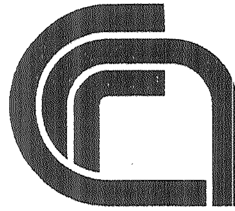


Consiglio Nazionale delle Ricerche

# Heterogeneous Multiphases Mapping: un Algoritmo di Mapping per Metacomputer

Ranieri Baraglia, Domenico Laforenza
Antonio Panciatici, Fabio Ravaglia

Rapporto
CNUCE-B4-1998-01

CNUCE

Pisa

# Heterogeneous Multiphases Mapping: un Algoritmo di Mapping per Metacomputer

R. Baraglia, D. Laforenza, A. Panciatici, F. Ravaglia.
CNUCE - Istituto del Consiglio Nazionale delle Ricerche
via S. Maria, 36 56100 Pisa (Italy) Tel. 050-593111 - Fax 050-904052
e-mail: (R.Baraglia, D.Laforenza)@cnuce.cnr.it

February 2, 1998

**Abstract**

In questo rapporto tecnico viene descritto l'algoritmo *Heterogeneous Multiphases Mapping* (*HMM*). HMM effettua il mapping di un applicazione parallela su un sistema eterogeneo distribuito. L'applicazione viene rappresenta mediante due grafi diretti aciclici a cui sono associate informazioni quali: costi computazionali e di comunicazione, disponibilitá di risorse uniche e presenza di costrutti paralleli. Il sistema viene rappresentato attraverso un grafo simmetrico con pesi non uniformi sui nodi, che rappresentano i singolo elaboratori, e sugli archi, che rappresentano le connessione esistenti tra gli elaboratori. HMM, utilizzando la tecnica di ricerca locale, ottiene un mapping subottimo dell'applicazione parallela sul sistema eterogeneo distribuito. I test svolti dimostrano come HMM ottenga risultati migliori o simili ad altri algoritmi presenti in letteratura trovando in alcuni casi la soluzione ottima.

## 1 Introduzione

Esistono in ambito scientifico classi di problemi la cui risoluzione richiede da un punto di vista elaborativo, l'utilizzo di differenti modelli computazionali (SIMD, MIMD, sequenziale, ecc.). L'esigenza di fornire un adeguato supporto di calcolo per le applicazioni parallele costruite per tali problemi, ha portato alla soluzione detta Heterogeneous Computing (HC), ovvero all'utilizzo di differenti architetture, connesse tramite reti di comunicazione ad alta velocitá, come un singolo sistema di calcolo eterogeneo distribuito [12, 15].

Al fine di ottenere un efficiente sfruttamento di un sistema eterogeneo è necessaria la generazione di un appropriato mapping dei task costituenti l'applicazione

parallela sui processori del sistema. Il raggiungimento di questo obiettivo presenta notevoli difficoltà in quanto come è noto dalla letteratura [1, 4, 14] il problema del mapping è NP-completo. Infatti, non sono conosciute, se esistono, strategie aventi complessità polinomiale sulla lunghezza dell'input, che permettono di trovare la soluzione ottima per tale problema.

Gli algoritmi di mapping possono essere divisi in due classi: statici e dinamici. Nei primi l'allocazione dei task è decisa prima dell'esecuzione dell'applicazione. Nei secondi i task vengono allocati sui processori a run-time durante l'esecuzione dell'applicazione stessa.

Gli algoritmi statici hanno il vantaggio di poter utilizzare tecniche piuttosto complesse, in quanto il costo computazionale dovuto al loro utilizzo non aumenta il tempo di completamento dell'applicazione. Citiamo alcuni degli algoritmi presenti in letteratura e le tecniche da loro utilizzate: greedy, ricerca locale, branching [10, 19, 16, 17], genetiche [24, 21, 22], clustering [9].

La maggior parte di questi algoritmi modella l'applicazione parallela mediante un grafo diretto aciclico in cui ogni nodo rappresenta un task e ogni arco rappresenta la comunicazione tra due nodi. Ad ogni nodo è associato un valore rappresentante il costo computazionale del task corrispondente, ed ad ogni arco è associato un valore rappresentante il costo di comunicazione esistente tra i due nodi uniti dall'arco stesso. Generalmente sia i pesi sui nodi che quelli sugli archi sono diversi. Il sistema eterogeneo distribuito viene rappresentato, invece, mediante un grafo simmetrico con pesi diversi sugli archi e sui nodi. In questo caso un nodo rappresenta un processore del sistema e un arco il link esistente tra due processori. Il peso associato ad un nodo rappresenta la sua velocitá elaborativa mentre il peso associato ad un arco rappresenta la velocitá del corrispondente link.

L'algoritmo di mapping HMM da noi proposto permette di ottenere un mapping sub-ottimo di un'applicazione parallela su un metacomputer. Esso utilizza la tecnica di *ricerca locale* combinata con una tecnica derivata da quella di *Simulated Annealing*. Le considerazioni che ci hanno condotto verso questa soluzione sono:

- i risultati ottenibili utilizzando tecniche di branching dipendono strettamente dalla qualità della funzione di valutazione impiegata, che, per questo tipo di problema, è di difficile individuazione. Ci è sembrato soprattutto difficile determinare una buona funzione di stima che permetta di valutare per eccesso il costo della migliore soluzione, ottenibile a partire dalla soluzione parziale in analisi;
- i risultati ottenibili utilizzando algoritmi genetici ci sono sembrati strettamente correlati alle funzioni di probabilità utilizzate, per cui, sia a causa della difficoltà di trovare buone funzioni di probabilità, che a causa della incertezza (anche in previsione della fase di test) sulla bontà di tali funzioni, si è ritenuto opportuno scartare questa soluzione;

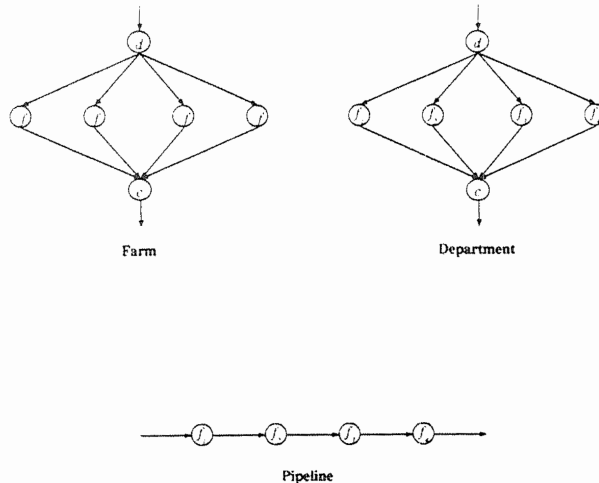



Figure 1: Struttura di alcuni costrutti di parallelismo.

- ci sono, invece, sembrate più interessanti le tecniche di clustering proposte in [5, 8], in quanto, partendo da presupposti più conformi ad un ambiente reale, riescono generalmente ad ottenere mapping migliori rispetto alle soluzioni descritte nei due punti precedenti;

- l'implementazione di applicazioni parallele può essere fatta utilizzando forme di parallelismo (*farm*, *pipeline*, *geometric*, ecc. (Fig. 1) [11, 13]) che possono essere applicate adottando modelli di programmazione sia ristretti che non ristretti [23, 6, 3]. Attualmente sono disponibili linguaggi paralleli quale, ad esempio P3L [2], e ambienti visuali di programmazione quali, ad esempio Enterprise [18], che si basano sull'uso di costrutti paralleli per l'implementazione di applicazioni parallele secondo una metodologia di programmazione strutturata facilitando il compito del programmatore.

  Usando le forme di parallelismo per l'implementazione del programma parallelo esso è ben rappresentabile mediante un grafo diretto aciclico.

  In esso le forme di parallelismo costituenti l'applicazione possono essere trattate come insiemi di cluster gerarchici in cui ciascun cluster raggruppa i moduli di un costrutto parallelo. Le forme di parallelismo esprimono computazioni parallele in cui i pattern di comunicazione sono strutturati e ben definiti e all'interno delle quali, generalmente, è concentrata una cospicua parte della comunicazione. Nell'ottica di ridurre i tempi di comunicazione, solitamente, risulta conveniente allocare una forma di parallelismo su un

singolo elaboratore parallelo.

In questo modo è possibile allocare i moduli dell'applicazione operando in più fasi e considerando in ogni fase un livello di clusterizzazione a partire da un livello in cui ciascun cluster rappresenta una forma di parallelismo fino ad arrivare all'ultimo livello in cui ciascun cluster è costituito da un singolo modulo. In tal modo è possibile trattare anche implementazioni di forme di parallelismo che al loro interno sfruttano altre forme di parallelismo.

Analizzando tutti i livelli di clustering è possibile individuare la soluzione che inizialmente dà il migliore mapping nell'ottica di ridurre il costo delle comunicazioni ed il tempo di CPU dell'applicazione. La soluzione trovata può essere ulteriormente elaborata tramite tecniche di ottimizzazione nell'obiettivo di trovare mapping più efficienti.

L'algoritmo HMM operando come precedentemente descritto, permette di effettuare dei mapping di DAG che rappresentano applicazioni parallele ottenute usando modelli di programmazione ristretti e non ristretti.

La restante parte dell'articolo e organizzata come segue. Nella sezione 2 e 3, sono descritti i grafi dell'applicazione e del sistema distribuito eterogeneo. Nella sezione 4 é descritto il funzionamento dell'algoritmo HMM. Nella sezione 5 sono presentati i risulati ottenuti confrontando HMM con altri algoritmi presenti in letteratura. Infine nella sezione 6 sono riportate le conclusioni.

## 2 Rappresentazione delle Applicazioni e dei Metacomputer

Secondo la metodologia sviluppata, l'applicazione parallela è rappresentata con due DAG indicati con $G_N$ e $G_A$ e il metacomputer è rappresentato da un grafo simmetrico $G_M$.

Il grafo $G_N$ dà una rappresentazione gerarchica dell'applicazione (Fig. 2) mettendo in evidenza i vincoli di precedenza esistenti tra i costrutti paralleli costituenti l'applicazione e rappresentati dai nodi $n_i$ di $G_N$ con $n_i \in \{n_1, n_2, .., n_{N_{G_N}}\}$. Il parallelismo è espresso sia a livello dei nodi $n_i$ sia al loro interno.

Come mostra la Figura 3 ogni nodo $n_i$ di $G_N$ è a sua volta strutturato in modo gerarchico in più *stadi*. I nodi di $G_N$ possono essere *gerarchici* o *atomici*. I nodi atomici rappresentano i singoli task atomici dell'applicazione. I nodi gerarchici appartenenti ad uno stadio $S_v$ di $n_i$ riflettono l'ordine secondo il quale sono stati usati i costrutti paralleli. Questo tipo di incapsulamento ricorsivo termina all'ultimo stadio rappresentato da un DAG in cui i nodi rappresentano tutti i task atomici. Quindi un nodo gerarchico è un nodo che rappresenta un modulo parallelo dell'applicazione che è composto da più nodi atomici (task atomici) comunicanti tra loro secondo una o più costrutti paralleli anch'essi organizzati in modo gerarchico. Il modulo parallelo rappresentato dal nodo in Figura 3, ad

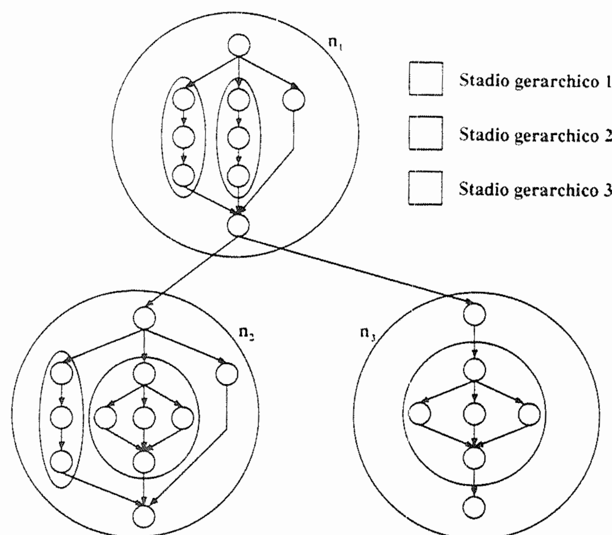



Figure 2: Esempio di DAG $G_N$ rappresentante l'applicazione.

esempio, è visto: allo stadio uno come un unico nodo gerarchico; allo stadio due composto da tre nodi atomici, da un nodo gerarchico rappresentante un task parallelo implementato usando un costrutto farm e da un nodo gerarchico rappresentante un task parallelo implementato con un costrutto pipeline; allo stadio tre composto da undici nodi atomici rappresentanti tutti task sequenziali.

Il grafo $G_A$ fornisce una rappresentazione dell'applicazione (Fig. 4) in cui i nodi in esso contenuti rappresentano task atomici. Il numero di nodi di $G_A$ è dato dalla quantità $N_{G_A} = \sum_{i=1}^{N_{G_N}} N_{n_i}$ dove $N_{n_i}$ rappresenta il numero di task atomici appartenenti al generico nodo $n_i \in G_N$. $N_{n_i}$ è un intero positivo maggiore o uguale ad uno, in quanto, come evidenziato dalla Figura 2, ogni $n_i \in G_N$ è costituito da un sottografo rappresentante qualche costrutto parallelo oppure da un nodo atomico. Per questo motivo, in generale, un nodo $n_i$ del grafo $G_N$ potrà essere espanso, secondo la gerarchia in esso presente, fino ad ottenere i suoi $N_{n_i}$ task atomici.

Poiché il costrutto *loop* è rappresentato mediante un grafo ciclico, si è ritenuto necessario descivere tale costrutto, nel grafo dell'applicazione, come un nodo atomico al quale è associata una stima del suo peso computazionale.

Le informazioni associate ad ogni nodo gerarchico a qualsiasi stadio sono:

- ambiente software richiesto per la sua esecuzione (compilatori, S.O., librerie, ecc.);

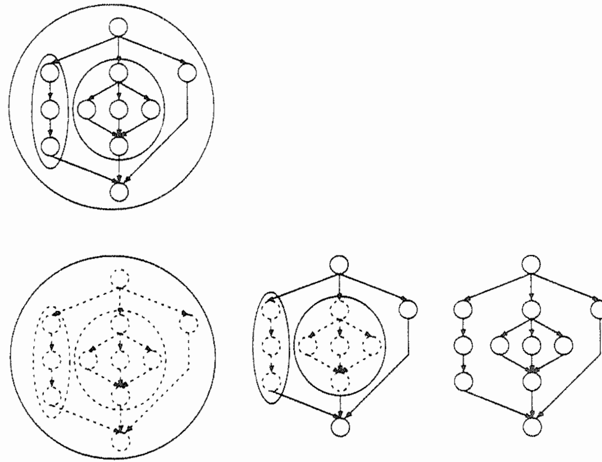
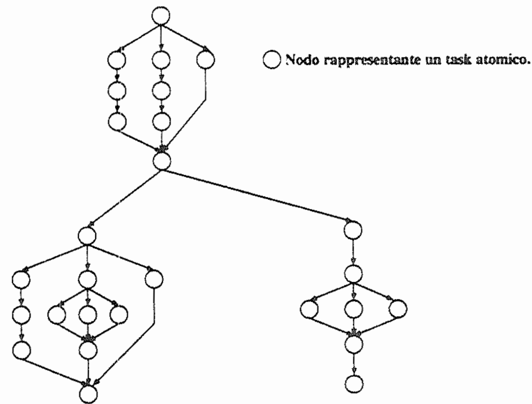


Figure 3: Esempio di stadi di un nodo del grafo $G_N$.




A questo livello i paradigmi paralleli non hanno significato. L'applicazione viene vista come un insieme di task atomici eseguiti in parallelo secondo le loro relazioni di precedenza.

