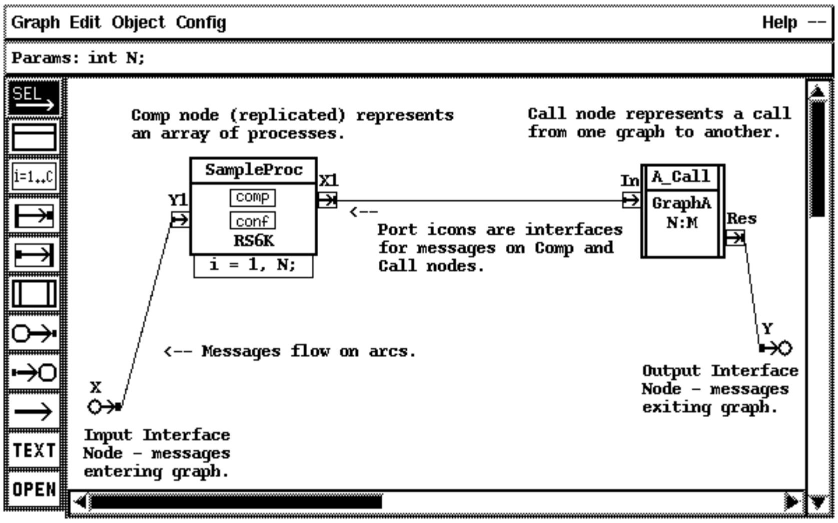

Figura B.13: Esempio di grafo VPE

- Nodi **interface**

  Il grafo "chiamato" attraverso un nodo call deve contenere due nodi particolari, chiamati *nodo input interface*, e *nodo output interface*: la loro funzione è quella di gestire il passaggio dei parametri tra il grafo chiamante e il grafo chiamato.

In figura B.13 viene mostrato un esempio di grafo VPE, in cui compaiono i quattro costrutti descritti.

A differenza di HeNCE e CODE, in cui la comunicazione tra i diversi moduli può avvenire solo all'inizio o alla fine della propria esecuzione, in VPE ogni modulo può interagire con gli altri moduli in qualsiasi momento.

B.5.2 Compilazione ed esecuzione

Una volta che l'applicazione parallela è stata definita, il sistema sostituisce le chiamate alle routine VPE con chiamate alle procedure della libreria di comunicazione utilizzata (PVM o MPI).

Il programma viene poi compilato ed eseguito.

Per quel che riguarda il problema del mapping, ogni modulo viene "associato" al calcolatore specificato dal programmatore durante la definizione dell'applicazione; nel caso in cui sia stato definito solamente il tipo di computer da utilizzare, viene scelta la "prima" macchina disponibile di quell'architettura.

B.6 Enterprise

Enterprise [52] è un ambiente visuale che permette:

- la definizione
- la codifica
- l'esecuzione
- il debugging
- il tracing

di applicazioni che devono essere eseguite in un ambiente distribuito, costituito da un insieme di workstation Unix.

Enterprise può utilizzare come sistema di message-passing PVM [25], ISIS [15], o NMP [35].

B.6.1 Definizione di un'applicazione parallela

Come visto, la definizione di un'applicazione parallela può essere divisa in due fasi: la definizione delle componenti sequenziali, e la specifica delle relazioni tra questi elementi. Con riferimento a questo modello, Enterprise usa una metodologia di programmazione basata sull'uso di due componenti:

- Il **Programming Model**

  Permette di definire il codice dei vari moduli costituenti l'applicazione parallela.

- Il **Metaprogramming Model**

  Permette di specificare le relazioni che esistono tra i vari moduli, attraverso la definizione del cosiddetto *asset graph*.

Il programmatore, nella definizione dell'applicazione parallela, ha quindi la possibilità di operare in due possibili modi:

1. disegnare l'asset graph, e poi definire il codice associato alle varie procedure;
2. scrivere il codice associato alle procedure, e in un secondo tempo definire l'asset graph.

Programming model

Nel programming model un'applicazione parallela viene definita attraverso un insieme di moduli, che interagiscono tra loro.

Ogni modulo è costituito da una *entry procedure*, che può essere invocata dagli altri moduli, e da una serie di *internal procedure*, che possono essere chiamate solo dalle altre procedure definite nello stesso modulo. L'entry procedure e le internal procedure vengono scritte dal programmatore in linguaggio C. Ogni modulo può accedere solo alle variabili locali: non è ammesso l'uso di variabili globali.

Utilizzando esclusivamente il programming model, l'unico strumento per definire le interazioni tra le diverse componenti è l'uso delle chiamate ai moduli. Considerando ad esempio un modulo A che invoca durante la sua esecuzione un modulo B, si possono presentare due possibili casi:

1. Il modulo chiamato B deve restituire un valore al modulo chiamante A (*f-call*).

   In questo caso A blocca la sua esecuzione, in attesa che B restituisca il dato.

2. Il modulo chiamato B non deve restiture un valore al modulo chiamante A (*p-call*).

   In questo caso A non deve attendere il completamento dell'esecuzione del modulo B: i due moduli possono essere eseguiti in parallelo (*parallelismo puramente asincrono*).

Metaprogramming model

Nel metaprogramming model il programmatore definisce la struttura del programma, costruendo l'asset graph.

