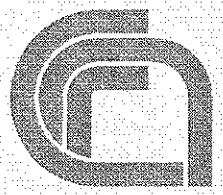


Consiglio Nazionale delle Ricerche

Task scheduling in ambiente eterogeneo
Studio, implementazione e valutazione di un algoritmo di clustering per il mapping su metacalcolatori

R. Baraglia, D. Laforenza
G. Faieta, M. Formica

CNUCE C96-1

Task scheduling in ambiente eterogeneo
Studio, implementazione e valutazione di un algoritmo di clustering per il mapping su metacalcolatori

R. Baraglia, D. Laforenza
G. Faieta, M. Formica
CNUCE - Instituto del Consiglio Nazionale delle Ricerche
Via S. Maria, 36 - I56100 Pisa (Italy)
Tel. +39-50-593111 - Fax +39-50-904052
e-mail: R.Baraglia@cnuce.cnr.it, D.Laforenza@cnuce.cnr.it

19 Gennaio 1996

Sommario. È noto che il problema del mapping su ambiente omogeneo è NP-completo [Bok81]. Gli ambienti eterogenei sono più complessi di quelli omogenei e pertanto il problema del mapping è ancora più difficile. Tra le difficoltà addizionali citiamo: la differente affinità che ciascuna classe architetturale esibisce rispetto a differenti tipi di codice e la irregolarità nella topologia e nelle prestazioni delle reti di interconnessione.

Per ovviare al problema della NP-completezza del mapping, sono state adottate diverse soluzioni di tipo euristico. In tal modo, si riesce a risolvere il problema del mapping in tempi contenuti, pur rinunciando alla certezza di raggiungere l'ottimo. Va comunque precisato che tali algoritmi, in genere, riescono a raggiungere buone approssimazioni della soluzione ottima con una frequenza elevata.

La presente nota tecnica è così organizzata: nella parte 1 vengono descritti alcuni dei principali lavori relativi al mapping in ambito di metacomputing; nelle parti 2-5 viene presentato un nuovo algoritmo di mapping del quale vengono in dettaglio presentati gli aspetti implementativi. Infine, le parti 6-10, dopo la presentazione di un caso di studio, mostrano le prestazioni offerte dal citato algoritmo.

Keywords: Heterogeneous Computing, Mapping, Metacomputing, Performance Evaluation,Task Scheduling.

1 Rassegna di algoritmi di mapping

Gli algoritmi della seguente rassegna sono stati scelti in base alla loro possibilità di utilizzo in ambito di metacomputing. Sono stati pertanto inclusi sia gli algoritmi espressamente concepiti per ambienti di calcolo eterogeneo, sia quelli che, a nostro avviso, possono essere adattati per tali ambienti.

1.1 Load Balanced Mincut

Un algoritmo euristico molto elegante è quello proposto da Selvakumar e Murthy [SM94]; nella sua versione base l'algoritmo è adatto a sistemi con due processori dove esistono risorse disponibili

su entrambi e risorse, definite *uniche*, disponibili su uno solo dei due processori. L'insieme dei due processori viene modellato per mezzo di un vettore che ne indica la performance e una matrice che, per ogni coppia (processore,risorsa), indica se la risorsa è disponibile su tale processore. L'insieme dei processi è modellato da 4 matrici: un vettore COMP che indica il costo computazionale di ciascun processo; una matrice simmetrica COMN che specifica i costi di comunicazione fra ogni coppia di processi; una matrice RUL che, per ogni coppia (processo,risorsa), indica il costo dell'uso di tale risorsa da parte del processo specificato nel caso in cui la risorsa sia disponibile sullo stesso processore dove è allocato il processo; una matrice RUR che associa ad ogni coppia (processo,risorsa) il costo dell'uso di tale risorsa da parte del processo indicato nel caso in cui la risorsa non sia disponibile sul processore dove viene allocato il processo.