Figure 4: Esempio di DAG $G_A$ rappresentante l'applicazione parallela

- costo computazionale, ottenuto mediante la somma dei costi computazionali dei nodi in esso contenuti;
- numero di processori necessari per l'esecuzione parallela dei nodi atomici in esso contenuti;
- quantità di memoria necessaria per l'esecuzione, ottenuta sommando le richieste dei singoli nodi in esso contenuti;
- modello computazionale (SIMD, MIMD, sequenziale, ecc.) ad esso associato;
- quantità di comunicazione, calcolata come somma dei dati scambiati all'interno di $n_i$.

Le informazioni associate ad ogni nodo atomico sono:

- ambiente software necessario per la sua esecuzione;
- costo computazionale;
- modello computazionale usato (SIMD, sequenziale, ecc.);
- quantità di memoria necessaria per la sua esecuzione;

L'unica informazione associata ad ogni arco orientato di $G_A$ e di $G_N$ è la quantità di dati che un nodo padre, di qualunque tipo esso sia (atomico o gerarchico), invia al nodo figlio (atomico o gerarchico).

Il grafo $G_M$ (Fig. 5) è simmetrico e totalmente connesso. Ognuno dei suoi $N_{G_M}$ nodi rappresenta un elaboratore (parallelo o sequenziale) ed è indicato con $E_m$ con $1 \leq m \leq N_{G_M}$.
Le informazioni associate ad ogni nodo di $G_M$ sono:

- classe architetturale (sequenziale, SIMD, MIMD, ecc.);
- ambiente software per l'esecuzione di applicazioni sequenziali o parallele;
- numero di processori;
- quantità di memoria disponibile per l'esecuzione dei programmi. Quando l'elaboratore è shared memory si considera tutta la memoria disponibile. Quando l'elaboratore è a memoria distribuita si considera la memoria associata ad ogni suo processore;
- caratteristiche di trasmissione del sistema di comunicazione (banda e latenza);
- potenza computazionale globale dell'elaboratore e di ogni suo singolo processore.

L'unica informazione associata ad ogni arco di $G_M$ è la banda di comunicazione tra gli elaboratori.

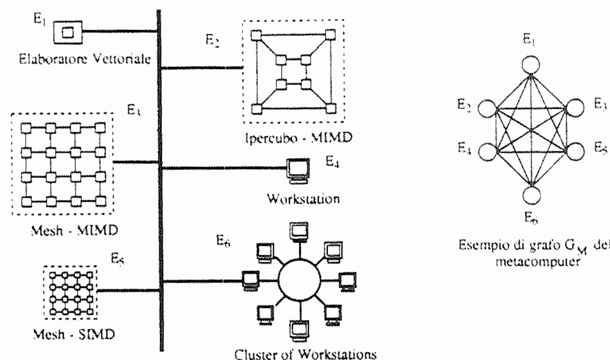



Figure 5: Esempio di un metacomputer e sua rappresentazione mediante il grafo $G_M$.

# 3 Struttura dell'Algoritmo

La Figura 6 mostra lo pseudo codice dell'algoritmo HMM. Per rispettare i vincoli di precedenza tra i nodi del grafo dell'applicazione $G_N$, quest'ultimo viene suddiviso in *fasi* $F_h$ (Fig. 7). Così facendo i nodi appartenenti alla fase $F_1$ saranno tutti quelli che non hanno padri, i nodi appartenenti alla fase $F_h$ quelli che hanno almeno un padre tra i nodi della fase $F_{h-1}$ e nessun padre nelle fasi successive o in $F_h$. In questo modo il numero di fasi in cui viene suddiviso $G_N$ corrisponde alla profondità di $G_N$ stesso. La scelta di suddividere il grafo $G_N$ in più fasi è giustificata dal fatto che, supponendo l'applicazione sia implementata usando forme di parallelismo, il grafo rappresentante l'applicazione sarà costituito da sottografi gerarchici (ad esempio un grafo associato ad un pipeline può avere come nodi worker dei sottografi associati ad un farm) contenuti a loro volta nei nodi $n_i$ di $G_N$. Struttuando $G_N$ in questo modo, le relazioni di precedenza tra i suoi nodi riflettono l'organizzazione sequenziale delle fasi costituenti l'applicazione.

Un generico nodo di $n_i \in F_h$ allo stadio $S_v$, viene indicato con $\tilde{n}_{n_i}^{F_h(S_v)}$; in particolare, allo stadio uno, il nodo $n_i \in F_h$ coinciderà con $\tilde{n}_{n_i}^{F_h(S_1)}$. L'insieme dei nodi appartenenti al $v$-esimo stadio del nodo $\tilde{n}_{n_i}^{F_h(S_1)} \in F_h$ è indicato con $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$.

Per conoscere in un dato istante il numero di processori disponibili su ogni elaboratore $E_m$ del metacomputer, viene associato ad ognuno di essi, un vettore $V_{E_m}$ di dimensione pari al suo numero di processori. L'$u$-esima entrata del vettore $V_{E_m}$, cioè $V_{E_m}[u]$, contiene un valore rappresentante il tempo in cui l'$u$-esimo processore $p_{E_m}^{(u)}$ di $E_m$ sarà disponibile per eseguire un nuovo lavoro.

```
Algoritmo HMM (G_N, G_A, G_M)
  begin
    dividi_in_fasi (G_N)
    while (ci sono fasi da analizzare) do
       begin
         ordina_nodi (nodi della fase corrente)
         while (non sono stati analizzati tutti
              i nodi della fase corrente) do
            begin
              if (esiste una macchina per allocare il nodo n_i
                in modo atomico) then
                cerca_migliore_allocazione (n_i)
              else
                marca il nodo come non allocato
            end
         while (ci sono nodi non allocati) do
            cerca_migliore_allocazione (n_i)
    end
    Sol_best = Sol_corr (Sol_corr = soluzione iniziale)
    while (numero totale di iterazioni < T_max and
          numero totale di iterazioni senza
          miglioramenti < P_max) do
       begin
         cerca_cammino_critico (Sol_corr)
         while (ci sono nodi n_i del cammino critico non analizzati) do
            begin
              cerca_macchina_migliore (n_i)
              calcola e memorizza la nuova soluzione Sol_new
            end
         scegli soluzione migliore Sol_new nell'intorno di Sol_corr
         if (Sol_new > Sol_corr and
           numero di iterazioni < P_max) then
           Sol_corr = Sol_new
         else
           numero iterazioni fatte senza miglioramenti = P_max
         if (Sol_new < Sol_corr) then
           begin
             Sol_corr = Sol_new
             if (Sol_new < Sol_best) then
               Sol_best = Sol_new
           end
       end
       return Sol_best
    end
```

Figure 6: Pseudo-codice dell'algoritmo $HMM$.

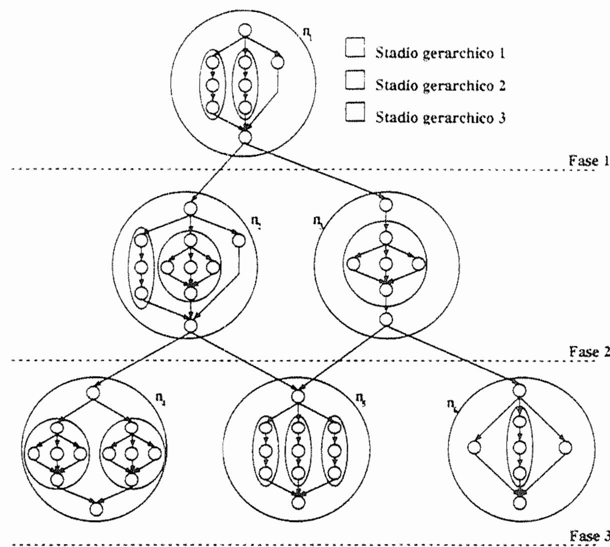



Figure 7: Suddivisione del grafo del $G_N$ dell'applicazione in fasi.

Inizialmente, per ogni elaboratore $E_m$, il vettore $V_{E_m}$ avrà tutte le componenti nulle, in quanto, il metacomputer, viene considerato totalmente a disposizione dell'applicazione. Durante l'esecuzione dell'algoritmo HMM ogni vettore $V_{E_m}$ verrà aggiornato in base al carico computazionale dovuto all'applicazione trattata e associato allo stesso $E_m$. L'algoritmo HMM non considera il carico presente sugli elaboratori del metacomputer dovuto ad applicazioni differenti da quella trattata. Ciò è dovuto alle seguenti considerazioni.

- Essendo HMM un algoritmo statico, prendere decisioni di mapping in base ai valori di carico presenti in un certo istante può risultare poco significativo, perché l'allocazione dei task fatta da HMM, può avvenire con valori di carico anche assai diversi da quelli precedentemente rilevati.
- Sarebbe utile avere delle informazioni sulla distribuzione e sulla varianza del carico sugli elaboratori in un periodo medio lungo. In questo modo si potrebbe stabilire, per ogni elaboratore, un valore di massimo carico sotto al quale poter assegnare nuovi processi. Utilizzando delle informazioni sul carico abbastanza precise, HMM, per ciascuna allocazione processo-processore potrebbe selezionare l'elaboratore meno carico e quindi ridurre ulteriormente il tempo di completamento dell'applicazione. Comunque, tener conto del carico comporta un non indifferente aumento della complessità del problema.

Vedere il grafo $G_N$ strutturato logicamente in fasi e stadi, permette di semplificare il problema e quindi di effettuare delle scelte di mapping vantaggiose.

La semplificazione del problema del mapping è dovuta al fatto che, l'algoritmo di ricerca locale su cui si basa HMM, determina l'assegnamento dei nodi del grafo dell'applicazione sugli elaboratori del metacomputer operando separatamente sulle singole fasi del grafo $G_N$. Quindi, si scompone il problema in sottoproblemi di dimensione minore che sono più facilmente trattabili. È da osservare che, essendo le fasi $F_h$ di $G_N$ messe in relazione dalle comunicazioni tra i nodi $\tilde{n}_{n_i}^{F_h(S_1)}$ contenuti in ognuna di esse, le scelte effettuate dall'algoritmo di ricerca locale per il mapping relativo alla $h$-esima fase dovranno considerare le informazioni sulle comunicazioni *inter-fase* presenti tra i nodi $\tilde{n}_{n_i}^{F_h(S_v)} \in F_h$ e i nodi $\tilde{n}_{n_j}^{F_k(S_u)} \in F_k$ per $k < h$.

Scelte di mapping vantaggiose sono possibili grazie all'introduzione del concetto di stadio, in quanto, ogni volta che si tenta di allocare un nodo $n_i$, si riesce a individuare (con la strategia di seguito descritta) lo stadio $S_v$ per il quale si effettua il mapping che riduce maggiormente il tempo di completamento del nodo stesso.

Questo risultato si ottiene mediante la seguente strategia: a partire dal nodo $\tilde{n}_{n_i}^{F_h(S_1)}$, si determina l'elaboratore $E_m$ (§3.3) più adatto ad eseguirlo secondo un criterio di scelta che tiene conto delle comunicazioni e dell'affinità codice-elaboratore (§3.1). Si calcola poi il tempo di completamento di $\tilde{n}_{n_i}^{F_h(S_1)}$ su $E_m$ (§3.4). Fatto ciò si espande $\tilde{n}_{n_i}^{F_h(S_1)}$ allo stadio successivo a quello corrente, e per ognuno dei nodi $\tilde{n}_{n_i}^{F_h(S_2)} \in I^{S_2}(\tilde{n}_{n_i}^{F_h(S_1)})$ si cerca la migliore allocazione sugli elaboratori del metacomputer utilizzando il criterio illustrato in precedenza. Si calcola così il tempo di completamento di $I^{S_2}(\tilde{n}_{n_i}^{F_h(S_1)})$ (nodo $n_i$ espanso allo stadio $S_2$) e si confronta con quello calcolato per $\tilde{n}_{n_i}^{F_h(S_1)}$ selezionando come mapping corrente quello che permette ad $\tilde{n}_{n_i}^{F_h(S_1)}$ di terminare prima la sua esecuzione. Questo procedimento viene ripetuto per tutti gli stadi del nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ in modo da poter scegliere il mapping che ottiene il minor tempo di completamento.

## 3.1 Affinità Codice-Elaboratore

La funzione `set_affinita` permette di associare ad ogni nodo $\tilde{n}_{n_i}^{F_h(S_v)}$, di ogni stadio del grafo $G_N$, un vettore affinità utilizzato da HMM per le scelte di mapping.

La funzione `set_affinita` prende in input un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ di $G_N$ ed esegue $N_{G_M}$ iterazioni, una per ogni elaboratore del metacomputer, generando un vettore affinità di dimensione $N_{G_M}$ per ogni $\tilde{n}_{n_i}^{F_h(S_v)}$. L'$i$-esima componente di tale vettore contiene il valore rappresentante il grado di affinità tra il nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ e l'elaboratore $E_i$. Al parametro affinità è associato uno dei due seguenti valori: 1 quando le risorse (software, memoria, processori) presenti sull'elaboratore sono

```
Algoritmo set_affinita
  for each nodo n_i ∈ G_N (gerarchico o atomico) do
    for each Elaboratore E_i del metacomputer do
    begin
      if (memoria richiesta da n_i ≤ memoria presente su E_j and
        software richiesto da n_i ≤ software presente su E_j and
        processori richiesti da n_i ≤ processori disponibili su E_j)
      then
        if (modello computazionale n_i = classe architetturale E_m)
        then
          Aff[n_i,E_i] = 1 else
          Aff[n_i,E_i] = -1
      else
        Aff[n_i,E_i] = -1
    end
```

Figure 8: Pseudo-codice della funzione `set_affinita`.

tali da permettere l'esecuzione del task trattato e il modello computazionale di quest'ultimo coincide con il modello architetturale dell'elaboratore; -1 altrimenti.

La Figura 28 mostra lo pseudo codice della funzione `set_affinita`.

Per calcolare il valore di affinità tra un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ ed un elaboratore $E_i$ del metacomputer, la funzione `set_affinita` esamina i seguenti parametri:

- ambiente software presente sull'elaboratore $E_i$;
- ambiente software necessario per l'esecuzione di $\tilde{n}_{n_i}^{F_h(S_v)}$;
- numero di processori presenti sulla piattaforma di calcolo (MPP, Network di Workstation, ecc.) $E_i$;
- numero di processori richiesti dal nodo $\tilde{n}_{n_i}^{F_h(S_v)}$;
- tipo di codice del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$;
- tipo di architettura di $E_i$;
- memoria disponibile sull'elaboratore $E_i$;
- memoria richiesta dal nodo $\tilde{n}_{n_i}^{F_h(S_v)}$.

La funzione `set_affinita` è composta da due parti principali: il controllo sull'ammissibilità del mapping e il controllo sulla corrispondenza tra il modello computazionale del task e il modello architetturale dell'elaboratore. Quest'ultimo

controllo viene eseguito soltanto nel caso in cui il primo controllo abbia dato esito positivo. Il mapping di un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ su un elaboratore $E_i$ è ammesso se sono verificate le seguenti tre condizioni:

- il numero di processori richiesti da $\tilde{n}_{n_i}^{F_h(S_v)}$ è minore o uguale al numero di processori disponibili su $E_i$;
- l'ambiente software richiesto da $\tilde{n}_{n_i}^{F_h(S_v)}$ è presente su $E_i$;
- la memoria richiesta dal nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ è disponibile sull'elaboratore $E_i$ quando questo è a memoria condivisa. La memoria associata ad almeno un processore di $E_i$, non ancora assegnato ad altri nodi atomici di $\tilde{n}_{n_i}^{F_h(S_v)}$, è sufficiente per eseguire il nodo atomico di $\tilde{n}_{n_i}^{F_h(S_v)}$, quando l'elaboratore è a memoria distribuita. Quest'ultima condizione deve essere soddisfatta per tutti i nodi atomici di $\tilde{n}_{n_i}^{F_h(S_v)}$.