La definizione dell'asset graph avviene selezionando alcuni elementi, detti *asset*, che ne costituiscono i nodi. Un asset può anche essere replicato, in modo che vi sia più di un processo che ne esegue il codice associato: in questo caso il programmatore deve specificare il fattore di replicazione, ovvero il numero di istanze che devono essere eseguite.

Esistono vari tipi di asset:

- **Enterprise**

  L'applicazione parallela è all'inizio rappresentata da un singolo asset di tipo enterprise. Durante la fase di sviluppo, questo viene "espanso" con altri asset, come rappresentato nell'esempio di figura B.14.

  Un asset enterprise non può essere replicato.

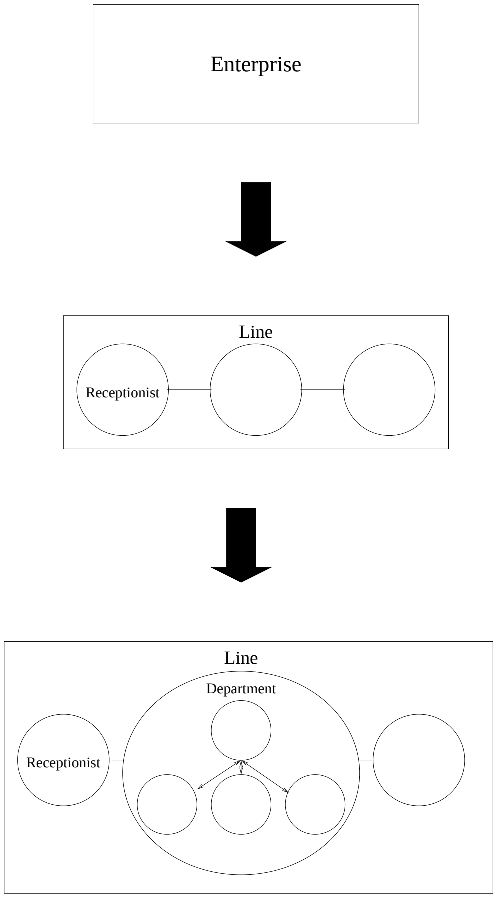

Figura B.14: Esempio di “espansione” di un asset enterprise

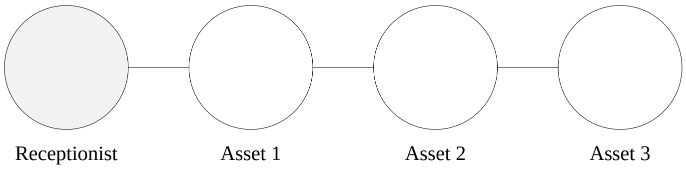

Figura B.15: Esempio di utilizzo dell'asset line

- **Individual**

  Un asset individual rappresenta un singolo modulo di tipo sequenziale. Ad esso è associata una procedura, che viene eseguita quando viene invocata da una procedura di un altro asset di tipo individual.

  Un asset individual non può contenere altri asset.

- **Line**

  Un asset di tipo line è costituito da un insieme di asset ordinati sequenzialmente. Il primo di questi asset è di tipo receptionist: suo compito è quello di ricevere i messaggi che arrivano in input all'asset line, e di trasmetterli all'asset successivo. Ogni asset componente un asset di tipo line può invocare solo l'asset successivo (v. figura B.15).

- **Department**

  Un asset di tipo department è costituito da un certo numero di asset (che possono anche essere di tipo diverso), ognuno dei quali svolge una specifica funzione. Tra gli asset componenti un asset department deve esistere un asset di tipo receptionist, che di volta in volta riceve dall'esterno un messaggio, e lo inoltra ad un altro opportuno asset. L'eventuale messaggio di ritorno elaborato da questo asset viene inviato al receptionist, che poi lo inoltra all'esterno.

- **Division** e **Representative**

  L'asset di tipo division viene utilizzato per parallelizzare una procedura ricorsiva, progettata secondo il modello *divide-et-impera*.

  Un asset di questo tipo contiene un asset receptionist, e un asset representative, che rappresenta l'insieme delle chiamate ricorsive alla procedura associata all'asset division. Replicando l'asset representative, è possibile ottenere un'esecuzione parallela di questa procedura. Se con m si indica il fattore di replicazione, si ha una situazione come quella rappresentata in figura B.16.

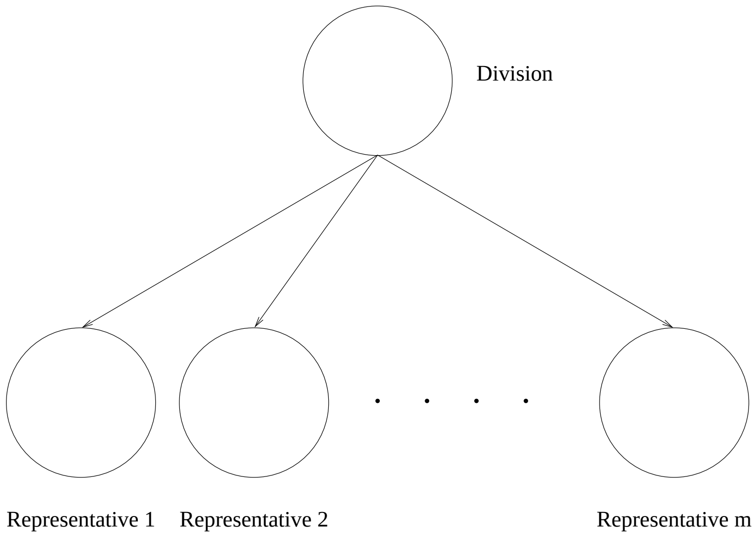

Figura B.16: Esempio di utilizzo degli asset division e representative

Per aumentare il grado di parallelismo, si può sostituire l'asset representative con un asset division, e replicare l'asset representative contenuto in quest'ultimo asset division, come rappresentato in figura B.17.