L'algoritmo si divide in due fasi: nella prima viene costruito un grafo i cui nodi sono i processi e le risorse *uniche*. I processi sono connessi tra loro con archi il cui peso è dato dalla matrice dei costi di comunicazione; inoltre, ciascuna risorsa è connessa a ciascun processo con archi il cui peso è ottenuto mediante una opportuna composizione dei valori delle matrici RUL e RUR, in modo da tenere conto sia dei costi per l'uso di risorse locali, sia remote. Le risorse uniche disponibili sul primo processore sono connesse tra loro da archi di peso infinito e lo stesso vale per le risorse uniche disponibili sul secondo processore. Ai nodi processo è associato un peso che indica il loro costo computazionale, mentre a due dei nodi risorsa viene assegnato un peso fittizio che consente di tenere conto della diversità di performance tra i due processori.

La seconda fase dell'algoritmo cerca un taglio di costo minimo tale che la differenza tra il totale dei pesi dei nodi delle due porzioni di grafo (lo sbilanciamento del carico) non ecceda il peso del nodo di peso massimo. Il problema del Load Balanced Mincut (LBM) è NP-completo [SM94] e pertanto si adotta una tecnica euristica per ottenere una soluzione subottima in tempo ragionevole. In questo modo LBM tenta di minimizzare il tempo di completamento cercando di minimizzare simultaneamente le tre componenti che lo influenzano: sbilanciamento del carico, costi di comunicazione e costi di accesso a risorse uniche.

L'algoritmo può essere facilmente adattato al caso generale con un numero qualsiasi di processori con una tecnica *divide et impera*. La complessità nel caso con due processori è $O((n+u)^2)$ dove n è il numero di processi e u il numero di risorse uniche. Inoltre l'algoritmo esposto raggiunge l'ottimo nel 18.75% dei casi ed un valore minore di 1.1 volte l'ottimo nel 60% dei casi.

L'algoritmo è molto interessante ma risulta inadatto in situazioni in cui le macchine sono eterogenee non solo dal punto di vista della performance ma anche da quello architetturale: in tal caso risulta impossibile, in presenza di tipi diversi di codice, definire una graduatoria di performance tra le varie macchine che compongono il sistema.

1.2 Optimal Selection Theory

Uno dei primi approcci al problema del mapping su ambienti di calcolo eterogeneo è quello proposto da Freund [Fre89]; la tecnica funziona al meglio quando le necessità computazionali di una singola applicazione sono eterogenee e significative porzioni del codice sono loosely-coupled. Sotto tali condizioni, il problema del mapping viene formulato come un problema di programmazione intera la cui funzione obiettivo è di minimizzare il tempo di esecuzione della applicazione e il vincolo è dato dal costo complessivo delle macchine utilizzate dall'applicazione. Pur partendo dalla considerazione che le applicazioni di interesse sono tipicamente di tipo loosely-coupled, in questo approccio non vengono presi in considerazione i tempi di comunicazione fra i vari segmenti di codice costituenti l'applicazione originaria e ciò ne limita il campo di applicabilità. Una seconda limitazione proviene dal fatto che si cerchi una soluzione che sia ottima; il problema del mapping è formulato in maniera analoga al problema dello zaino [Ber90] e pertanto esiste una vasta letteratura su come trovare una soluzione ottima nel minor tempo possibile. D'altro canto il problema è NP-completo [Ber90] e quindi la ricerca di una soluzione ottima per problemi di dimensioni anche non eccessivamente grandi può richiedere tempi troppo lunghi. Ulteriori limitazioni sono date dall'aver assunto che il numero di macchine per ogni classe architetturale sia illimitato, che le sezioni di