Nel caso in cui anche solo una di tali richieste rimanga inevasa, l'entrata relativa all'elaboratore $E_i$ nel vettore affinità viene posta a -1. Questo significa che il nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ non sarà eseguito sull'elaboratore $E_i$ e che `set_affinita` inizierà a valutare l'affinità tra $\tilde{n}_{n_i}^{F_h(S_v)}$ e l'elaboratore successivo ad $E_i$. Se invece le condizioni relative al controllo sull'ammissibilità del mapping sono tutte soddisfatte, `set_affinita` verifica la corrispondenza tra il modello computazionale del task e il modello architetturale dell'elaboratore. Se anche quest'ultimo controllo è soddifatto il paramento affinità viene posto ad 1; in caso contrario il paramento affinità viene posto a -1. La funzione `set_affinita` è organizzata in due parti distinte perché, tra i possibili sviluppi futuri di HMM, è prevista la realizzazione di uno strumento che, analizzando i risultati ottenuti dall'esecuzione di benchmark, produca valori del parametro affinità più precisi rispetto agli elaboratori disponibili.

## 3.2 Ricerca della Soluzione Iniziale

L'algoritmo HMM individua la soluzione iniziale analizzando ognuna delle $N_F$ fasi $F_h$ in cui è stato suddiviso il grafo $G_N$ dell'applicazione. A partire dalla fase uno, l'algoritmo HMM, elabora nell'ordine tutte le $N_F$ fasi di $G_N$. Per ognuna di esse trova una soluzione parziale iniziale, in base alla quale tutti i nodi atomici, contenuti nei nodi appartenenti alla fase $F_h$ in analisi, vengono allocati sui processori degli elaboratori del metacomputer. Per rendere più chiara la strategia adottata nella ricerca della soluzione iniziale, è importante sottolineare che una volta assegnati tutti i task di una fase sui processori di un elaboratore, questi task non vengono rimossi per tutta la durata della procedura di ricerca della soluzione iniziale. Di conseguenza, le scelte di mapping per i task di una generica fase $F_h$, saranno prese tenendo in considerazione i mapping già eseguiti per la fase $F_k$ con $k < h$.

La soluzione iniziale parziale ottenuta dall'analisi della fase $F_h$ viene determinata mediante i seguenti passi.

**passo 1** Si ordinano in modo non crescente rispetto al loro carico computazionale tutti i nodi $\tilde{n}_{n_i}^{F_h(S_1)}$ (appartenenti alla fase $F_h$) del grafo $G_N$.

**passo 2** Si seleziona, in base all'ordinamento fatto al passo uno, il primo nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ non ancora analizzato. Si controlla poi se è presente nel metacomputer almeno un elaboratore in grado di eseguire $\tilde{n}_{n_i}^{F_h(S_1)}$. Questo significa che nel vettore affinità associato a $\tilde{n}_{n_i}^{F_h(S_1)}$ deve esserci almeno una componente di valore 1. Se il controllo da esito positivo viene eseguito il passo tre per determinare la migliore allocazione di $\tilde{n}_{n_i}^{F_h(S_1)}$ sugli elaboratori del metacomputer candidati.

Se invece il controllo da esito negativo, ovvero non esiste alcun elaboratore in grado di eseguire il nodo $\tilde{n}_{n_i}^{F_h(S_1)}$, si marca tale nodo come "non allocato" e si passa ad analizzare il nodo successivo. Si continua l'esecuzione dall'inizio del passo due.

Se non vi sono piú nodi da analizzare (ancora da allocare e senza marca di "non allocato") viene eseguito il passo quattro.

Con questa strategia si alloca allo stadio uno, secondo l'ordinamento stabilito al passo uno, ogni nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ per il quale esiste un elaboratore con sufficienti risorse per eseguirlo. Facendo così si riescono ad allocare sugli elaboratori paralleli del metacomputer, task implementati sfruttando forme di parallelismo strutturate, regolari e di tipo coarse-grain.

**passo 3** Si valutano i mapping applicabili al nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ selezionato al passo precedente e si sceglie tra questi quello per cui si ottiene il tempo di completamento minore.

Il procedimento inizia a partire dal nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ allo stadio uno. Per esso si valuta poi il tempo di completamento che impiegherebbe se $\tilde{n}_{n_i}^{F_h(S_1)}$ venisse eseguito sull'elaboratore selezionato. Successivamente si scende dallo stadio corrente $v$ a quello successivo $v+1$ in modo da individuare i task di $\tilde{n}_{n_i}^{F_h(S_1)}$ appartenenti all'insieme $I^{S_{v+1}}(\tilde{n}_{n_i}^{F_h(S_1)})$. I nodi di tale insieme vengono poi ordinati in base ai vincoli di precedenza presenti tra di essi, dopo di che, nell'ordine, vengono selezionati, e per ognuno di essi viene individuato l'elaboratore più adatto ad eseguirli. Fatto questo si è praticamente individuato il mapping relativo allo stadio $v+1$ del nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ e quindi, è possibile calcolare il tempo di completamento di $I^{S_{v+1}}(\tilde{n}_{n_i}^{F_h(S_1)})$ per poterlo confrontare con quello migliore trovato per uno degli stadi precedenti a $v+1$.

Mediante gli esempi mostrati nelle Figure 9, 10 e 11, che rappresentano graficamente la tecnica di mapping precedentemente descritta, è possibile rendere più chiaro il funzionamento di HMM.

Per il mapping del nodo $n_2$ allo stadio uno (Fig. 9), si suppone individuato dall'algoritmo l'elaboratore Mesh - MIMD. Ad ognuno degli undici nodi atomici compresi in $n_2$, è stato assegnato un processore di tale elaboratore. In Figura 10, invece, viene mostrato un possibile mapping di $n_2$ espanso allo stadio due. In questo caso il nodo $n_2$ viene visto come composto da 5 nodi: tre nodi atomici contrassegnati rispettivamente dai numeri 1, 10 ed 11, un *pipeline* (nodo gerarchico numero 12) composto dai nodi atomici 2, 3 e 4, ed un *farm* (nodo gerarchico numero 13) formato dai nodi atomici 5, 6, 7, 8 e 9. L'algoritmo HMM individua, per ognuno di questi cinque nodi, l'elaboratore più adatto ad eseguirlo (ad esempio in Figura 10, si è supposto che l'elaboratore migliore per l'esecuzione del pipeline sia l'Ipercubo - MIMD, per l'esecuzione del farm sia il Mesh - MIMD, ecc.) ed alloca i nodi atomici di ognuno di essi sui processori dell'elaboratore scelto. Analogamente, la Figura 11 mostra un possibile mapping per il nodo $n_2$ scomposto allo stadio tre. A questo stadio tutti i nodi sono atomici e quindi i costrutti di parallelismo non vengono tenuti in considerazione per le scelte di mapping.

Il nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ viene allocato allo stadio $v$ per il quale si è ottenuto, con la tecnica sopra descritta, il tempo di completamento minore. La Figura 12 mostra un esempio di tre possibili allocazioni di un nodo $n_i$ espanso ai suoi stadi: allo stadio uno il nodo è allocato su un solo elaboratore, allo stadio due i nodi di $I^{S_2}(\tilde{n}_{n_i}^{F_h(S_1)})$ sono ditribuiti tra due elaboratori e allo stadio tre i nodi di $I^{S_3}(\tilde{n}_{n_i}^{F_h(S_1)})$ sono ditribuiti tra tre elaboratori.

Il controllo del programma ritorna al passo due.

passo 4 Si controlla se esistono dei nodi marcati come "non allocati". In caso affermativo, con il procedimento descritto al passo tre, si allocano questi nodi nello stesso ordine stabilito al passo uno. È chiaro che, in questo caso, la ricerca degli elaboratori più adatti ad eseguire il nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ non ancora allocato, inizierà dal primo stadio $S_v$ per il quale esiste, per ogni task appartenente a $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$, un elaboratore in grado di eseguirlo. Se non esiste alcun nodo marcato come "non allocato", questo procedimento termina restituendo il mapping iniziale parziale relativo alla fase $F_h$ analizzata.

Quando tutte le fasi sono state elaborate, l'algoritmo termina restituendo la soluzione iniziale $Sol_{corr}$, in cui ai processori (non necessariamente tutti) degli elaboratori appartenenti al metacomputer sono stati assegnati i task paralleli secondo il criterio sopra spiegato. È importante sottolineare che nella soluzione $Sol_{corr}$, i nodi $n_i$ di $G_N$, appartenenti ad una generica fase $F_h$, possono essere allocati a differenti stadi anche sullo stesso elaboratore. Ad esempio può accadere che due nodi $n_1$ e $n_2$ appartenti alla $h$-esima fase siano allocati rispettivamente su $E_m$, allo stadio uno il primo, ed all'ultimo stadio il secondo.

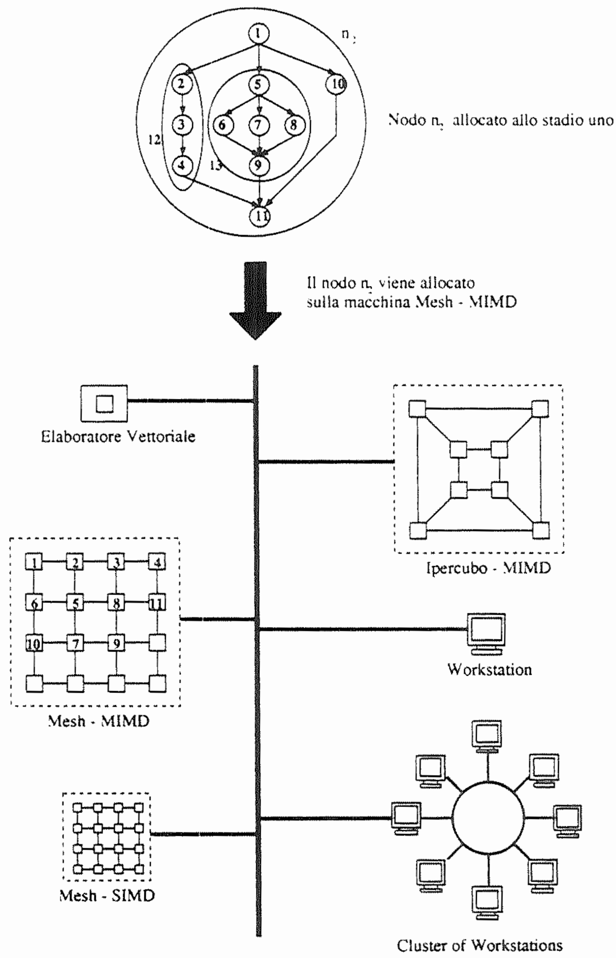



Figure 9: Esempio di allocazione del nodo $n_2$ allo stadio uno su un elaboratore del metacomputer.

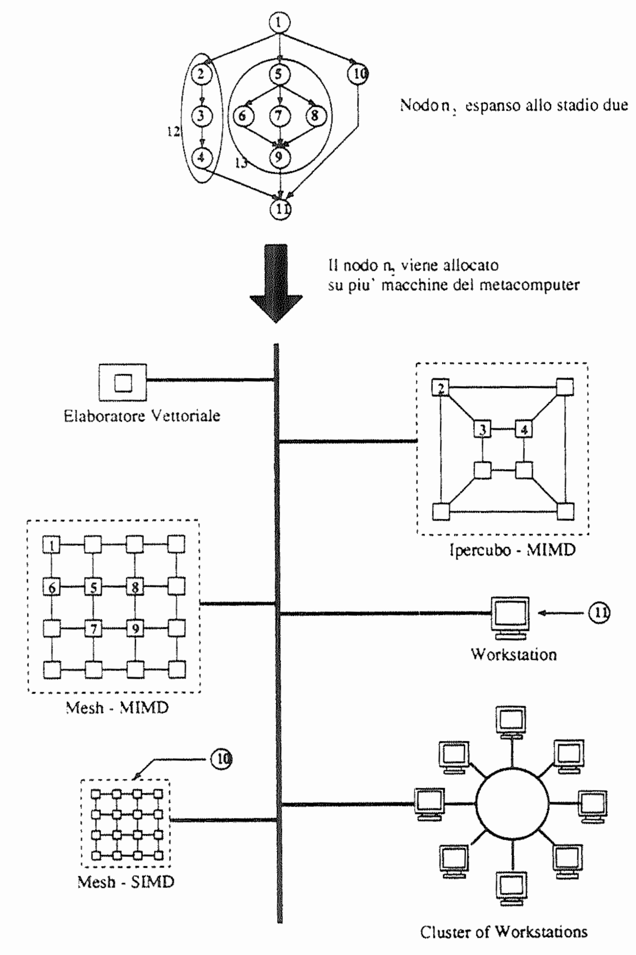



Figure 10: Esempio di allocazione del nodo $n_2$ allo stadio due su più elaboratori del metacomputer.

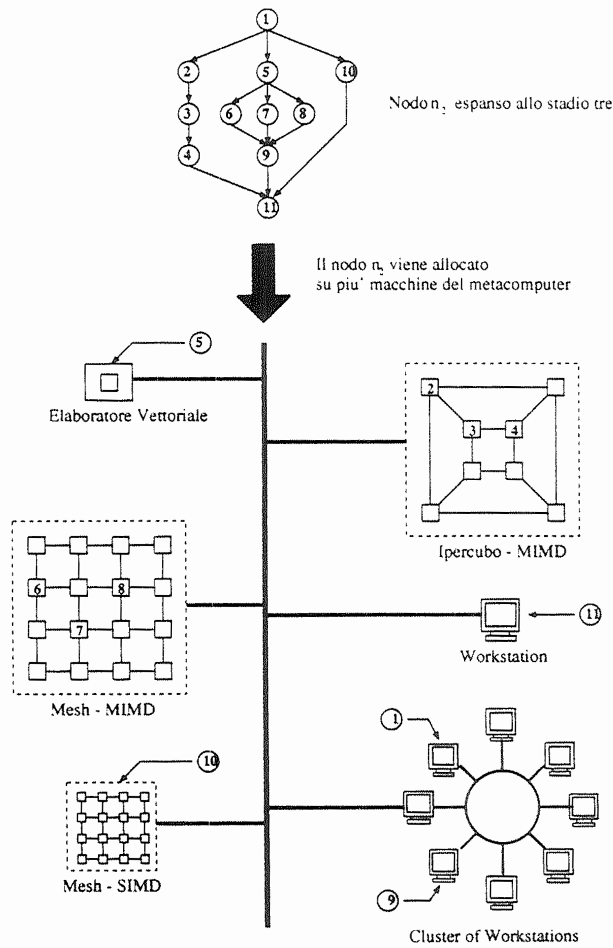



Figure 11: Esempio di allocazione del nodo $n_2$ allo stadio tre su più elaboratori del metacomputer.