- **Receptionist**

  Come visto, la funzione svolta da un asset di tipo receptionist varia a seconda dell'asset composto in cui esso è inserito. In generale si può dire che l'asset receptionist è l'unico asset, tra quelli contenuti in quello composto a cui appartiene, ad essere "visibile" dall'esterno. Svolge infatti funzione di interfaccia tra gli asset contenuti nell'asset composto e tutti gli altri asset del programma.

  Un asset receptionist non può essere replicato.

- **Service**

  Ad un asset di tipo service è associata una procedura sequenziale che, a differenza di un asset di tipo individual, non può invocare nessun'altra procedura del programma. Inoltre essa può essere chiamata da una qualsiasi altra procedura del programma, indipendentemente dalle regole di visibilità degli asset del programma: questo può permettere la condivisione di variabili tra tutte le procedure del programma.

  Un asset service non può contenere altri asset.

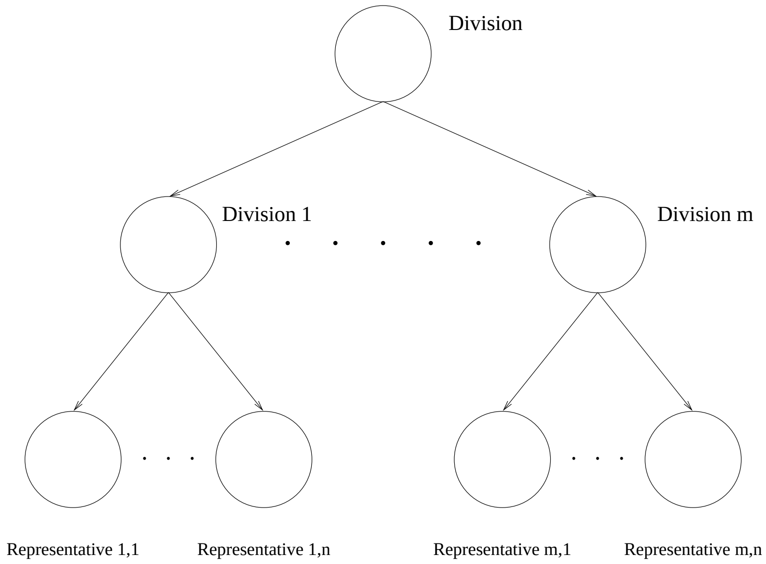

Figura B.17: Esempio di sostituzione di un asset division con un asset representative, per aumentare il grado di parallelismo

B.6.2 Compilazione

Una volta che la definizione dell'applicazione parallela è stata completata, attraverso la specifica dell'asset graph, e le varie procedure sono state codificate, il sistema integra il codice utente con chiamate a routine della libreria utilizzata (PVM, ISIS, o NMP), in modo da gestire le comunicazioni e le sincronizzazioni tra i vari processi. Il tutto viene poi compilato su ogni calcolatore costituente la macchina virtuale considerata.

B.6.3 Esecuzione

Quando l'applicazione parallela deve essere eseguita, viene creato un processo per ogni asset componente l'asset graph. Questi processi devono essere assegnati ai vari processori per la loro elaborazione: Enterprise individua quali calcolatori sono disponibili e quali invece stanno processando un job, in modo da bilanciare il load complessivo.

B.6.4 Tracing e debugging

Enterprise permette di "animare" l'esecuzione di un programma, in modo da verificare quali calcolatori vengono usati, qual è lo stato dei vari processi (idle, busy, wait), quanti e quali messaggi vengono scambiati tra i vari moduli, ecc...

L'animazione dell'esecuzione del programma è un utile strumento anche per il debugging: si possono infatti mettere in evidenza eventuali errori presenti nel codice, o nella struttura dell'applicazione.

Un altro strumento molto utile è la possibilità di registrare in un *event file* tutti gli eventi significativi avvenuti durante l'esecuzione. Utilizzando l'event file è poi possibile simulare una riesecuzione del programma, definendo dei breakpoint dove interrompere l'esecuzione, per poter esaminare il valore delle varie variabili.

B.7 Paralex

Paralex [8] [9] [12] [22] è un ambiente visuale per:

- la definizione
- la codifica
- l'esecuzione
- il tracing

di applicazioni parallele che devono essere eseguite in ambienti eterogenei e distribuiti, costituiti da una rete di workstation Unix, che usano ISIS [15] come sistema di message-passing. Paralex fornisce inoltre funzionalità per la gestione del fault tolerance e del load balancing dinamico.