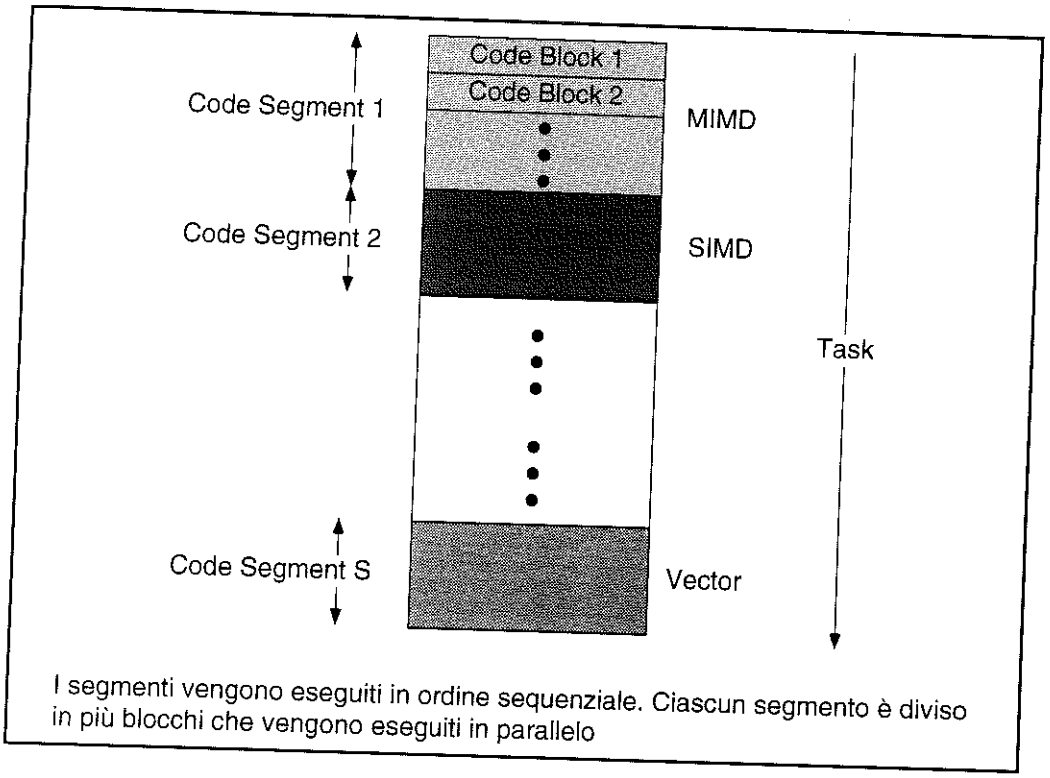


Figura 1: Formato dell'input per OST e AOST

codice parallelizzabile possano essere suddivise in moduli della stessa dimensione e che l'esecuzione dei moduli di una applicazione sia totalmente ordinata nel tempo (fig. 1). Quest'ultima assunzione, in particolare, impedisce che moduli indipendenti possano essere eseguiti in parallelo.

1.3 Augmented Optimal Selection Theory

Alcune limitazioni del modello OST sono state rimosse con l'introduzione di AOST. Una delle principali limitazioni di OST è l'assunzione che il numero di macchine per ogni classe architetturale sia illimitato. Pertanto, ogni code block è eseguito sulla macchina più adatta (scelta "ottima"). AOST, al contrario, rimuove il vincolo citato e tiene conto della performance dei code block nel caso di una scelta non ottima di macchine. Resta tuttavia il vincolo sulla esecuzione sequenziale dei vari segmenti. Ciò implica che viene considerato solo il parallelismo tra porzioni di codice dello stesso tipo, ossia non è possibile avere un "parallelismo eterogeneo", dove macchine di classi architetturali differenti lavorano in parallelo. Una ulteriore limitazione di AOST è data dal fatto di non considerare i dettagli legati al mapping dei code block sui nodi di una macchina parallela. Ad esempio, nella classe architetturale MIMD-DM possiamo annoverare sia macchine di tipo ipercubo, sia macchine di tipo mesh. È noto che i problemi di FFT (Fast Fourier Transform), per via dei pattern di comunicazione che utilizzano, si adattano meglio a strutture ad ipercubo piuttosto che mesh. Per questo tipo di problemi, oltre a quelli citati in precedenza, sono stati proposti ulteriori miglioramenti al modello AOST (descritti nel seguito), in modo da tentare di risolvere i vari problemi connessi al mapping su ambiente eterogeneo.