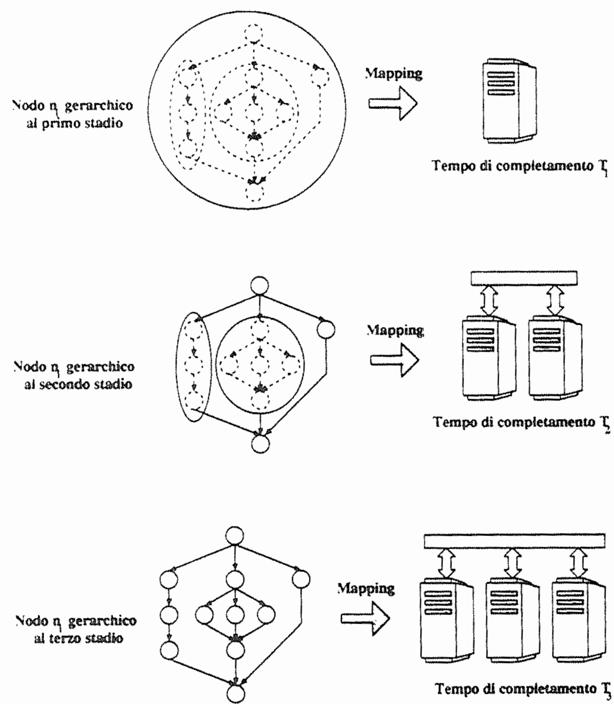



Figure 12: Esempio di allocazione di un nodo $n_i$ del grafo $G_N$. Si determina la migliore allocazione dei nodi di $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ con $v \in \{1, 2, 3\}$ e si sceglie quella per la quale si ottiene il minor tempo di completamento.

È altresì importante osservare che durante la fase di ricerca della soluzione iniziale abbiamo cercato di rendere più basso possibile il costo dovuto alle comunicazioni tra i task, mentre nessuna attività è stata prevista per bilanciare il carico tra gli elaboratori del metacomputer. Vedremo nelle successive fasi della ricerca locale come opera HMM per raggiungere un buon trade-off tra costo delle comunicazioni e bilanciamento del carico.

## 3.3 Criterio di Scelta dell'Elaboratore

Abbiamo utilizzato due differenti criteri per la scelta dell'elaboratore più adatto per l'allocazione di un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$: il primo viene utilizzato durante la ricerca della soluzione iniziale; il secondo viene utilizzato nelle succesive fasi della ricerca locale.

### 3.3.1 Fase di Ricerca della Soluzione Iniziale

Il criterio di scelta dell'elaboratore più adatto per l'allocazione di un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ si basa sul valore

$$\max\left(Aff_{\tilde{n}_{n_i}^{F_h(S_v)}}[E_m] \times W_{E_m} + D_{E_m}\right) \quad m = 1, .., N_{G_M} \tag{1}$$

dove $Aff_{\tilde{n}_{n_i}^{F_h(S_v)}}[E_m]$ rappresenta l'affinitá di $\tilde{n}_{n_i}^{F_h(S_v)}$ con l'elaboratore $E_m$, $W_{E_m}$ rappresenta la potenza computazionale di $E_m$ e $D_{E_m}$ vale zero se l'elaboratore $E_m$ considerato non è quello su cui é allocato il padre con il quale $\tilde{n}_{n_i}^{F_h(S_v)}$ comunica di più; assume il valore dell'effettiva quantitá di dati scambiata con il padre con cui $\tilde{n}_{n_i}^{F_h(S_v)}$ comunica di più, se l'elaboratore $E_m$ considerato è quello su cui è allocato tale nodo padre.

L'elaboratore selezionato per l'allocazione di un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ è quello per cui si ottiene il massimo valore della quantità calcolata in (1).

Nel caso in cui a più elaboratori corrisponda il valore massimo calcolato in (1), viene scelto tra questi quello meno carico per l'allocazione del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$.

L'utilizzo di tale formulazione per la scelta dell'elaboratore $E_m$ del metacomputer su cui allocare un task $\tilde{n}_{n_i}^{F_h(S_v)}$ è giustificato dal fatto che, oltre ad assegnare $\tilde{n}_{n_i}^{F_h(S_1)}$ ad un elaboratore con cui ha una buona affintà, vogliamo anche cercare di ridurre al minimo l'effetto delle comunicazioni sul tempo di completamento. Scegliere l'elaboratore $E_m$ per il quale il valore espresso in (1) è massimo, significa scegliere, grazie all'affinità, un elaboratore che sia affine al task e, grazie al termine $D_{E_m}$ previlegiare l'elaboratore su cui è allocato il nodo padre con cui $\tilde{n}_{n_i}^{F_h(S_v)}$ scambia la maggiore quantitá di dati. In questo modo si riesce a ridurre l'overhead dovuto alle comunicazioni.

### 3.3.2 Fasi Successive della Ricerca Locale

Il criterio di scelta dell'elaboratore più adatto per l'allocazione di un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ nelle fasi successive della ricerca locale si basa sul valore

$$\max\left\{\frac{Aff_{\tilde{n}_{n_i}^{F_h(S_v)}}[E_m] \times W_{E_m}}{C_{E_m}}\right\} \qquad con\ E_m \neq E_n \tag{2}$$

dove $C_{E_m}$ rappresenta il carico computazionale dell'elaboratore $E_m$ e $E_n$ è l'elaboratore su cui è attualmente allocato $\tilde{n}_{n_i}^{F_h(S_v)}$. Se a più elaboratori corrisponde lo stesso valore calcolato con l'espressione 2, si seleziona tra questi quello a cui è associato il valore del rapporto $\frac{C_{E_m}}{W_{E_m}}$ più piccolo.

L'utilizzo di tale formulazione per la scelta dell'elaboratore $E_m$ del metacomputer su cui allocare un nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ è giustificato dal fatto che vogliamo individuare un elaboratore affine a $\tilde{n}_{n_i}^{F_h(S_v)}$ nell'ottica di bilanciare per quanto possibile il carico sugli elaboratori del metacomputer.

## 3.4 Calcolo dei Tempi di Esecuzione dei Task

Vediamo adesso come viene calcolato il tempo di completamento dei task associati ad un nodo su un elaboratore.

Le informazioni utilizzate per questo scopo, associate ad ogni nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ e ai suoi nodi $\tilde{n}_{n_i}^{F_h(S_v)} \in I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ con $1 < v \leq \#stadi$ in cui può essere scomposto, sono: il costo computazionale $C_{\tilde{n}_{n_i}^{F_h(S_v)}}$, la stella entrante $BS(\tilde{n}_{n_i}^{F_h(S_v)})$ e la quantità totale $Q(\tilde{n}_{n_i}^{F_h(S_v)})$ delle comunicazioni interne a $\tilde{n}_{n_i}^{F_h(S_v)}$ (Fig. 13).

Le informazioni relative alla disponibilità (tempo di completamento dei task ad esso asseganti) dell'$u$-esimo processore di un generico elaboratore $E_m$ del metacomputer sono memorizzate nell'$u$-esima posizione $V_{E_m}[u]$ del vettore $V_{E_m}$.

I passi eseguiti dalla procedura che valuta il tempo di completamento $Time^{compl}(I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)}))$ dei nodi $\tilde{n}_{n_i}^{F_h(S_v)} \in I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ eseguiti sul sottinsieme $I_M(I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)}))$ degli elaboratori candidati sono i seguenti.

**passo 1** Si valuta il tempo di completamento del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$, scelto tra tutti quelli di $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ non ancora analizzati secondo l'ordine di esecuzione (rispettando quindi i vincoli di precedenza tra task). Nella formalizzione che segue, per ragioni di semplicità, abbiamo supposto che il nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ sia allocato sull'elaboratore $E_1$.

**passo 2** Il tempo di completamento di $\tilde{n}_{n_i}^{F_h(S_v)}$ è stimato mediante la seguente formula:

$$Time_{E_1}^{compl}(\tilde{n}_{n_i}^{F_h(S_v)}) = Time_{E_1}^{att}(\tilde{n}_{n_i}^{F_h(S_v)}) + Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)}) \tag{3}$$

Identificatore nodo
Costo di comunicazione
Costo computazionale

Stella entrante di $n_2$ — $BS(n_2) = (n_1, n_2)$

Quantita' di comunicazione interna ad $n_2$ — $Q(n_2) = 243$

Carico computazionale di $n_2$ — $C_{n_2} = 121$

Esempio di calcolo dei parametri associati ad $n_2$

Stella entrante del nodo 12 — $BS(12) = (1,2)$

Quantita' di comunicazione interna al nodo 12 — $Q(12) = 40$

Carico computazionale del nodo 12 — $C_{12} = 30$

Esempio di calcolo dei parametri associati al nodo 12 (pipeline) appartenente allo stadio due di $n_2$

Stella entrante del nodo 9 — $BS(9) = (6,9); (7,9); (8,9)$

Quantita' di comunicazione interna al nodo 9 — $Q(9) = 0$

Carico computazionale del nodo 9 — $C_9 = 10$

Esempio di calcolo dei parametri associati al nodo 9 appartenente allo stadio tre di $n_2$

Figure 13: Esempio di parametri associati ai task dell'applicazione.

dove la quantità $Time_{E_1}^{att}(\tilde{n}_{n_i}^{F_h(S_v)})$ indica il tempo che $\tilde{n}_{n_i}^{F_h(S_v)}$ deve attendere prima di poter iniziare la sua esecuzione sull'elaboratore $E_1$ e $Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)})$ il tempo di elaborazione di $\tilde{n}_{n_i}^{F_h(S_v)}$ su $E_1$.

Il valore di $Time_{E_1}^{att}(\tilde{n}_{n_i}^{F_h(S_v)})$ si calcola tramite la seguente espressione:

$$Time_{E_1}^{att}(\tilde{n}_{n_i}^{F_h(S_v)}) = \max\{Time_{E_1}^{com}(BS(\tilde{n}_{n_i}^{F_h(S_v)})), Time_{E_1}^{proc}(V[E_1])\} \quad (4)$$

La formula 4 permette di calcolare il tempo che il nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ deve attendere prima di poter iniziare la sua esecuzione. Tale tempo è il valore massimo tra il tempo di attesa dovuto alle comunicazioni, indicato con $Time_{E_1}^{com}(BS(\tilde{n}_{n_i}^{F_h(S_v)}))$, di $\tilde{n}_{n_i}^{F_h(S_v)}$ con i nodi della sua stella entrante e il tempo di attesa per ottenere tutti i processori, indicato con $Time_{E_1}^{proc}(V[E_1])$, richiesti da $\tilde{n}_{n_i}^{F_h(S_v)}$ per poter essere eseguito.

Il calcolo del tempo $Time_{E_1}^{com}(BS(\tilde{n}_{n_i}^{F_h(S_v)}))$ viene effettuato mediante la seguente espressione:

$$\begin{aligned} Time_{E_1}^{com}(BS(\tilde{n}_{n_i}^{F_h(S_v)})) &= \max\left\{Time_{E_2}^{fine}(\bar{n}) + \frac{D_{\bar{n},\tilde{n}_{n_i}^{F_h(S_v)}}}{B_{E_2,E_1}}\right\} \quad (5) \\ &\forall\ \bar{n}\ con\ (\bar{n}, \tilde{n}_{n_i}^{F_h(S_v)}) \in BS(\tilde{n}_{n_i}^{F_h(S_v)}) \end{aligned}$$

in cui $D_{\bar{n},\tilde{n}_{n_i}^{F_h(S_v)}}$ indica la quantità di dati che $\bar{n}$ invia a $\tilde{n}_{n_i}^{F_h(S_v)}$, $B_{E_2,E_1}$ la banda di connessione tra l'elaboratore $E_1$ su cui si vuole allocare $\tilde{n}_{n_i}^{F_h(S_v)}$ e l'elaboratore $E_2$ su cui è stato allocato $\bar{n}$ e $Time_{E_2}^{fine}(\bar{n})$ indica l'istante di tempo in cui il nodo $\bar{n}$ padre di $\tilde{n}_{n_i}^{F_h(S_v)}$ termina la sua esecuzione. In altre parole, il tempo di attesa dovuto alle comunicazioni di $\tilde{n}_{n_i}^{F_h(S_v)}$ si calcola effettuando tante somme, quanti sono gli archi nella stella entrante $BS(\tilde{n}_{n_i}^{F_h(S_v)})$ di $\tilde{n}_{n_i}^{F_h(S_v)}$, tra il tempo di terminazione di un nodo padre $\bar{n}$ di $\tilde{n}_{n_i}^{F_h(S_v)}$ ed il tempo di comunicazione tra $\bar{n}$ ed $\tilde{n}_{n_i}^{F_h(S_v)}$ calcolato mediante il rapporto $\frac{D_{\bar{n},\tilde{n}_{n_i}^{F_h(S_v)}}}{B_{E_2,E_1}}$. A $Time_{E_1}^{com}(BS(\tilde{n}_{n_i}^{F_h(S_v)}))$, viene poi associato il maggiore tra i valori ottenuti con queste somme.

Il calcolo del tempo di attesa $Time_{E_1}^{proc}(V[E_1])$ viene fatto mediante la seguente formula:

$$Time_{E_1}^{proc}(V[E_1]) = Avaiable(E_1, V_{E_1}, P(\tilde{n}_{n_i}^{F_h(S_v)})) \quad (6)$$

La funzione *Avaiable* prende come parametri, l'identificatore $E_1$ dell'elaboratore sul quale viene eseguito il mapping di $\tilde{n}_{n_i}^{F_h(S_v)}$, il vettore $V_{E_1}$ contenente

le informazione sui tempi in cui i processori di $E_1$ terminano i lavori ad essi assegnati e il numero di processori $P(\tilde{n}_{n_i}^{F_h(S_v)})$ richesto per l'esecuzione del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$. Il valore restituito dalla funzione *Avaiable* viene assegnato alla variabile $Time_{E_1}^{proc}(V[E_1])$ e rappresenta l'istante di tempo in cui saranno disponibili $P(\tilde{n}_{n_i}^{F_h(S_v)})$ processori su $E_1$.

La quantità $Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)})$ rappresenta la stima del tempo di elaborazione del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ sull'elaboratore $E_1$. Poiché il nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ può essere composto da più nodi atomici, ognuno dei quali verrà allocato su un diverso processore di $E_1$ (diverso dai processori dello stesso elaboratore assegnati agli altri nodi di $\tilde{n}_{n_i}^{F_h(S_v)}$), per stimare il tempo $Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)})$ in modo attendibile è necessario considerare sia le comunicazioni *intra-nodo* (che sono quelle tra i nodi atomici di $\tilde{n}_{n_i}^{F_h(S_v)}$), che il costo computazionale di $\tilde{n}_{n_i}^{F_h(S_v)}$. Conseguentemente la quantità $Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)})$ può essere espressa mediante la seguente formula:

$$Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)}) = \frac{Q(\tilde{n}_{n_i}^{F_h(S_v)})}{B^{int}(E_1) \times Card(\tilde{n}_{n_i}^{F_h(S_v)})} + \frac{C_{\tilde{n}_{n_i}^{F_h(S_v)}} \times Aff_{\tilde{n}_{n_i}^{F_h(S_v)}}[E_1]}{W_{E_1}} \quad (7)$$

nella quale l'espressione

$$\frac{Q(\tilde{n}_{n_i}^{F_h(S_v)})}{B^{int}(E_1) \times Card(\tilde{n}_{n_i}^{F_h(S_v)})}$$

rappresenta il costo medio dovuto alle comunicazioni intra-nodo e la quantità

$$\frac{C_{\tilde{n}_{n_i}^{F_h(S_v)}} \times Aff_{\tilde{n}_{n_i}^{F_h(S_v)}}[E_1]}{W_{E_1}}$$

rappresenta una stima del tempo medio di calcolo necessario per eseguire il nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ sui processori dell'elaboratore $E_1$ ad esso assegnati.
È importante fare una precisazione riguardo alla formulazione espressa in (7). A $Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)})$ calcolato con la formula (7) viene associato un valore approssimato. Questa approssimazione è dovuta all'aver considerato il nodo gerarchico $\tilde{n}_{n_i}^{F_h(S_v)}$ ad un generico livello $v$ ($1 < v \leq \#stadi$) come un nodo atomico. Conseguentemente i costi di computazione e di comunicazione ad esso associati sono stati stimati come, rispettivamente, il rapporto tra la somma dei costi computazionali e la potenza

dell'elaboratore usato e il rapporto tra la somma dei costi di comunicazione dei nodi atomici in esso contenuti e la banda di trasmissione dei canali di comunicazione usati. Va da se che all'ultimo stadio tale stima risulterà meno approssimata. Il valore approssimato di $Time_{E_1}^{elab}(\tilde{n}_{n_i}^{F_h(S_v)})$ serve per valutare più velocemente il tempo di completamento di ogni nodo $\tilde{n}_{n_i}^{F_h(S_1)}$ allo stadio $S_v$. Ogni volta che viene individuato uno stadio $S_v$ per il quale il tempo di completamento di $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ è minore del migliore tempo di completamento di $I^{S_u}(\tilde{n}_{n_i}^{F_h(S_1)})$ ad uno stadio precedente ($u < v$), i nodi atomici di $\tilde{n}_{n_i}^{F_h(S_v)}$ vengono allocati sui processori disponibili di $E_1$, ed il tempo di completamento di ognuno di tali nodi viene calcolato in modo meno approssimato.