B.7.1 Definizione della macchina virtuale

In Paralex prima di tutto è necessario definire la macchina virtuale che poi eseguirà l'applicazione parallela. Ogni calcolatore viene descritto attraverso un insieme di attributi, scelti dal programmatore; questi possono essere, ad esempio, il nome dell'architettura, il sistema operativo, un parametro che descrive le performance, ecc...

Una definizione della macchina virtuale può essere del tipo:

```
osde00       osf, perf=100, color, memory=128
osde02       osf, perf=90, memory=64
hpcms01      hpux, perf=100, color, memory=128
lnxpc01      linux, perf=50, memory=128
cmssun1      solaris, perf=110, memory=256
```

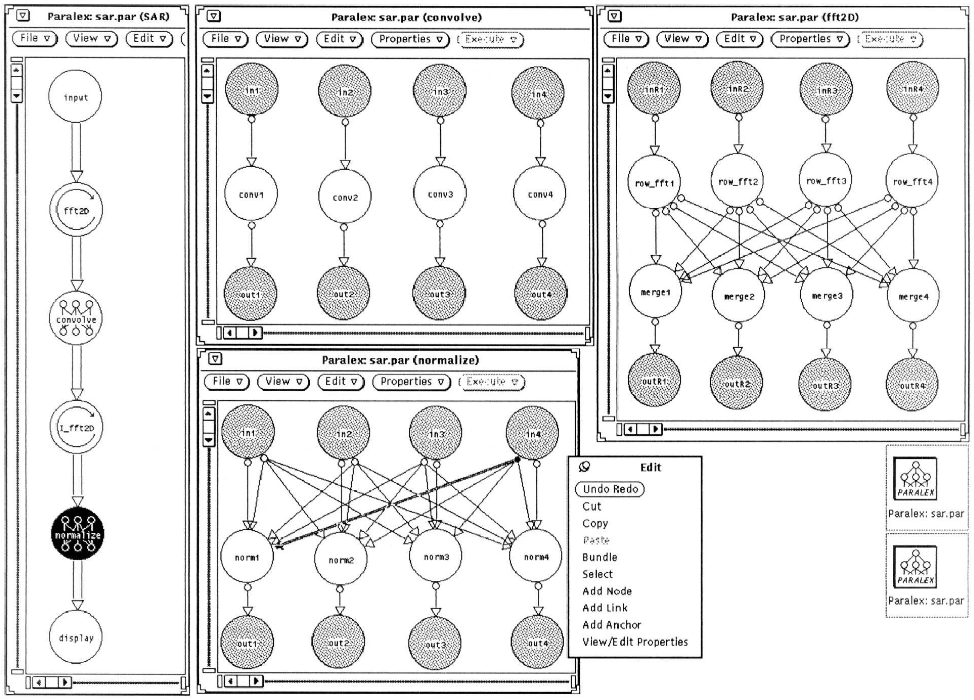

Figura B.18: Tool Paralex per la definizione di un'applicazione

Questi attributi vengono poi utilizzati dal sistema per la scelta dei calcolatori più adatti ad eseguire i vari moduli dell'applicazione parallela.

B.7.2 Definizione di un'applicazione

Attraverso un apposito tool grafico, mostrato in figura B.18, il programmatore definisce l'applicazione parallela attraverso un grafo diretto aciclico, in cui i nodi rappresentano unità computazionali, e gli archi definiscono "canali" attraverso i quali "passano" i dati tra i nodi connessi. La computazione associata ad un nodo può essere eseguita solo quando tutti gli archi incidenti quel nodo "contengono" valori.

Ogni nodo rappresenta una funzione, con uno o più input, e un solo output. Questa funzione deve essere "pura": non sono ammessi effetti collaterali, e non è possibile l'interazione con oggetti esterni, quali file, device, ecc...

Il codice associato a ciascun nodo può essere scritto in un qualsiasi linguaggio di programmazione, "compatibile" con il linguaggio C (cioè una procedura scritta in questo linguaggio deve poter essere invocata da un programma scritto in linguaggio C); è anche possibile associare ad un nodo un file binario eseguibile.

Per ogni nodo si devono specificare le caratteristiche del calcolatore che dovrà eseguirne il codice associato: questo viene fatto definendo un'espressione booleana, in cui si considerano gli attributi utilizzati nella definizione dei computer costituenti la macchina virtuale. In questo modo si può scegliere, ad esempio, il tipo di architettura del sistema che dovrà eseguire il modulo, le performance minime richieste per tale computer, ecc....

B.7.3 Compilazione

Una volta che il grafo di computazione è stato definito, ed è stato specificato il codice associato ai vari nodi, il programma può essere compilato.

Poiché si considera un ambiente eterogeneo, in cui i vari calcolatori possono usare diverse rappresentazioni interne dei dati, c'è la necessità di una "infrastuttura" comune, che consenta di rappresentare i dati allo stesso modo: a tal fine in Paralex viene utilizzato il toolkit ISIS.

L'applicazione parallela viene compilata remotamente sui diversi calcolatori che poi dovranno eseguirla.