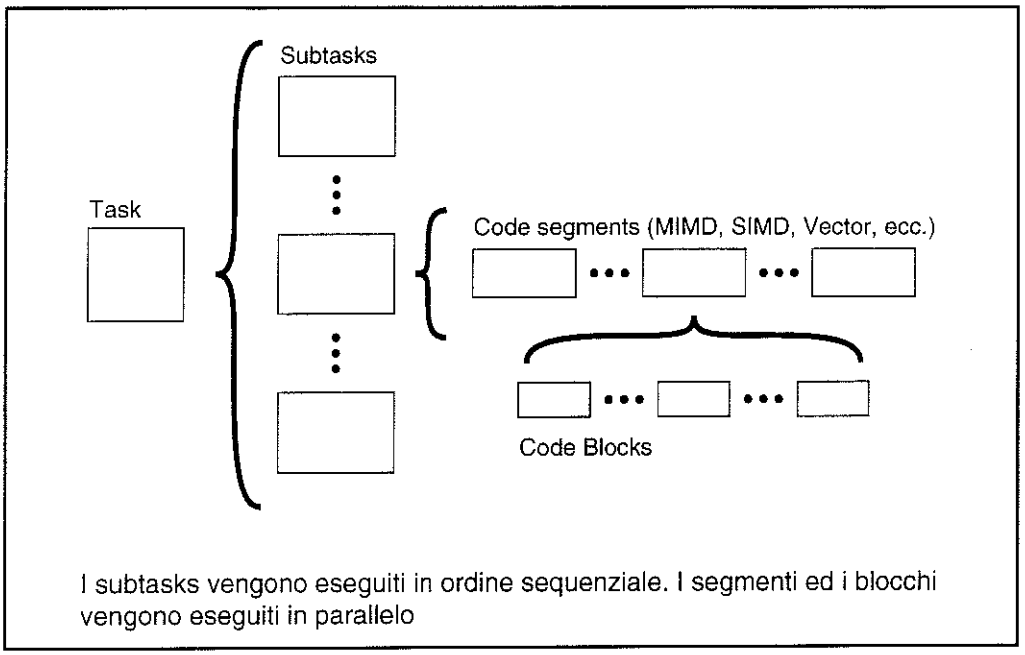


Figura 2: Formato dell'input per HOST

1.4 HOST e HCM

Heterogeneous Optimal Selection Theory (HOST) [CEKS93] è una estensione di AOST. In questo modello vengono presi in considerazione sia gli aspetti legati al mapping a basso livello, sui singoli nodi di una macchina parallela, sia quelli legati al parallelismo tra parti di codice di tipo differente ("parallelismo eterogeneo"). Pertanto, in questo modello (fig. 2), si riesce ad esplicitare un maggior grado di parallelismo rispetto al modello AOST. Una formulazione equivalente (fig. 3)del problema viene data per mezzo di Hierarchical Cluster-M (HCM). HCM è un modello che permette sia di descrivere un insieme di risorse di calcolo interconnesse (Cluster-M Representation), sia di specificare un algoritmo in maniera indipendente dall'architettura della risorsa di calcolo (Cluster-M Specification). HCM, inoltre, è un modello gerarchico e permette quindi di rappresentare gli oggetti a diversi livelli di astrazione: ad esempio, una macchina parallela può essere vista sia come un'unica entità, sia in termini dei singoli processori che la compongono. Il modello prende anche in considerazione la topologia delle interconnessioni ma non fornisce una visione quantitativa delle interconnessioni stesse; links allo stesso livello di clustering (stesso livello di astrazione) sono considerati uguali, anche se essi rappresentano oggetti fisici quantitativamente differenti (ad esempio una Ethernet da 10 Mbps ed una HiPPI da 800 Mbps).

Il mapping di una Cluster-M Specification su una Cluster-M Representation viene effettuato per mezzo di una euristica che opera "a livelli". Si parte dal livello di astrazione più alto (code segments) fino a giungere al più basso (instructions). Per una descrizione dettagliata delle regole di mapping si rimanda al riferimento [CEKS93]. Non vengono fornite le prestazioni di tale algoritmo di mapping.