È inoltre importante osservare che, cercando di allocare un nodo gerarchico $\tilde{n}_{n_i}^{F_h(S_1)}$ allo stadio che ne permette la più veloce esecuzione, ogni volta che viene trovata un'allocazione ad uno stadio $S_w$, migliore di quella corrente HMM deve aggiornare le sue strutture dati per memorizzare questa allocazione.

**passo 3** Se non sono stati analizzati tutti i nodi $\tilde{n}_{n_i}^{F_h(S_v)} \in I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ si torna direttamante al passo uno. Altrimenti, l'algoritmo termina restituendo il valore del tempo $Time_{E_1}^{compl}(\tilde{n}_{n_i}^{F_h(S_v)})$ relativo al nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ che completa la sua esecuzione per ultimo tra tutti quelli nell'insieme $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$.

Vediamo con un esempio come l'algoritmo HMM calcola il tempo di completamento di un nodo $n_i$ a tutti gli stadi, in modo da individuare lo stadio per il quale si ottiene un mapping che permetta di ottenerene il minor tempo di completamento.

Supponiamo di aver già allocato sul metacomputer di Figura 5 il nodo $n_1$ del grafo $G_N$ di Figura 2 e che il prossimo nodo di $G_N$ da elaborare sia $n_2$. Tramite le Figure 9, 10, 11 precedentemente introdotte che mostrano, rispettivamente, per ognuno dei tre stadi in cui è espandibile $n_2$, tre suoi possibili mapping, è possibile capire come vengono calcolati i tempi di elaborazione dei task.

In Figura 9, il nodo $n_2$, allo stadio uno, è stato allocato sull'elaboratore Mesh - MIMD che indicheremo con $E_1$. Per calcolare il tempo di completamento $Time_{E_1}^{compl}(\tilde{n}_{n_2}^{F_h(S_1)})$ (3) di $n_2$ su tale elaboratore, è necessario valutare il tempo che $n_2$ deve attendere prima di poter iniziare l'esecuzione e stimare il suo tempo di esecuzione su $E_1$.

Il tempo $Time_{E_1}^{att}(\tilde{n}_{n_2}^{F_h(S_1)})$ (4) che $n_2$ deve attendere prima di iniziare l'esecuzione è dato dal massimo tra il valore del tempo $Time_{E_1}^{com}(BS(\tilde{n}_{n_2}^{F_h(S_1)}))$ (5) di attesa, dovuto alla ricezione dei dati inviatigli dal nodo padre $n_1$, che si suppone allocato sull'elaboratore $E_2$, ed il valore del tempo $Time_{E_1}^{proc}(V[E_1])$ (6) che deve attendere affinchè abbia a disposizione sull'elaboratore $E_1$ i processori necessari per l'esecuzione.

Il tempo $Time_{E_1}^{elab}(\tilde{n}_{n_2}^{F_h(S_1)})$ necessario per l'esecuzione di $n_2$ sull'elaboratore $E_1$ (Mesh - MIMD), è stimato utilizzando la formula (7) nella quale, la prima componente della somma, permette di stimare il tempo medio delle comunicazione tra i nodi atomici interni ad $n_2$ e la seconda permette di stimare il tempo necessario per il calcolo di $n_2$. Quest'ultimo tempo viene valutato considerando il carico computazionale associato ad $n_2$ (ottenuto dalla somma del carico computazionale associato ad ognuno dei suoi nodi atomici), la potenza computazionale di $E_1$ e l'affinità tra $n_2$ ed $E_1$. Avendo considerato quali valori del parametro affinità 1 e -1 e poiché HMM effettua l'allocazione solo se il parametro affinità è uguale a 1, nel nostro caso il parametro affinità non produce effetti sul calcolo del tempo di elaborazione. Comunque la formulazione da noi utilizzata permette di mettere in relazione il tempo di completamento di un modulo del programma parallelo con l'elaboratore usato anche nel caso in cui non ci sia completa corrispondenza tra codice ed elaboratore.

Tramite la Figura 10 che mostra un possibile mapping del nodo $n_2$ espanso allo stadio due, è possibile capire come la procedura del calcolo dei tempi di completamento, permette di calcolare la stima di tale tempo per il nodo $n_2$.
In questo caso, il nodo $n_2$ viene eseguito su più elaboratori differenti, in particolare il nodo 1 ed il nodo 13 (farm) sono assegnati all'elaboratore Mesh - MIMD, il nodo 12 è assegnato all'Ipercubo - MIMD, il nodo 10 al Mesh - SIMD ed il nodo 11 alla Workstation. Dalla Figura 10 si vede chiaramente che l'esecuzione di $n_2$ inizia con il nodo 1. Terminata la sua esecuzione, il nodo 1, invia dati ai nodi 12, 13 e 10 che possono così iniziare in parallelo la loro esecuzione. Il nodo 11 viene elaborato per ultimo e può iniziare la sua esecuzione soltanto dopo aver ricevuto i dati necessari inviatigli attraverso gli archi nella sua stella entrante.
Per calcolare il tempo di completamento del nodo $n_2$ (scomposto allo stadio due ed allocato sugli elaboratori del metacomputer come rappresentato in Figura 10), la procedura per il calcolo dei tempi di completamento, inizia stimando il tempo di completamento del nodo 1 sull'elaboratore Mesh - MIMD. Per far ciò considera sia il massimo tra il tempo di attesa per ottenere i dati inviati al nodo 1 attraverso gli archi presenti nella sua stella entrante e il tempo per ottenere i processori dell'elaboratore che il tempo necessario per la sua computazione.
Una volta stimato l'istante di terminazione del nodo 1, la procedura per il calcolo dei tempi stima il tempo di completamento dei nodi 12, 13 e 10. Anche in questo caso, per ognuno di tali nodi, mediante la stella entrante, viene valutato l'istante di tempo in cui ognuno di essi ottiene i dati necessari per poter essere eseguito. Mediante il vettore $V_{E_m}$ relativo all'elaboratore assegnato al nodo in esame, viene, invece, valutato l'istante di tempo in cui è disponibile il numero di processori richiesti per l'esecuzione di tale nodo. Poi considerando il valore massimo tra i due istanti di tempo precedentemente calcolati, viene stimato, per ogni nodo, il tempo necessario per la sua esecuzione.
Una volta calcolati i tempi di terminazione dei nodi 12, 13 e 10, viene calcolato il tempo di esecuzione del nodo 11. Mediante l'esame della sua stella

entrante viene valutato l'istante di tempo in cui tutte le informazioni necessarie per la sua esecuzione sono disponibili. Tramite l'esame del vettore $V_{E_m}$ associato all'elaboratore a cui il nodo 11 è assegnato, viene valutato il tempo in cui un processore diviene disponibile per eseguirlo. Considerando il valore massimo tra i due tempi prima detti, viene calcolato il tempo necessario per l'esecuzione del task ad esso associato.

In Figura 11, viene mostrato il mapping del nodo $n_2$ scomposto al terzo stadio. In questo caso tutti i nodi sono atomici e vengono allocati sugli elaboratori del metacomputer senza considerare i costrutti paralleli considerati negli stadi precedenti. Per quello che riguarda il calcolo del tempo di esecuzione valgono le considerazioni già fatte nei due esempi precedenti. In questo esempio, quindi, il primo nodo ad essere eseguito è quello etichettato con 1, il quale, alla sua terminazione invia i dati ai nodi 2, 5 e 10 in modo da permettergli di iniziare concorrentemente l'esecuzione. Una volta terminata l'esecuzione, questi nodi, comunicano con i loro figli, e così via fino all'esecuzione del nodo 11 che inizia la sua esecuzione dopo la ricezione dei dati inviatigli dai nodi 4, 9 e 10.

## 3.5 Funzione di Trasformazione

La funzione di trasformazione utilizzata dall'algoritmo di ricerca locale permette di trovare la soluzione migliore $Sol_{best}$ partendo dalla soluzione iniziale $Sol_{corr}$. Poiché quest'ultima potrebbe risultare la migliore si pone $Sol_{best} = Sol_{corr}$. Prima di descrivere in modo più dettagliato quali sono i passi eseguiti dalla funzione di trasformazione, è necessario precisare che tale funzione considera atomici i nodi della soluzione iniziale $Sol_{corr}$, ottenuti dai nodi $n_i \in G_N$ espansi allo stadio per il quale si è ottenuto il miglior tempo di completamento dell'applicazione.

Detto questo è possibile analizzare dettagliatamente la strategia su cui si basa la funzione di trasformazione.

**passo 1** Vengono verificati i due seguenti criteri di terminazione della ricerca locale:

a) il numero totale di iterazioni eseguite è minore o uguale al limite massimo consentito $T_{max}$;

b) il numero di iterazioni eseguite senza migliorare la soluzione $Sol_{corr}$ è minore o uguale al limite massimo previsto $P_{max}$.

Se anche una sola delle condizioni sopra elencate è verificata, l'algoritmo HMM termina restituendo la soluzione $Sol_{best}$. Altrimenti viene determinato il cammino critico relativo alla soluzione corrente $Sol_{corr}$ nel grafo $G_E$ di esecuzione (Fig. 14).

Il cammino critico è il cammino di costo massimo nel grafo $G_E$ di esecuzione dell'applicazione. Il costo di tale cammino, che coincide con il

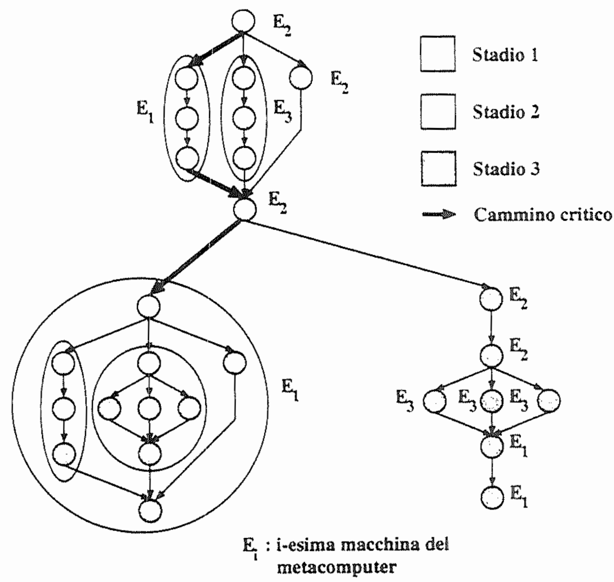



Figure 14: Grafo $G_E$ di esecuzione dell'applicazione.

tempo di completamento dell'applicazione, è calcolato sommando i tempi di elaborazione e di comunicazione relativi ai nodi $\tilde{n}_{n_i}^{F_h(S_v)}$ che gli appartengono.

**passo 2** Si analizza l'intorno della soluzione corrente $Sol_{corr}$ alla ricerca di nuove soluzioni.

L'intorno di una soluzione è l'insieme di tutte le soluzioni da essa ottenibili spostando un solo nodo del cammino critico dall'elaboratore su cui risiede, all'elaboratore individuato mediante l'espressione 2 (§3.3). È allora chiaro che la cardinalità dell'intorno di una soluzione è al più pari alla lunghezza del cammino critico.

Il passo due si sviluppa scegliendo nel cammino critico associato a $Sol_{corr}$, a partire dalla radice fino ad arrivare alla foglia, il primo nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ per il quale non è ancora stata valutata la possibilità di trasferimento su un altro elaboratore. Per tale nodo si cerca l'elaboratore $E_m$ più adatto ad eseguirlo mediante l'espressione 2. Successivamente si passa ad eseguire il passo tre.

Se non esiste alcun elaboratore $E_m$ candidato per eseguire $\tilde{n}_{n_i}^{F_h(S_v)}$, il controllo del programma ritorna all'inizio del passo due ed il nodo analizzato non viene spostato.

Se sono stati analizzati tutti i nodi del cammino critico la procedura di ricerca locale esegue il passo quattro.

**passo 3** Si calcola il tempo di completamento della nuova soluzione individuata e si ritorna al passo due.

**passo 4** L'algoritmo confronta il tempo di completamento dell'applicazione relativo a tutte le soluzioni di mapping individuate nell'intorno di $Sol_{corr}$. Tra queste soluzioni HMM sceglie la migliore $Sol_{new}$.

Quando la soluzione $Sol_{new}$ è peggiore di $Sol_{corr}$, ma non è stato raggiunto il numero massimo di iterazioni consentito senza migliorare la soluzione $Sol_{corr}$, si pone $Sol_{corr} = Sol_{new}$, si lascia invariata $Sol_{best}$ e si torna al passo uno. Se invece tale limite è stato raggiunto l'algorimto termina restituendo la soluzione $Sol_{best}$.