B.7.4 Fault tolerance

Quando il programmatore definisce il grafo di computazione, ha la possibilità di associare ad esso un livello di *fault tolerance*: se viene specificato un livello di fault tolerance pari a k, Paralex genera l'opportuno codice in modo che ogni nodo del grafo venga replicato su $k+1$ nodi distinti. Quindi, se vi sono fino a k fallimenti, il successo della computazione è comunque garantito.

Anche per la gestione del fault tolerance viene usato il toolkit ISIS.

B.7.5 Esecuzione

Prima che un programma possa essere eseguito, ogni nodo del grafo di computazione (ed eventualmente anche le sue repliche, se viene utilizzata la funzionalità di fault tolerance) deve essere associato ad un host del sistema distribuito considerato.

In questo problema di mapping, le unità elementari considerate non sono i singoli nodi, ma *catene*: una catena è definita come una sequenza di nodi che devono essere eseguiti sequenzialmente. Questi nodi devono essere "compatibili" tra loro, ovvero deve essere possibile eseguire il codice ad essi associato su uno stesso calcolatore.

Associare un'intera catena ad un nodo permette di ridurre l'overhead di comunicazione: in questo modo, infatti, il "traffico" tra i nodi che costituiscono la catena viene mantenuto locale.

L'algoritmo di mapping utilizzato da Paralex è di tipo euristico, in cui si considera un load balancing di tipo dinamico: periodicamente viene controllato il load di ogni macchina del sistema distribuito (anche il load non dovuto a job Paralex) e, se necessario, viene rivisto il mapping tra nodi e processori inizialmente adottato.

B.7.6 Tracing

Paralex permette all'utente di visualizzare lo stato di esecuzione di un'applicazione. È possibile controllare lo stato dei vari nodi, il nome dei calcolatori in cui essi sono in esecuzione, il flusso dei dati attraverso gli archi del grafo di computazione, ecc...

B.8 Meander

Meander [59] [60] è un insieme di tool integrati che permettono di gestire le varie fasi dello sviluppo di un'applicazione parallela:

- definizione del programma
- codifica
- compilazione
- esecuzione
- debugging

Meander può essere utilizzato per un sistema *transputer*, o per un ambiente costituito da un insieme omogeneo di workstation.

In figura B.19 è mostrato uno schema logico che descrive l'architettura del sistema Meander.

B.8.1 Definizione di un'applicazione

Un'applicazione parallela viene definita attraverso il cosiddetto *specification graph*: un grafo diretto costituito da *graph fragment*, connessi tra loro attraverso archi.

Un graph fragment è un sottografo, inserito nel grafo attraverso al più un arco entrante e un arco uscente di tipo *causal*. Gli archi di tipo causal definiscono il flusso sequenziale dei dati e delle informazioni di controllo.

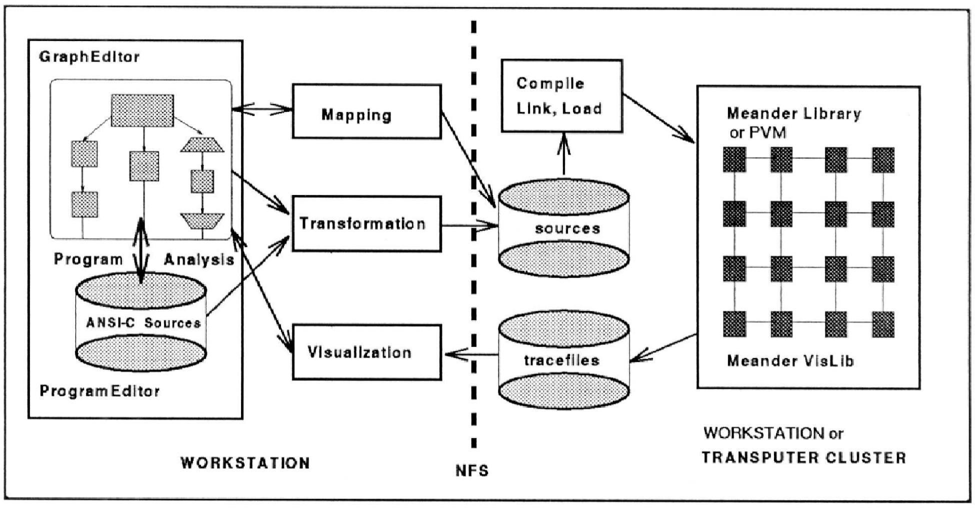

Figura B.19: Overview del sistema Meander

Il parallelismo dell'applicazione viene definito attraverso i nodi *create-child* e *wait-child*. Tra questi due nodi vengono inseriti, attraverso archi di tipo causal, una serie di graph fragment, che devono essere eseguiti in parallelo.

Ogni nodo dello specification graph rappresenta un processo sequenziale: ad esso viene associato una porzione di codice, scritta in linguaggio C.

Attraverso Meander è poi possibile definire altri costrutti, quali il costrutto di comunicazione, rappresentato attraverso i nodi *send* e *receive*, l'operazione condizionale, attraverso il costrutto *alt*, e l'iterazione, rappresentata dal costrutto *do*.

In figura B.20 è mostrato l'esempio di uno specification graph, realizzato attraverso l'apposito tool grafico di Meander.