1.5 SOS e Cluster-M SOS

"Mentre HOST è una prova dell'esistenza di una soluzione ottima al problema del mapping di un task eterogeneo su un insieme eterogeneo di macchine, Synthesis of Systems (SOS) è una tecnica per ottenere tale soluzione ottima" [DEP94]. SOS comporta la formulazione e la risoluzione di

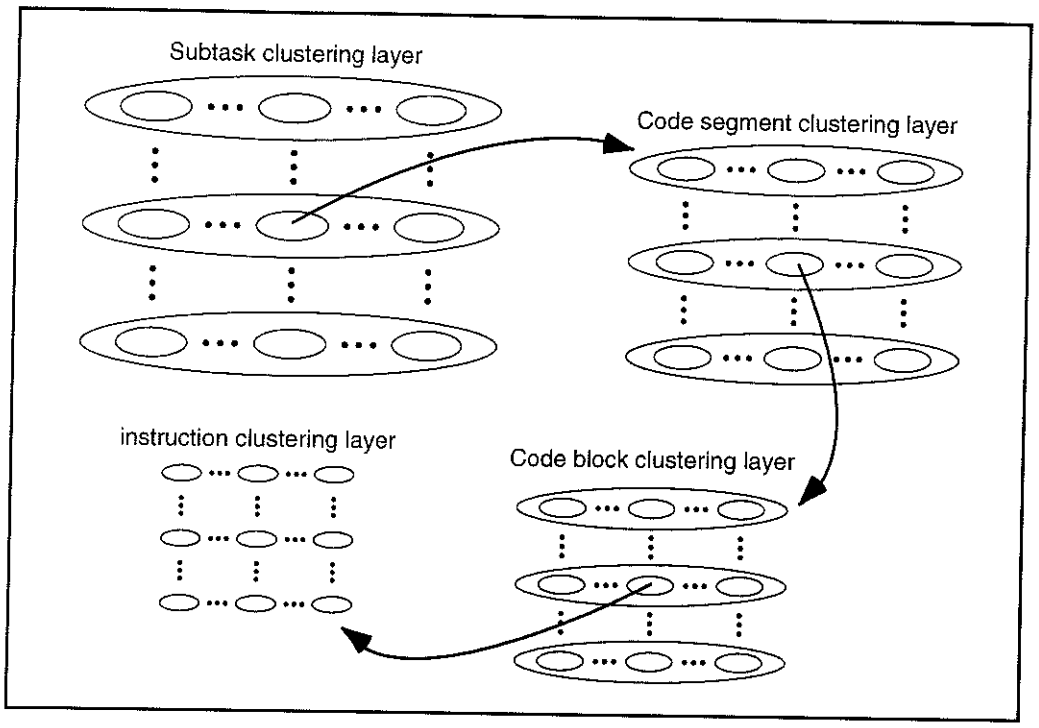


Figura 3: Specifica dell'input per HOST per mezzo di Hierarchical Cluster-M

un problema di programmazione mista lineare-intera. Tale problema comprende una funzione obiettivo ed una serie di vincoli. La funzione obiettivo può essere la minimizzazione del costo del sistema oppure la massimizzazione della performance. I vincoli comprendono: vincoli sulla correttezza (sincronizzazione e precedenze) e vincoli sul tempo e sul costo imposti a piacimento dal programmatore. Il problema di programmazione mista viene risolto all'ottimo utilizzando uno dei tanti pacchetti disponibili.

Il grave difetto di SOS è che la risoluzione del problema di programmazione mista richiede tempi troppo elevati [DEP94]. Per tale ragione è stata escogitata una soluzione alternativa. Utilizzando una rappresentazione "a livelli" del problema, per mezzo di Cluster-M, il problema originario viene suddiviso in una serie di sottoproblemi di dimensioni inferiori. I singoli sottoproblemi sono anch'essi modellati per mezzo di problemi di programmazione mista, ma le loro dimensioni limitate consentono di ottenere una soluzione in tempi inferiori.

Purtroppo, come accennato in precedenza, non esistono riferimenti ai costi di comunicazione; un secondo problema è dato dalla velocità non proprio elevata dell'algoritmo di mapping. Un problema composto da 8 processi e 8 processori richiede un tempo di 14.3 secondi su una macchina con prestazioni di picco pari a 100 MIPS [DEP94]. Un tempo simile ci sembra molto alto, soprattutto se confrontato con i tempi richiesti dall'algoritmo da noi sviluppato.