Quando la soluzione $Sol_{new}$ trovata è migliore della soluzione $Sol_{corr}$ si pone $Sol_{corr} = Sol_{new}$. Si confronta poi $Sol_{new}$ con $Sol_{best}$ e si pone $Sol_{best} = Sol_{new}$ se a $Sol_{new}$ corrisponde un tempo di completamento inferiore a quello corrispondente a $Sol_{best}$. Si torna così al passo uno.

| $Aff_{\tilde{n}_{n_i}^{F_h(S_v)}}[E_m]$ | Valore di affinità tra il codice di $\tilde{n}_{n_i}^{F_h(S_v)}$ e l'elaboratore $E_m$ |
|---|---|
| $B_{E_2,E_1}$ | Banda di connessione tra l'elaboratore $E_2$ e l'elaboratore $E_1$ |
| $B^{int}(E_1)$ | Banda interna di una connessione tra processori nell'elaboratore $E_1$ |
| $BS(\tilde{n}_{n_i}^{F_h(S_v)})$ | Stella entrante del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ |
| $Card(\tilde{n}_{n_i}^{F_h(S_v)})$ | Numero dei nodi atomici interni a $\tilde{n}_{n_i}^{F_h(S_v)}$ |
| $C_{E_m}$ | Carico computazionale dell'elaboratore $E_m$ |
| $C_{\tilde{n}_{n_i}^{F_h(S_v)}}$ | Carico computazionale del nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ |
| $D_{E_m}$ | Variabile che assume valore 0 quando un nodo non è allocato sull'elaboratore su cui è allocato il padre con cui comunica di più. Assume il valore dell'effettiva quantità scambiata tra i due nodi, altrimenti |
| $D_{\tilde{n},\tilde{n}_{n_i}^{F_h(S_v)}}$ | Quantità di dati che $\tilde{n}$ invia a $\tilde{n}_{n_i}^{F_h(S_v)}$ |
| $E_m$ | $m$-esimo elaboratore del metacomputer |
| $F_h$ | $h$-esima fase di $G_N$ |
| $G_A$ | Grafo dell'applicazione espanso all'ultimo livello di gerarchia. Ogni suo nodo è un task atomico |
| $G_E$ | Grafo dell'esecuzione dell'applicazione sugli elaboratori del metacomputer |
| $G_M$ | Grafo del metacomputer. I suoi nodi sono elaboratori e gli archi le connessiono tra tali elaboratori |
| $G_N$ | Grafo gerarchico dell'applicazione |
| $I_M(I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)}))$ | Insieme degli elaboratori candidati per l'allocazione dei nodi di $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ |
| $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ | insieme dei nodi contenuti nello stadio $v$-esimo di $n_i$ che appartiene alla fase $h$-esima |
| $N_{G_A}$ | Numero di nodi del grafo $G_A$ |
| $N_{G_N}$ | Numero di nodi del grafo $G_N$ |
| $N_{G_M}$ | Numero di nodi (elaboratori) del grafo $G_M$ |
| $N_{n_i}$ | Numero di nodi (task atomici) di $n_i$ espanso al suo ultimo livello di gerarchia |
| $N_F$ | Numero di fasi in cui è stato suddiviso il grafo $G_N$ |
| $n_i = \tilde{n}_{n_i}^{F_h(S_1)}$ | Generico nodo di $G_N$ al primo livello della gerarchia |
| $\tilde{n}_{n_i}^{F_h(S_v)}$ | generico nodo appartenente all'insieme $I^{S_v}(\tilde{n}_{n_i}^{F_h(S_1)})$ |
| $P_{max}$ | Numero massimo di iterazioni consentito senza migliorare la soluzione $Sol_{corr}$ |
| $P(\tilde{n}_{n_i}^{F_h(S_v)})$ | Numero di processori necessari al task $\tilde{n}_{n_i}^{F_h(S_v)}$ per poter essere eseguito |
| $p_{E_m}^{(u)}$ | $u$-esimo processore dell'elaboratore $E_m$ |
| $Q(\tilde{n}_{n_i}^{F_h(S_v)})$ | Ammontare delle comunicazioni interne al nodo $\tilde{n}_{n_i}^{F_h(S_v)}$ |
| $Sol_{best}$ | Soluzione migliore individuata da HMM |
| $Sol_{corr}$ | Soluzione corrente |
| $Sol_{new}$ | Nuova soluzione |
| $S_v$ | $v$-esimo stadio di un nodo gerarchico dell'applicazione |
| $Time_{E_1}^{att}(\tilde{n}_{n_i}^{F_h(S_v)})$ | Tempo che $\tilde{n}_{n_i}^{F_h(S_v)}$ deve attendere prima di poter iniziare la sua esecuzione |
| $Time_{E_1}^{com}(BS(\tilde{n}_{n_i}^{F_h(S_v)}))$ | Tempo che è necessario attendere per ottenere i dati inviati dai nodi $\tilde{n}$ a $\tilde{n}_{n_i}^{F_h(S_v)}$ |
| $Time_{E_1}^{compl}(\tilde{n}_{n_i}^{F_h(S_v)})$ | Tempo di completamento di $\tilde{n}_{n_i}^{F_h(S_v)}$ sull'elaboratore $E_1$ |

# 4 Valutazione di HMM

La validazione di *Heterogeneous Multiphases Mapping (HMM)* è stata fatta confrontando i risultati da esso ottenuti elaborando i grafi dell'applicazione e dei metacomputer usati per valutare l'algoritmo di mapping *Heterogeneous Task Graph onto Heterogeneous System Graph (HTHS)* [9] proposto da M. Eshaghian e Y. Wu, con quelli ottenuti dall'elaborazione degli stessi grafi da parte di HTHS e dell'algoritmo di mapping esaustivo da noi realizzato.

La valutazione di HMM è stata fatta sia confrontando i risultati da esso ottenuti elaborando i grafi dell'applicazione e del metacomputer usati per valutare l'algoritmo di mapping *A Suboptimal Heterogeneous Mapping (SHM)* [7] proposto da M. Eshaghian, A. C. Parker e Y. Wu, con quelli ottenuti dall'elaborazione degli stessi grafi da parte di SHM e dall'algoritmo esaustivo, sia utilizzando un grafo di un'applicazione e di un metacomputer generati casualmente. In quest'ultimo caso non è stato possibile fare alcun confronto dei risultati ottenuti. Quest'ultimo test, infatti, è stato effettuato esclusivamente per mostrare come HMM riesce ad allocare i task di un'applicazione parallela sugli elaboratori di un metacomputer. Per fare questo test abbiamo utilizzato un grafo dell'applicazione ed un grafo del metacomputer di grosse dimensioni e ciò ha reso, ovviamente, impossibile l'utilizzo dell'algoritmo esaustivo per individuare il mapping ottimo che ci avrebbe permesso di verificare la bontà della soluzione individuata da HMM.

L'algoritmo esaustivo che abbiamo sviluppato individua tutti i possibili mapping processo-processore nel seguente modo:

- si determina il numero $m^n$ di possibili mapping, dove $n$ indica il numero dei task dell'applicazione ed $m$ il numero di processori presenti nel metacomputer;
- si utilizza un vettore $N$ di cardinalità $n$ in cui all'$i$-esima posizione è associato l'$i$-esimo task dell'applicazione;
- si generano tutti i numeri da 0 ad $m^n$ in base $m$ e, per ognuno di essi, si inseriscono le cifre che lo compongono nelle rispettive componenti del vettore $N$. Ad ognuno di tali numeri corrisponde il mapping che alloca l'$i$-esimo task dell'applicazione sul processore $N[i]$ per $1 \leq i \leq n$.

Prima di descrivere i risultati dei test effettuati è necessario fare una precisazione sull'uso del parametro affinità codice-elaboratore. Abbiamo visto nel Capitolo 5 che l'algoritmo HMM utilizza, per le scelte di mapping, il parametro affinità codice-elaboratore. Tale affinità permette di selezionare, per l'allocazione di ogni modulo dell'applicazione, gli elaboratori che, oltre ad avere le risorse (memoria, numero di processori, ambiente software, ecc.) necessarie per eseguirlo, hanno il modello architetturale corrisponednte al modello computazionale usato dal modulo trattato. L'algoritmo HMM, successivamente, tra gli elaboratori selezionati sceglie: nella fase di ricerca della soluzione iniziale quello che

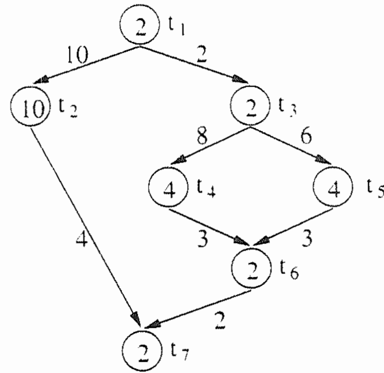


Figure 15: Grafo diretto aciclico rappresentante l'applicazione utilizzata per il test di HMM e di HTHS.

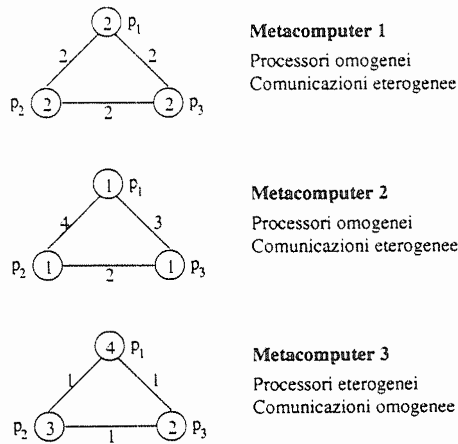



Figure 16: Grafi simmetrici, rappresentanti i metacomputer, utilizzati l'esecuzione dell'applicazione di Figura 6.1.

Figure 17: Diagrammi di Gantt descriventi il mapping dell'applicazione rappresentata dal grafo di Figura 6.1 sul metacomputer 1 rappresentato nel grafo di Figura 6.2.

permette di ridurre maggiormente il costo dovuto alle comunicazioni; nelle fasi successive della ricerca locale quello che permette di bilanciare meglio il carico computazionale tra gli elaboratori del metacomputer.

Non avendo a disposizione informazioni sulle risorse presenti sugli elaboratori del metacomputer, nel test dell'algoritmo HMM, abbiamo utilizzato l'affinità così come usata per il test degli algoritmi HTHS e SHM. In quest'ultimi i task appartenenti ad un dato modello computazionale sono stati eseguiti sugli elaboratori aventi corrispondente tipo architetturale.

I test sono stati eseguiti tutti su un elaboratore HP 9000/700.

## 4.1 Validazione

Per la validazione di HMM sono stati effettuati i test n. 1, n. 2 e n. 3. Per ognuno di tali test, l'applicazione rappresentata dal grafo di Figura 15 è stata eseguita su un elaboratore rappresentato da uno dei grafi di Figura 16.

**Test n. 1** Il grafo mostrato in Figura 15 rappresentante l'applicazione viene allocato sul metacomputer 1 rappresentato nel grafo di Figura 16. Nella Figura 15 i task dell'applicazione sono rappresentati dai nodi etichettati con $t_i$ con $1 \leq i \leq 7$; il costo computazionale di tali task è specificato dai numeri interni a tali nodi; il costo delle comunicazioni tra task è rappresentato dai numeri sugli archi. Nella Figura 16 gli elaboratori del metacomputer sono rappresentati dai nodi etichettati con $P_j$ con $1 \leq j \leq 3$; la potenza computazionale di ciascun elaboratore è rappresentata dal numero associato a ciascun nodo; la banda

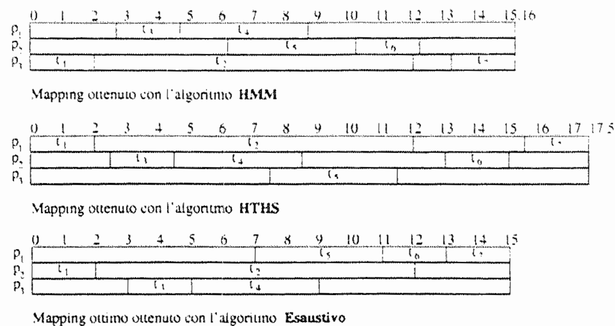
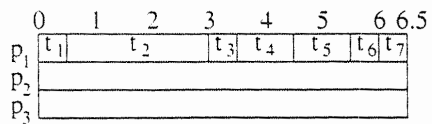



Figure 18: Diagrammi di Gantt descriventi il mapping dell'applicazione rappresentata dal grafo di Figura 6.1 sul metacomputer 2 rappresentato nel grafo di Figura 6.2.

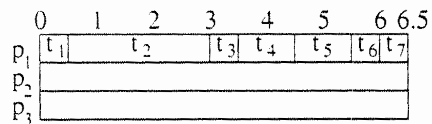
del canale di comunicazione tra due nodi è rappresentata dal numero associato all'arco che li connette. Mediante l'algoritmo esaustivo abbiamo individuato il mapping ottimo, quello cioè che permette all'applicazione di terminare nel minimo tempo possibile. Il tempo di completamento dell'applicazione allocata secondo tale mapping è pari a 9 secondi. La Figura 17 mostra come, in questo caso, l'algoritmo HMM riesce ad allocare i task dell'applicazione in modo ottimale. Ciascun diagramma di Gantt rappresentato in tale figura mostra su quale elaboratore $p_j$ è allocato ciasun task $t_i$ e quali sono i suoi tempi di inizio e di fine (rappresentati dai numeri in alto sul diagramma) elaborazione. HMM converge alla soluzione mostrata in Figura 17 dopo aver analizzato 25 tra i 2187 mapping possibili. L'esecuzione di HMM è terminata in 0.09 secondi; quella dell'algoritmo esaustivo è terminata in 1 secondo.

**Test n. 2** L'applicazione rappresentata dal grafo di Figura 15 viene allocata sui metacomputer 2 rappresentato nel grafo di Figura 16. Mediante l'algoritmo esaustivo abbiamo individuato sia il mapping ottimo per il quale si ottiene un tempo di completamento dell'applicazione di 15 secondi, sia il mapping peggiore per il quale si ottiene un tempo di completamento dell'applicazione di 37.5 secondi. La Figura 18 mostra come l'algoritmo HMM, allocando l'applicazione in modo tale da farla terminare in 15.16 secondi, individua un mapping peggiore di quello ottimo di circa lo 0.7 %. HMM converge alla soluzione mostrata in Figura 18 dopo aver analizzato 27 tra i 2187 mapping possibili. L'esecuzione di HMM è terminata in 0.24 secondi; quella dell'algoritmo esaustivo è terminata in 1 secondo.

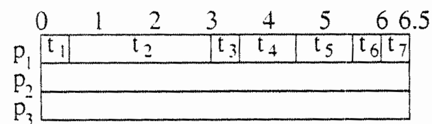
**Test n. 3** L'applicazione rappresentata dal grafo di Figura 15 viene allocata sul metacomputer 3 rappresentato nel grafo di Figura 16. Mediante l'algoritmo esaustivo abbiamo individuato il mapping ottimo per il quale si ottiene un tempo




Mapping ottenuto con l'algoritmo **HMM**




Mapping ottenuto con l'algoritmo **HTHS**




Mapping ottimo ottenuto con l'algoritmo
**Esaustivo**

Figure 19: Diagrammi di Gantt descriventi il mapping dell'applicazione rappres entata dal grafo di Figura 6.1 sul metacomputer 3 rappresentato nel grafo di Figura 6.2.

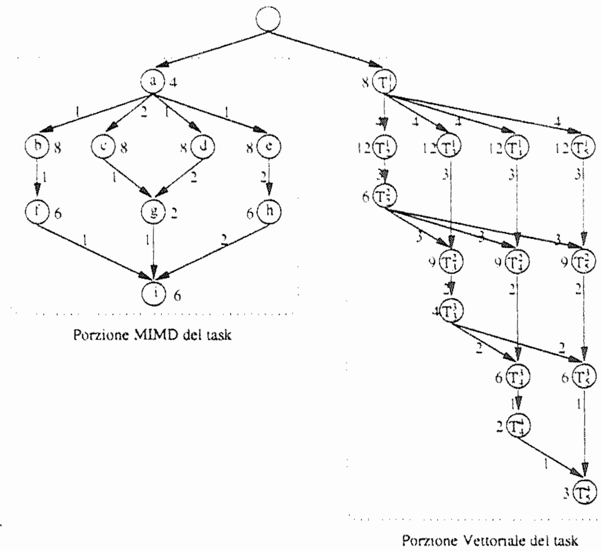



Figure 20: Grafo diretto aciclico rappresentante l'applicazione composta da codice MIMD e Vettoriale utilizzata per eseguire il test degli algoritmi HMM e SHM.

di completamento dell'applicazione di 6.5 secondi. La Figura 19 mostra come l'algoritmo HMM, riesce ad allocare i task dell'applicazione in modo ottimale. HMM converge alla soluzione mostrata in Figura 19 dopo aver analizzato 16 mapping tra i 2187 possibili. L'esecuzione di HMM è terminata in 0.11 secondi: quella dell'algoritmo esaustivo è terminata in 1 secondo.

## 4.2 Valutazione

Le Figure (Fig. 20 e Fig. 21) rappresentano, rispettivamente, i grafi della applicazione e del metacomputer utilizzati per i due test successivi. L'applicazione, come mostra il grafo di Figura 20, è composta da una parte di codice MIMD e da una parte di codice Vettoriale. Il metacomputer, come mostra il grafo di Figura 21 è composto da un elaboratore MIMD e da un elaboratore Vettoriale. Analogamente a quanto fatto in [7], abbiamo eseguito HMM in modo da allocare la parte MIMD dell'applicazione sull'elaboratore MIMD (Test n. 4) e la parte Vettoriale dell'applicazione sull'elaboratore Vettoriale (Test n. 5).