B.8.2 Compilazione

Una volta che lo specification graph è stato definito, il codice scritto dal programmatore viene "integrato" con le chiamate alle apposite procedure che si occupano della gestione dei processi e della comunicazione. Nel caso di un ambiente costituito da una rete di workstation viene utilizzata la libreria PVM, mentre nel caso di sistemi transputer viene utilizzata un'altra specifica libreria.

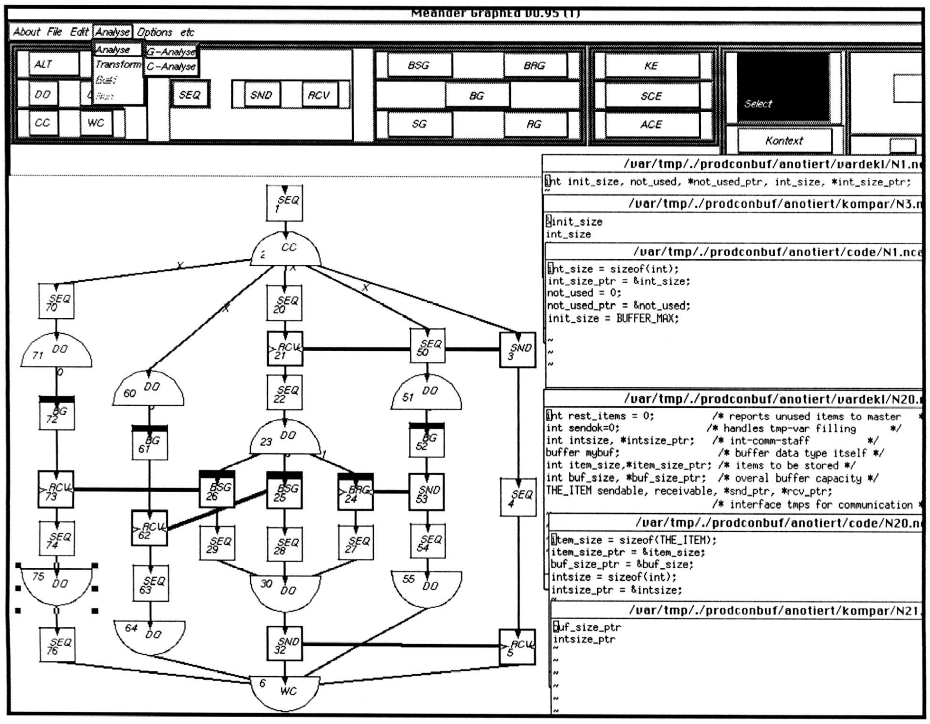

Figura B.20: Esempio di specification graph

B.8.3 Esecuzione

Per eseguire il programma parallelo è necessario associare i vari processi ai processori disponibili.

Per far questo, prima di tutto lo specification graph viene trasformato in un altro grafo, chiamato *process graph*: ogni nodo di questo grafo rappresenta un processo, mentre gli archi definiscono le comunicazioni tra questi processi. I vari nodi del process graph possono essere "annotati" con ulteriori parametri, che descrivono i requisiti necessari per l'esecuzione del codice associato (memoria richiesta, la necessità di particolari configurazioni hardware, ecc..)

Viene poi costruito un altro grafo, chiamato *hardware graph*, i cui nodi rappresentano i processori del sistema considerato, mentre gli archi definiscono le connessioni di rete che esistono tra essi. L'hardware graph può essere "annotato" dal programmatore con vari parametri, che descrivono le caratteristiche delle varie componenti (prestazioni dei processori, memoria presente, interfacce di rete usate, l'indicazione se i vari computer possono essere utilizzati in "presenza" di altri utenti, ecc...).

In figura B.21 viene mostrato un esempio di definizione di questi attributi per due calcolatori.

Le informazioni associate al process graph e all'hardware graph vengono utilizzate dal sistema, che automaticamente assegna ogni processo al processore più adatto.

B.8.4 Tracing e debugging

Meander permette di visualizzare l'esecuzione del programma in modalità off-line: quando il programma viene eseguito, viene prodotto un *tracefile*, dove vengono memorizzate le informazioni più significative. Il tracefile può essere poi utilizzato per simulare la ri-esecuzione del programma, che consente di esaminare lo stato dei vari nodi, il numero e la dimensione dei messaggi, il tempo impiegato da ogni processo per la computazione e per l'attesa di dati, ecc...

In figura B.22 è mostrato un esempio di visualizzazione dell'esecuzione di un programma.

Il tracing può essere uno strumento molto utile anche per il debugging e per il tuning dell'applicazione.

B.9 Considerazioni sugli ambienti esaminati

In questo paragrafo vengono valutate le caratteristiche dei vari ambienti considerati, in relazione con i requisiti richiesti per il sistema da realizzare.