1.6 Gli algoritmi della famiglia OST

In questo paragrafo riassumiamo brevemente le differenze esistenti tra i vari algoritmi della serie OST.

OST Una prima limitazione di OST è data dall'assumere che il numero di macchine per ogni classe architetturale sia illimitato. Inoltre, si assume che esiste sempre un insieme di macchine che corrisponde esattamente al set di codici da eseguire. In questo modo, ogni code-block può essere

assegnato all'architettura che meglio si presta ad eseguirlo e non sono necessarie scelte "di ripiego". Inoltre, si assume che ciascun code-segment possa essere diviso in blocchi della stessa dimensione, dando luogo a partizioni perfettamente bilanciate.

AOST AOST tiene conto della performance dei vari code-segment nel caso di una scelta non ottima di macchine, assumendo che il numero di macchine per ogni classe architetturale sia limitato. Una limitazione di AOST è data dall'assumere che l'esecuzione dei code-segment sia totalmente ordinata nel tempo e che il parallelismo sia possibile solo tra porzioni di codice dello stesso tipo, ossia solo tra code-block. Inoltre, non si tiene conto delle variazioni di performance dovute a differenti strategie di mapping dei vari code-block all'interno della singola macchina parallela.

HOST HOST estende AOST in due direzioni. In primo luogo vengono considerati anche i dettagli relativi al mapping dei vari code-blocks sui singoli nodi di una macchina parallela. Infatti, HOST considera aspetti quali la topologia di interconnessione delle singole macchine parallele e i patterns di comunicazione tra i vari code-block, al fine di avere un modello il più possibile preciso dei tempi di esecuzione. In secondo luogo, è possibile considerare il parallelismo presente tra porzioni di codice di tipo differente. Infatti, in HOST, i code-segment vengono eseguiti in parallelo (mentre in OST e AOST vengono eseguiti in sequenza), in modo che macchine architetturalmente differenti possano lavorare contemporaneamente.

1.7 One Level Reach-Out Greedy

OLROG è un algoritmo euristico che si avvale di una rappresentazione del problema molto dettagliata [LP92] ed è il migliore di una serie di algoritmi costruiti a partire da un semplice algoritmo Greedy. La computazione viene rappresentata per mezzo di un grafo diretto aciclico (DAG) e ad ogni nodo sono associate le informazioni di eterogeneità mentre agli archi è associato un valore che indica la quantità stimata di banda richiesta. L'insieme delle macchine viene modellato per mezzo di un grafo completo dove ai nodi sono associate le informazioni riguardanti i processori e agli archi sono associati i dati sul canale virtuale che connette i due nodi alle estremità dell'arco stesso (banda del canale, necessità di conversione di rappresentazioni, canale diretto/indiretto, ecc.).

L'algoritmo OLROG si compone di due fasi: nella prima il grafo dei processi viene suddiviso in sottografi indipendenti tali che tutti i processi di un dato sottografo possono essere eseguiti in parallelo e le comunicazioni avvengono tra due sottografi indipendenti vicini (fig. 4). Nella seconda fase i sottografi vengono esaminati uno alla volta: scelto un processo, esso viene assegnato al processore che prima riesce a completarlo, tenendo presenti le informazioni sul costo computazionale del processo, la performance della macchina rispetto al tipo di codice del processo scelto, i costi di comunicazione ed i tempi di attesa per il completamento di eventuali altri task assegnati al processore scelto. Prima di arrivare all'algoritmo OLROG sono stati ideati altri schemi di mapping denominati CLA, CLB CLC e CLD (CutLong A, ecc.) [LP94] che costituiscono una variazione dell'algoritmo base Greedy. La fase di suddivisione del grafo è la stessa vista per OLROG; nella seconda fase, dato un processo, si sceglie un processore in base al minimo tempo di completamento. A questo punto si cerca di eliminare l'arco più costoso risultante dal mapping iniziale: il tempo di completamento ottenuto con il task corrente assegnato al processore inizialmente scelto viene confrontato con il tempo di completamento ottenuto assegnando il task corrente al processore con cui comunica di più. Le differenze tra i vari algoritmi CutLong consistono nella differente gestione del pool di processori disponibili per il mapping iniziale (round robin, lista a priorità in base al carico assegnato, ecc.).