La Figura 22 mostra come il grafo dell'applicazione di Figura 20 viene allocato sul metacomputer rappresentato dal grafo di Figura 21 usando gli algoritmi HMM, SHM ed esaustivo. Tali risultati sono stati prodotti dall'unione di quelli ottenuti dal test n. 4 e dal test n. 5 descritti nel seguito del presente para-

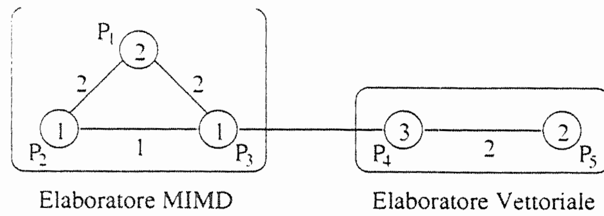



Figure 21: Grafo simmetrico rappresentante il metacomputer su cui è stato allocato il grafo di Figura 6.6 usando gli algoritmi HMM e SHM.

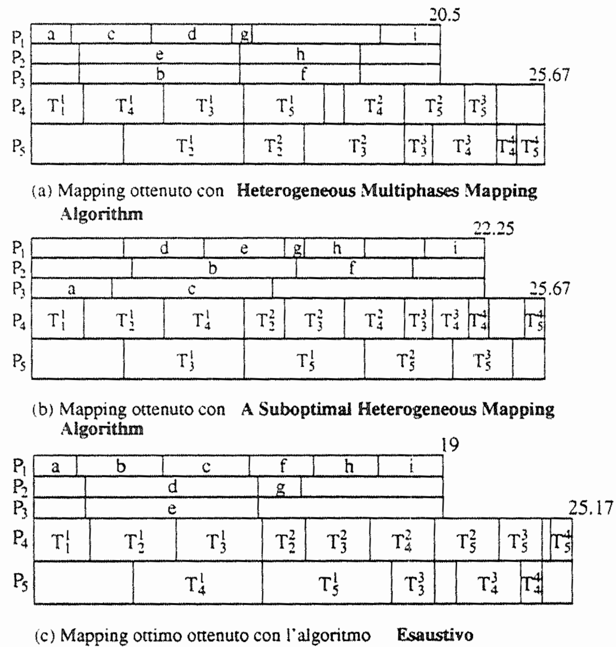



Figure 22: Diagrammi di Gantt descriventi l'allocazione dei task dell'applicazione rappresentata dal grafo di Figura 6.6 sul metacomputer rappresentato dal grafo di Figura 6.7 ottenuta, rispettivamente, con gli algoritmi HMM, SHM ed esaustivo.

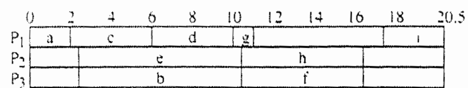
grafo. Come è possibile vedere dalla Figura 22, il tempo di completamento dell'applicazione non cambia se questa viene allocata con HMM o con SHM. È comunque da osservare che l'allocazione prodotta da HMM, rispetto a quella prodotta da SHM, permette di ridurre il tempo di completamento della parte di codice MIMD dell'applicazione e, di conseguenza, consente un migliore utilizzo dell'elaboratore MIMD che sarà disponibile per altre applicazioni in tempi più brevi.

Nelle Figure (Fig. 23 e Fig. 24), relative ai test n. 4 e n. 5, sono mostrati oltre ai diagrammi di Gantt rappresentanti i mapping prodotti dagli algoritmi HMM, SHM ed esaustivo precedentemente introdotti, altri diagrammi di Gantt rappresentanti i mapping ottenuti dagli algoritmi *Cluster-M Non Uniform Mapping Algorithm* [5], *Lo's Max-Flow/Min-Cut Algorithm* [17] e *Shen and Tsai's A* Searching Algorithm* [20]. I diagrammi di Gantt ottenuti dall'esecuzione di quest'ultimi algoritmi, sono stati introdotti soltanto per dare maggiore completezza al test.

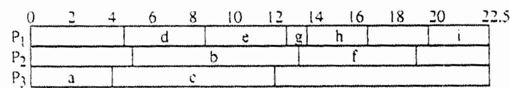
**Test n. 4** La porzione di codice MIMD dell'applicazione rappresentata dal grafo in Figura 20 viene allocata sull'elaboratore MIMD del metacomputer rappresentato dal grafo di Figura 21. Mediante l'algoritmo esaustivo abbiamo individuato sia il mapping ottimo per il quale si ottiene un tempo di completamento dell'applicazione di 19 secondi, sia il mapping peggiore per il quale si ottiene un tempo di completamento dell'applicazione di 60 secondi. La Figura 23 mostra come l'algoritmo HMM, allocando l'applicazione in modo tale da farla terminare in 20.5 secondi, individua un mapping peggiore di quello ottimo di circa il 3.6 %. HMM converge alla soluzione mostrata in Figura 23 dopo aver analizzato 21 tra i 19683 mapping possibili. L'esecuzione di HMM è terminata in 0.2 secondi; quella dell'algoritmo esaustivo è terminata in 19.75 secondi.

**Test n. 5** La porzione di codice Vettoriale dell'applicazione rappresentata dal grafo di Figura 20 viene allocata sull'elaboratore Vettoriale del metacomputer rappresentato dal grafo di Figura 21. Mediante l'algoritmo esaustivo abbiamo individuato sia il mapping ottimo per il quale si ottiene un tempo di completamento dell'applicazione di 25.17 secondi, sia il mapping peggiore per il quale si ottiene un tempo di completamento dell'applicazione di 55.69 secondi. La Figura 24 mostra come l'algoritmo HMM, allocando l'applicazione in modo tale da farla terminare in 25.67 secondi trova un mapping peggiore di quello ottimo di circa l'1.6 %. HMM converge alla soluzione mostrata in Figura 24 dopo aver analizzato 19 mapping tra i 16384 possibili. L'esecuzione di HMM è terminata in 0.51 secondi; quella dell'algoritmo esaustivo è terminata in 47.92 secondi.

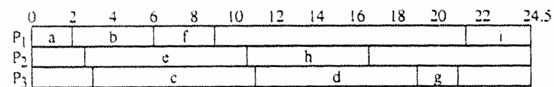
Analizzando i risultati degli ultimi due test, può sembrare strano che l'algoritmo esaustivo, per analizzare i 19683 mapping possibili del test n. 4, abbia impiegato 19.75 secondi, e per analizzare i 16384 possibili mapping del test n. 5, abbia impiegato 47.92 secondi. Quanto detto si verifica perchè, per ogni possibile mapping, il costo (in termini di tempo di CPU) per calcolare il tempo



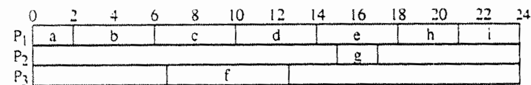

(a) Mapping ottenuto con **Heterogeneous Multiphases Mapping Algorithm**



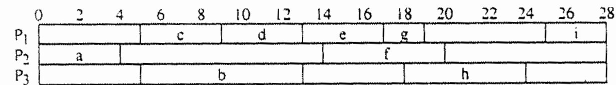

(b) Mapping ottenuto con **A Suboptimal Heterogeneous Mapping Algorithm**



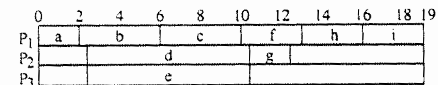

(c) Mapping ottenuto con **Cluster-M Non-Uniform Mapping Algorithm**




(d) Mapping ottenuto con **Lo's Max-Flow / Min-Cut Algorithm**




(e) Mapping ottenuto con **Shen and Tsai's A*Searching Algorithm**




(f) Mapping ottenuto con l'algoritmo **Esaustivo**

Figure 23: Diagrammi di Gantt descriventi l'allocazione dei task della parte di codice MIMD dell'applicazione rappresentata dal grafo di Figura 6.6 sui processori dell'elaboratore MIMD del metacomputer rappresentato dal grafo di Figura 6.7.

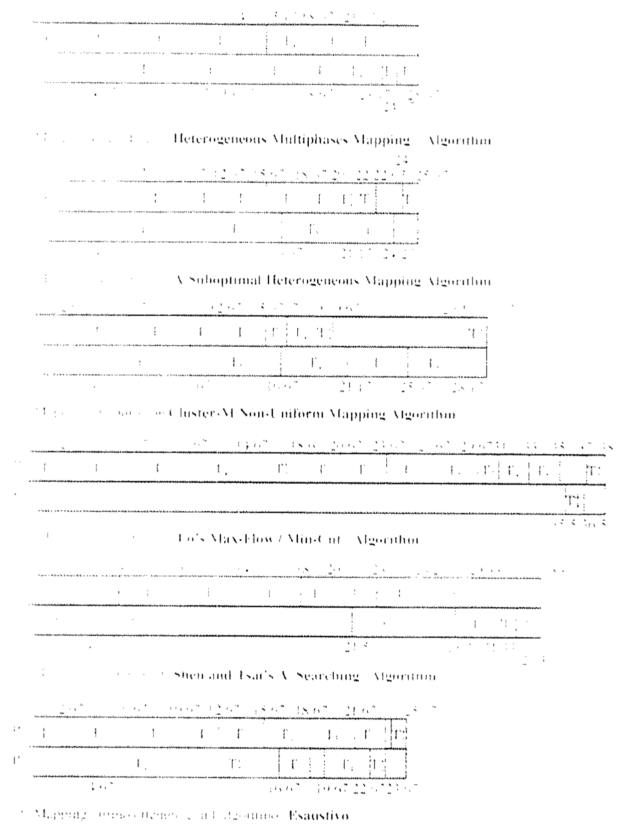



Figure 24: Diagrammi di Gantt descriventi l'allocazione della parte di codice Vettoriale dell'applicazione rappresentata dal grafo di Figura 6.6 sui processori dell'elaboratore Vettoriale del metacomputer rappresentato dal grafo di Figura

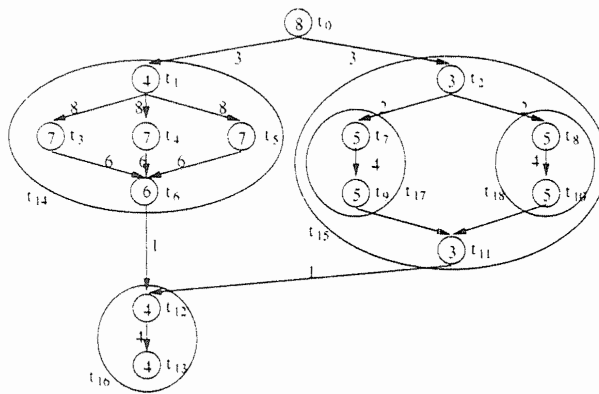


Figure 25: Grafo diretto aciclico rappresentante l'applicazione utilizzata per il test n. 6 di HMM.

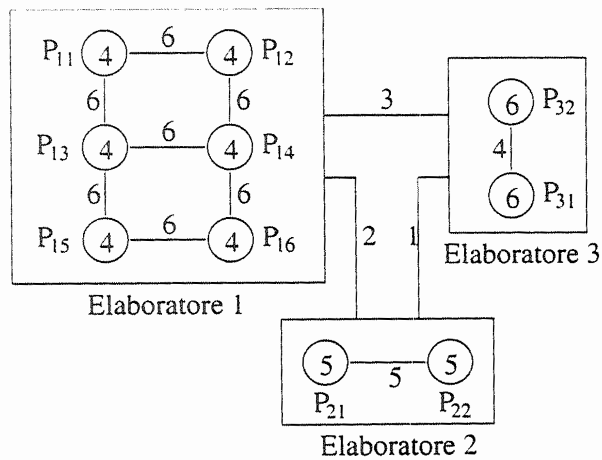



Figure 26: Grafo simmetrico rappresentante il metacomputer utilizzato per l'esecuzione dell'applicazione di Figura 6.11.

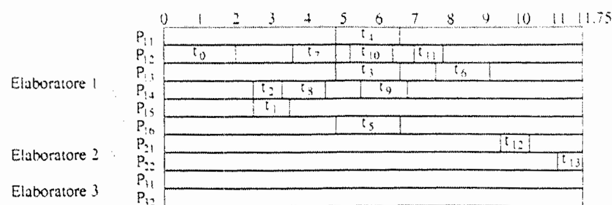


Figure 27: Diagramma di Gantt rappresentante l'allocazione prodotta da HMM, dell'applicazione rappresentata dal grafo di Figura 6.11, sul metacomputer rappresentato dal grafo di Figura 6.12.

di completamento del modulo vettoriale (test n.5) dell'applicazione, allocato sull'elaboratore vettoriale, è molto più alto di quello per calcolare il tempo di completamento del modulo MIMD dell'applicazione, allocato sull'elaboratore MIMD.

**Test n. 6** L'applicazione rappresentata dal grafo di Figura 25 viene allocata sul metacomputer rappresentato dal grafo di Figura 26. Dal Diagramma di Gantt di Figura 27 è possibile vedere come HMM riesce ad allocare i nodi gerarchici (sez. 2) del grafo di Figura 25 rappresentanti moduli paralleli dell'applicazione sui processori di uno stesso elaboratore. Da tale diagramma di Gantt è evidente come i nodi $t_3$, $t_4$ e $t_5$ rappresentanti i worker del farm rappresentato dal nodo gerarchico $t_{14}$, vengono eseguiti in parallelo sui processori dell'elaboratore 1 rappresentato nel grafo di Figura 26. Il nodo gerarchico $t_{15}$, invece, è stato allocato scomposto allo stadio due (sez. 2). In questo modo sono stati considerati per il mapping i nodi $t_2$, $t_{17}$, $t_{18}$, $t_{11}$. Tali nodi sono stati allocati tutti sui processori dell'elaboratore 1 rappresentato dal grafo di Figura 26. In nodi $t_{17}$, $t_{18}$ rappresentanti dei costrutti paralleli pipeline, sono stati allocati rispettivamente su due distinti processori, in modo da rendere possibile l'esecuzione in parallelo dei task atomici in essi contenuti. HMM alloca i nodi $t_{14}$ e $t_{15}$ in modo da renderne possibile l'esecuzione in parallelo. Infatti, come mostra la Figura 27, la loro esecuzione termina quasi contemporaneamente. Infine, il pipeline rappresentato dal nodo $t_{16}$, viene allocato su due processori distinti dell'elaboratore due rappresentato nel grafo di Figura 26. In questo modo i task atomici costituenti il nodo $t_{16}$ possono essere eseguiti in parallelo. L'esecuzione di HMM termina, dopo aver analizzato 29 tra i $10^{14}$ mapping possibili, in 0.59 secondi.

# 5 Conclusioni

I test effettuati mostrano come l'algoritmo di mapping HMM riesce ad allocare sugli elaboratori di un metacomputer i moduli di applicazioni parallele implementate utilizzando sia modelli ristretti che modelli non ristretti. La metrica

utilizzata per valutare la qualità dei mapping ottenuti, è il tempo di completamento dell'applicazione allocata sugli elaboratori del metacomputer secondo tali mapping. Nei casi in cui è stato possibile effettuare un confronto con i risultati prodotti da altri algoritmi di mapping, HMM ha sempre individuato dei mapping di qualità superiore o uguale a quella raggiunta da tali algoripseudo.textmi. I confronti dei risultati ottenuti da HMM con quelli prodotti dall'algoritmo esaustivo hanno dimostrato che HMM ha sempre raggiunto soluzioni molto vicine a quella ottima. La qualità del mapping prodotto nell'ultimo test. non è stata valutata. Comunque. da un'analisi del risultato ottenuto sembra che HMM sia riuscito ad allocare i task costituenti l'applicazione in modo da permetterne l'esecuzione in parallelo in accordo con i costrutti di parallelismo utilizzati. Tale mapping, inoltre, dovrebbe essere tale da consentire un buon utilizzo delle risorse del metacomputer.