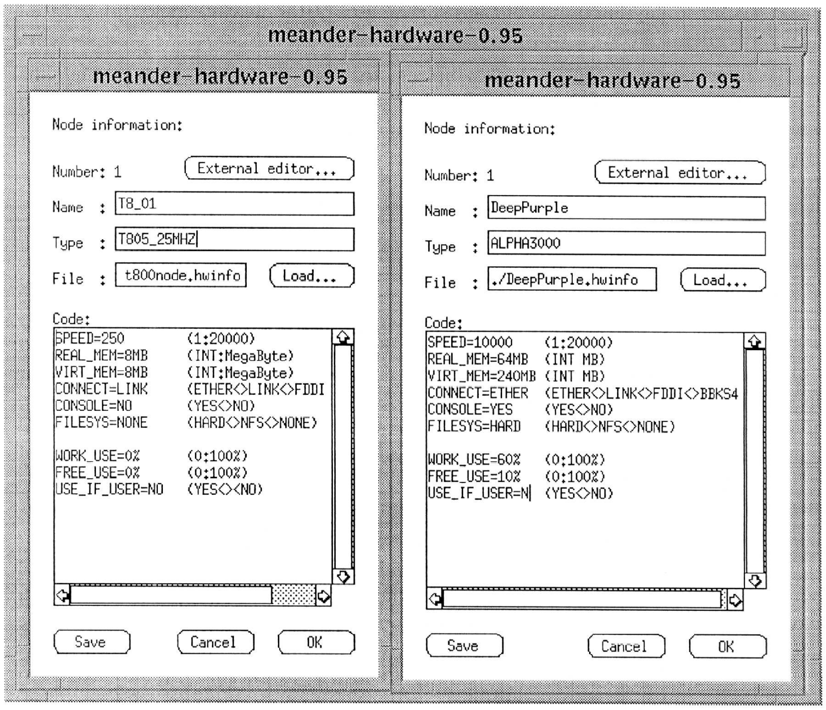

Figura B.21: Esempio di attributi di due nodi dell'hardware graph

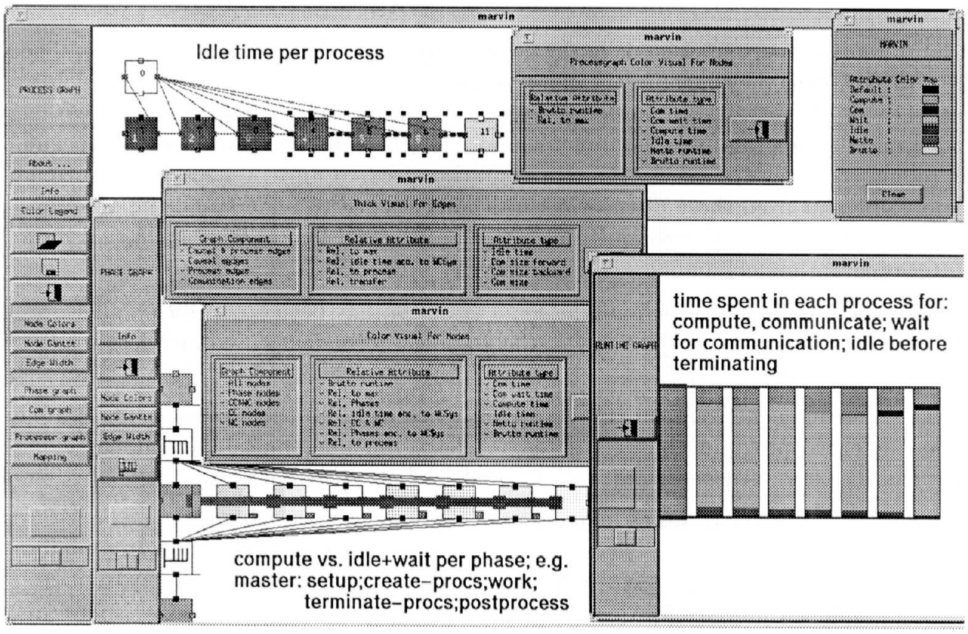

Figura B.22: Esempio di visualizzazione dell'esecuzione di un programma con Meander

L'analisi viene fatta esaminando singolarmente le diverse fasi della gestione dell'applicazione parallela, valutando, per ognuna di esse, le funzionalità disponibili nei diversi sistemi.

B.9.1 Ambiente di esecuzione

Come detto nel cap. 3, l'ambiente da considerare è una Wide Area Network, costituita da diversi siti connessi tra loro. Ogni sito rappresenta un metasistema, costituito da calcolatori collegati tra loro attraverso una rete locale di comunicazione.

Tutti gli ambienti esaminati, invece, fanno riferimento ad un ambiente di tipo LAN: non è contemplato il caso in cui i vari calcolatori sono posti in un ambiente geograficamente distribuito.

Per quel che riguarda le architetture dei calcolatori, è necessario considerare un ambiente eterogeneo costituito da workstation Unix e PC Windows NT.

La maggior parte degli ambienti analizzati (HeNCE, VPE, Enterprise, Paralex) possono essere utilizzati in un ambiente eterogeno di workstation Unix. Meander, invece, considera come ambiente di esecuzione un insieme omogeneo di calcolatori Unix. Infine in CODE l'applicazione può essere eseguita su una singola architettura MIMD. Allo stato attuale non esiste nessuna implementazione degli ambienti considerati per la piattaforma Windows NT.

B.9.2 Definizione dell'applicazione

Per quel che riguarda la definizione dell'applicazione, nel sistema da considerare deve essere possibile specificare le componenti sequenziali, e i vincoli di precedenza data-driven che esistono tra esse (un processo può iniziare l'esecuzione quando tutti i relativi input sono disponibili).

Questo è possibile in tutti gli ambienti esaminati, anche se con modalità diverse. L'unica limitazione è presente in HeNCE, in cui non si possono definire relazioni di precedenza tra nodi discendenti da nodi fratelli.

Nel sistema da realizzare i vari moduli dell'applicazione sono indipendenti tra loro. Non c'è necessità di interazione durante la loro esecuzione, e non è necessario prevedere l'uso di variabili globali, condivise tra le varie componenti sequenziali. Ogni modulo può avere zero o più input, e uno o più output. Ogni modulo è rappresentato da un programma binario eseguibile, o da uno script che definisce alcune variabili logiche, e che poi invoca un eseguibile.

Negli ambienti considerati, invece, ogni modulo sequenziale deve essere rappresentato attraverso una procedura scritta in linguaggio C (CODE, Enterprise, Meander), o a scelta in C o Fortran (HeNCE, VPE). Paralex invece consente di rappresentare ciascun modulo attraverso una procedura scritta in un linguaggio "compatibile" con il linguaggio C, o attraverso un programma binario eseguibile. In Paralex, però, ciascun modulo deve rappresentare una funzione con un solo output; inoltre tale funzione deve essere "pura", ossia essa non può manipolare nessun oggetto esterno (file, device, ecc...), cosa che invece deve essere possibile nel sistema DEMI da realizzare.

In nessuno degli ambienti esaminati viene considerato il problema della gestione dei dati: la loro archiviazione, il loro tipo, l'indicazione se i risultati intermedi devono essere salvati o se devono essere cancellati, ecc... In DEMI, invece, la gestione dei dati è un aspetto molto importante del problema.

B.9.3 Esecuzione del programma

Una volta che l'applicazione è stata definita e compilata, i vari moduli devono essere assegnati ai vari calcolatori per la loro esecuzione. L'obiettivo primario di questo problema di mapping è la massimizzazione del throughput.

I vari ambienti esaminati gestiscono questo problema di mapping in diversi modi: in alcuni sistemi (ad esempio VPE) il programmatore deve specificare il calcolatore o il tipo di calcolatore che deve eseguire il modulo (nel secondo caso per l'esecuzione viene scelto semplicemente il primo computer "libero" di quel tipo), in altri (come HeNCE) il programmatore deve specificare il "costo" dell'esecuzione dei vari moduli sui vari calcolatori (in questo caso il mapping viene effettuato in modo da minimizzare il costo totale di esecuzione). Paralex usa un algoritmo più sofisticato di load balancing dinamico, in cui, a differenza degli altri ambienti, si considera anche il load dovuto a job di utenti generici. Paralex, inoltre, consente di minimizzare l'overhead di comunicazione, associando ai vari processori catene di task, anziché singoli task.

Nel sistema DEMI, in presenza di utenti generici, in generale l'esecuzione dei moduli dell'applicazione parallela deve essere sospesa: gli utenti devono avere più alta priorità nell'utilizzo delle risorse di calcolo. Tra gli ambienti considerati solo Meander considera questo aspetto.

Altre funzionalità aggiuntive implementate in alcuni sistemi (come la gestione del fault tolerance prevista in Paralex) non sono richieste per il sistema da realizzare.

In nessuno degli ambienti esaminati sono implementate funzionalità quali il *checkpointing* e il *process migration*, che possono essere utili strumenti per realizzare la massimizzazione del throughput. Utilizzando il meccanismo del checkpointing, infatti, si ha la garanzia che un job viene sicuramente completato: anche se, ad esempio, vi è il crash di un sistema che sta eseguendo un job, l'elaborazione eseguita non viene comunque "persa". Il meccanismo di fault tolerance implementato nel sistema Paralex (in cui l'esecuzione del job viene replicata su più processori), invece, non consente di massimizzare il throughput (anzi, in generale, usando questa tecnica il throughput diminuisce).

La migrazione dei processi tra diversi computer della stessa architettura (in cui si utilizza il meccanismo del checkpointing), è un utile strumento per gestire i casi in cui utenti generici vogliono avere un utilizzo esclusivo delle loro workstation: quando un utente generico "reclama" l'uso di una workstation, i job in esecuzione su questo calcolatore possono essere migrati su un altro disponibile.

B.9.4 Monitor dell'esecuzione

Quasi tutti gli ambienti considerati (tranne CODE e VPE) prevedono una funzionalità che consente al programmatore di visualizzare lo stato di esecuzione del programma. Alcuni di essi (come Enterprise e Meander), consentono anche un'analisi off-line, per cui si può simulare la ri-esecuzione del programma, definendo dei breakpoint per controllare il valore delle variabili.

Nel sistema da realizzare è richiesto un supporto molto limitato per questo aspetto: è sufficiente poter verificare su che computer i vari moduli sono in esecuzione.

B.9.5 Conclusioni

Non esiste, quindi, un ambiente che presenti tutti i requisiti e tutte le funzionalità richieste. In alcuni sistemi sono implementate certe funzionalità, ma non ne sono presenti altre, indispensabili per il sistema da realizzare.

Altre funzionalità (ad esempio il supporto per l'architettura Windows Nt, la possibilità di gestire i dati, ecc...), invece, non sono implementate in nessuno degli ambienti esaminati.