# 6 Pseudo codice delle principali routines

**Algoritmo HMM**

Esegue il mapping di una metapplicazione su un metacomputer.
La Figura 28 mostra lo pseudo-codice di **HMM**.
Parametri usati:

- $G_H$ = grafo strutturato rappresentante l'applicazione;
- $G_T$ = grafo rappresentante l'applicazione a livello dei task atomici;
- $G_M$ = grafo rappresentante il metacomputer.

**Procedura GetGraphPhases**

Suddivide in fasi un grafo rappresentante l'applicazione o un suo task.
La Figura 29 mostra lo pseudo-codice di **GetGraphPhases**.
Parametri considerati:

- $NodeMatrix$ = matrice nodi-nodi associata al grafo strutturato rappresentante l'applicazione o ad un suo task;
- $NodeNum$ = numero di nodi costituenti il grafo strutturato rappresentante l'applicazione o un suo task.

**Procedura GetInitAlloc**

Individua il migliore stadio per l'allocazione di un nodo.
La Figura 30 mostra lo pseudo-codice di **GetInitAlloc**.
Parametri usati:

- $\tilde{n}$ = nodo strutturato da allocare;
- $ListaDeiNodi$ = lista di identificatori di nodi. Tale lista contiene gli identificatori dei nodi contenuti ad ogni stadio di $\tilde{n}$;

```
Algoritmo HMM (G_H, G_T, G_M)
 begin
   dividi_in_fasi (G_H)
   while (ci sono fasi da analizzare) do
      begin
        ordina_nodi (nodi della fase corrente)
        while (non sono stati analizzati tutti
              i nodi della fase corrente) do
           begin
             if (esiste una macchina per allocare il nodo n_i
               in modo atomico) then
               cerca_migliore_allocazione (n_i)
             else
              marca il nodo come non allocato
           end
        while (ci sono nodi non allocati) do
           cerca_migliore_allocazione (n_i)
   end
   Sol_best = Sol_corr
   while (numero totale di iterazioni < T_max and
          numero totale di iterazioni senza
          miglioramenti < P_max) do
      begin
        cerca_cammino_critico (Sol_corr)
        while (ci sono nodi del cammino non analizzati) do
           begin
             cerca_macchina_migliore (n_i)
             calcola nuova soluzione e memorizzala
           end
        scegli soluzione migliore Sol_new nell'intorno
        if (Sol_new > Sol_corr and
          numero di iterazioni < P_max) then
          Sol_corr = Sol_new
        else
         numero iterazioni fatte senza miglioramenti = P_max
        if (Sol_new < Sol_corr) then
          begin
          Sol_corr = Sol_new
          if (Sol_new < Sol_best) then
            Sol_best = Sol_new
          end
        end
      return Sol_best
   end
```

Figure 28: Pseudo codice dell'algoritmo HMM.

```
Procedure GetGraphPhases (NodeMatrix, NodeNum)
  begin
    h = 1; /* h = contatore fasi */
    Individua il nodo radice n_i;
    n_i = "analizzato";
    n_i ∈ F_h; /* F_h indica la h-esima fase */
    while Esistono nodi "non analizzati" do
      begin
        h = h + 1;
        for each n_j = "non analizzato" do
          begin
            Flag = false;
            for each n_k ∈ BS(n_j) do
              if n_k = "non analizzato" then Flag = true;
            if Flag = false then
              begin
                n_j ∈ F_h;
                n_j = "inserito";
              end
          end
        for each n_j = "inserito" do
          n_j = "analizzato"
      end
```

Figure 29: Pseudo codice della procedura **GetGraphPhases**.

```
Procedure GetInitAlloc (ñ, ListaDeiNodi, N_Stadi, ListaSottografi)
  begin
    for i = 1 to N_Stadi do
      begin
        Dividi in Fasi il grafo associato allo stadio S_i;
        for j = 1 to #Fasi(S_i) do
          begin
            Ordina i nodi della Fase F_j in modo crescente
            sul costo computazionale;
            for k = 1 to #Nodi(F_j) do
              begin
                Scegli la macchina per il
                nodo n_k; /* procedura InitGetBestMachine */
                Alloca il nodo sulla macchina;
                /* procedura AllocAtomicTask */
                Calcola e memorizza il suo tempo
                di completamento; /* procedura AllocAtomicTask */
              end
          end
          Memorizza il tempo di esecuzione del nodo che termina
          per ultimo tra quelli di ñ allo stadio S_i;
      end
      Seleziona lo stadio S_h per il quale si ottiene
      il minimo tempo di completamento;
      Return allocazione di ñ allo stadio S_h;
  end
```

Figure 30: Pseudo codice della procedura **GetInitAlloc**.

- $N_{Stadi}$ = numero stadi di $\tilde{n}$;
- $ListaSottografi$ = lista contenente gli identidicatori dei nodi contenuti nei sottografi rappresentanti i nodi appartenenti ai vari stadi di $\tilde{n}$.

**Procedura InitGetBestMachine**
Individua la macchina migliore per l'allocazione di un nodo.
La Figura 31 mostra lo pseudo-codice di **InitGetBestMachine**.
Parametro usato: $Nodo$ = generico nodo del grafo rappresentante l'applicazione;
**Procedura GetNew Sol**
Individua a partire dalla soluzione corrente una nuova soluzione.
La Figura 32 mostra lo pseudo-codice di **GetNew Sol**.
Parametri considerati:

```
Procedure InitGetBestMachine (Nodo)
  begin
    D = 0;
    MacchinaSelezionata = "none";
    for each  n_j ∈ BS(Nodo)  do
      if  DatiInviati(n_j ⟶ Nodo) > D then
         begin
           D = DatiInviati(n_j ⟶ Nodo);
           M = Macchina(n_j); /* macchina su cui risiede n_j */
         end
    Temp = 0;
    for i = 1 to #Macchine do
      begin
        if  (i = M  and  Temp < Aff(Nodo, Macchina(i))×
        Potenza(Macchina(i)) +D) then
           begin
             Temp = Aff(Nodo, Macchina(i)) × Potenza(Macchina(i)) +D;
             MacchinaSelezionata = Macchina(i);
           end
        if  (i ≠ M  and  Temp < Aff(Nodo, Macchina(i))×
        Potenza(Macchina(i))) then
           begin
             Temp = Aff(Nodo, Macchina(i)) × Potenza(Macchina(i));
             MacchinaSelezionata = Macchina(i);
           end
      end
      Return MacchinaSelezionata
  end
```

Figure 31: Pseudo codice della procedura **InitGetBestMachine**.

```
Procedure GetNewSolution (G_Sol_corr, SelectTask)
  begin
    Individua la macchina migliore M_best per
    SelectTask (procedura LocGetBestMachine);
    Alloca SelectTask su M_best;
    Individua il nodo n_1 di SelectTask che
    viene eseguito per primo;
    Analizza la stella entrante BS(n_1) e calcola
    il tempo di inizio di SelectTask su M_best;
    for each i = 1 to #NodiAtomiciApplicazione do
      begin
        Calcola il tempo di inizio e di fine
        di SelectTask su M_best;
        Aggiorna la lista TaskAllocList dei
        tempi di esecuzione;
      end;
    T = 0;
    for each i = 1 to #NodiAtomiciApplicazione do
      if TempoDiCompletamento(TaskAtomico_i) > T then
      T = TempoDiCompletamento(TaskAtomico_i);
    Associa T al tempo di completamento della
    nuova soluzione New_Sol;
    Return(New_Sol);
  end;
```

Figure 32: Pseudo codice della procedura **GetNewSolution**.

- $G_{Sol_{corr}}$ = grafo della soluzione corrente;
- $SelectTask$ = task (generalmente strutturato) selezionato per essere riallocato.

**Procedura AllocAtomicTask**

Alloca, sui processori della macchina selezionata, i task atomici di un nodo strutturato e calcola per ognuno di essi i tempi di inizio e fine esecuzione.

La Figura 33 mostra lo pseudo-codice di **AllocAtomicTask**.

Parametri usati:

- $n_s$ = generico nodo strutturato;
- $M_s$ = macchina selezionata per $n_s$.

**Procedura GetFreeCPU**

```
Procedure AllocAtomicTask (n_s, M_s)
  begin
    Individua la lista ordinata L_CPU delle CPU di M_s su
    cui eseguire i task atomici di n_s (procedura GetFreeCPU);
    dividi_in_fasi (n_s); /* procedura GetGraphPhases */
    for i=1 to #Fasi(n_s) do
      for j=1 to #Nodi(fase F_i) do
        begin
          MaxTimeDati = 0;
          for each n_p ∈ BS(Nodo_j) do
            if MaxTimeDati < T_d
            /* T_p = Tempo in cui Nodo_j ottiene i dati da n_p */
            then MaxTimeDati = T_d;
            StartTime(Nodo_j) = MaxTimeDati;
            Seleziona la prima CPU (CPU_corr) di
            L_CPU non ancora utilizzata;
            Calcola il tempo di esecuzione T_esec(Nodo_j)
            di Nodo_j su CPU_corr;
            EndTime(Nodo_j) = StartTime(Nodo_j) + T_esec(Nodo_j);
        end
  end
```

Figure 33: Pseudo codice della procedura **AllocAtomicTask**.

```
Procedure GetFreeCPU (Macchina, N_CPU)
  begin
    Individua tutte le CPU di Macchina; /* sono Tot_CPU */
    Ordina le CPU in modo crescente sul loro tempo
    di fine esecuzione; /* viene usato il QuickSort */
    Seleziona nella lista ordinata le prime N_CPU;
  end
```

Figure 34: Pseudo codice della procedura **GetFreeCPU**.

Restituisce le prime $N_{CPU}$ libere di una macchina.
La Figura 34 mostra lo pseudo-codice di **GetFreeCPU**.
Parametri usati:

- Macchina = macchina selezionata per l'allocazione di un nodo;
- $N_{CPU}$ = numero CPU richieste per l'allocazione di un nodo.

**Procedura GetCriticalPath**
Individua il cammino critico sul grafo di esecuzione della applicazione. Il cammino critico è quello al quale corrisponde il tempo di completamento dell'applicazione sugli elaboratori del matacomputer.
La Figura 35 mostra lo pseudo-codice di **GetCriticalPath**.
Parametro usato: $G_S$ = grafo rappresentante un mapping codice-elaboratore (Soluzione corrente).

# References


[1] H. H. Ali, H. El-Rewini, and T. G. Lewis. Task Scheduling in Parallel and Distributed Systems. *PTR Prentice Hall*, June 1994.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. $P^3L$: A Structured High-Level Parallel Languade and its Structured Support. *Technical Report HPL-PSC-93-55, Hewlett Packard Laboratories, Pisa Science Center*, 1993.

[3] F. Baiardi, A. Tomasi, and M. Vanneschi. *Architettura dei Sistemi di Elaboarazione.* Franco Angeli, 1987.

[4] S. H. Bokhari. On the Mapping Problem. *IEEE Transaction on Computers*, 3:207–214, March 1981.

```
Procedure GetCriticalPath (G_S)
  begin
    Individua il nodo n_u di G_S che termina per ultimo;
    Individua il nodo n_r di G_S che inizia per primo;
    /* n_r e n_u appartengono entrambi al cammino critico */
    while N_r ≠ N_u do
       begin
       FirstAtomicTask = Primo_{n_u}; /* Task di n_u
       che inizia per primo */
         for i = 1 to #Archi_{G_A} do /* G_A = grafo dei task atomici */
           begin
             Individua i padri di FirstAtomicTask;
             for each p ∈ BS(FirstAtomicTask) do
               Calcola il tempo in cui p fa iniziare FirstAtomicTask;
             Padre = p | p fa iniziare per ultimo FirstAtomicTask;
           end
           Analizza i nodi di G_S e individua quello
           (n_p) a cui appartiene padre;
           n_u = n_p;
           n_u appartiene al cammino critico;
       end
  end
```

Figure 35: Pseudo codice della procedura **GetCriticalPath**.

[5] S. Chen, M. M. Eshaghian, and Y. C. Wu. Mapping Arbitrary Non-Uniform Task Graphs onto Arbitrary Non-Uniform System Graphs. *International Conference on Parallel Processing*, 2:191–195, 1995.

[6] M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. *Parallel Programming Models Based on the Restricted Computation Structure Approach.* Dipartimento di Informatica Università di Pisa, Technical report n. R/3/133, 1983.

[7] M. M. Eshaghian, A. C. Parker, and Y. C. Wu. A Suboptimal Heterogeneous Mapping. Technical Report. Departement of Computer and Information Science, New Jersey Institute of Tecnology Newark, New Jersey. Avaiable at ftp://ftp.njit.edu/pub/cis/mary/Cluster-M/, 1996.

[8] M. M. Eshaghian and Muhammad E. Shaaban. Cluster-M Parallel Programming Paradigm. *International Journal of High Speed Computing*, 6(2):287–309, 1994.

[9] M. M. Eshaghian and Y. C. Wu. Heterogeneous Task Graphs onto Heterogeneous System Graphs. *Proceedings of Sixth Heterogeneous Computing Workshop. IEEE Computer Society Press*, 1:147–160, April 1997.

[10] S. M. Figueira and F. Berman. Mapping Parallel Application to Distributed Heterogeneous Systems. Techinical Report. Departement of Computer Science and Engineering University of California, San Diego, 1996.

[11] G. C. Fox. *Solving Problem on Concurrent Processors.* PTR Prentice Hall, 1988.

[12] R. F. Freund and H. J. Siegel. Heterogeneous Processing. *IEEE Computer*, 26(6):13–17, June 1993.

[13] A. J. G. Hey. Reconfigurable Transputer Networks: Pratical Concurrent Computation .In R. J. Elliot and C. A. R. Hoare. *Scientific Applications of Multiprocessors, Prentice Hall*, pages 39–54, 1989.

[14] D. S. Johnson and M. R. Gary. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[15] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. L. Wang. Heterogeneous Computing: Challenges and Opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[16] A. Lee and C. H. Cheng. A Knowledge-based Approach for Task Allocation in Heterogeneous Distributed Computer Systems . *Expert Systems*, 12(4):303–311, 1995.

[17] V. M. Lo. Heuristic Algorithms for Task Assignament in Distributed Systems. *IEEE Transaction on Computers*. 37(11):1384–1397, November 1988.

[18] G. Lobe. P. Lu. S. Melax. I. Parsons. J. Schaffer. C. Smith. and D. Szafron. The Enterprise Model for Developing Distributed Applications. *Technical Report TR-92-20. University of Alberta*. 1992.

[19] S. C. S. Porto and C. C. Ribeiro. A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints . *International Journal of High Speed Computing*, 7(1):45–71. 1995.

[20] C. Shen and W. Tsai. A Graph Matching Approach to Optimal Task Assignament in Distributed Computing Systems Using a Minmax Criterion. *IEEE Transaction on Computers*. C-34(3):197–203. March 1985.

[21] P. Shroff. D. W. Watson. N. S. Flann. and R. F. Freund. Genetic Simulated Annealing for Scheduling Data-Dependent Task in Heterogeneous Environtments. Technical Report. Heterogeneous Computing Team NC-COSC RDTE Division 4221 San Diego. California 92152-7446.

[22] H. Singh and A.Youssef. Mapping and Scheduling Heterogeneous Task Graph using Genetic Algorithms. Fifty Workshop on Heterogeneous Computing at the Tenth International Parallel Processing Symposium, Honolulu, Haway, April 1996.

[23] M. Vanneschi. *Architectures and Programming Issues in General Purpose Parallel Computers in General Purpose Parallel Computers*. Ed. ETS, 1995.

[24] L. Wang, H. J. Siegel, and V. P. Roychowdhury. A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments. Technical Report. Parallel Processing Laboratory, School of Electrical and Computer Engineering, Purdue University USA, 1996.