Mentre gli algoritmi CutLong ottimizzano separatamente il tempo di completamento ed i costi di comunicazione, l'algoritmo OLROG considera le due variabili in un solo colpo e pertanto, a fronte di un maggior costo computazionale, è in grado di fornire risultati migliori [LP94]. Una caratteristica comune agli algoritmi CL e OLROG è il vincolo sulle modalità di interazione tra i processi che compongono una applicazione: il grafo delle dipendenze deve essere aciclico e pertanto

non è possibile modellare una computazione composta da processi che interagiscono con frequenza non fissata (esempio: schemi client-server).

1.8 Clustering

Un algoritmo di mapping adatto a grafi di processi qualsiasi è quello proposto da Bowen et al. [BNG92]. In tale algoritmo non si fanno assunzioni sull'ordine con cui i processi vengono eseguiti e pertanto il grafo dei processi non è aciclico; ad ogni arco viene assegnato un peso che indica la quantità di informazione scambiata fra i processi connessi da tale arco.

L'insieme dei processori viene modellato per mezzo di un grafo completo ai cui archi è associato il costo della comunicazione tra i nodi interconnessi, sia che esista un link fisico tra di essi, sia che i nodi non siano fisicamente interconnessi ma esista un percorso tra i due. Nel secondo caso il costo deve tenere conto anche degli overhead dovuti al routing, buffering, ecc. Ad ogni nodo è associata una coppia di valori che indica il massimo e minimo carico ammessi (massimo e minimo numero di processi) per tale nodo.

Data questa rappresentazione del problema, l'algoritmo proposto cerca di minimizzare il costo di comunicazione tra i processi rispettando i vincoli sul carico assegnato ai processori. Poiché il problema è formulato come un assegnamento quadratico, esso risulta intrattabile e pertanto l'algoritmo proposto si avvale di una euristica che permette di ottenere risultati subottimi in tempi ragionevoli.

In una prima fase, i grafi dei processi e dei processori vengono sottoposti ad un processo di *clustering* gerarchico (fig. 5): i nodi connessi da archi *simili* (la differenza percentuale fra i loro costi deve essere inferiore ad un parametro di soglia T) vengono sostituiti da un unico nodo che li rappresenta. Dopo un primo passo di *clustering* il grafo sarà composto da un numero minore di nodi che rappresentano gli insiemi connessi da archi *simili*. Al grafo risultante viene applicato di nuovo il procedimento di *clustering* ripetutamente fino ad ottenere un grafo con un solo nodo che rappresenta tutto il grafo di partenza. Il *clustering* induce una struttura ad albero sul grafo dove le foglie di tale albero sono proprio i nodi del grafo e la radice è l'unico nodo del grafo risultante all'ultimo passo di *clustering*. La struttura ad albero permette di vedere il grafo a vari livelli di astrazione: da oggetto unico (la radice) fino ai singoli nodi (le foglie). La seconda fase dell'algoritmo si occupa di mappare l'albero dei processi sull'albero dei processori rispettando i vincoli sul carico. La minimizzazione del costo di comunicazione avviene in maniera implicita grazie alla fase di clustering: infatti, partendo dalla radice, i cluster di processi che comunicano meno pesantemente vengono assegnati a cluster di processori connessi da linee più costose mentre i cluster di processi che comunicano di più (in basso nell'albero) vengono assegnati a cluster di processori connessi da canali poco costosi.

La complessità della fase di clustering è $O((3e + (d+1)n) \log n)$ dove d è il grado dei nodi del grafo, n il numero dei nodi ed e il numero degli archi; per valori di d piccoli e costanti e per grafi sparsi, la complessità si riduce a $O(n \log n)$. La complessità della fase di allocazione è $O(n)$ dove n è il numero di nodi del grafo dei processori. L'algoritmo proposto, pertanto, è molto veloce e riesce a produrre risultati molto vicini all'ottimo.

Uno dei principali svantaggi dell'algoritmo è che l'eterogeneità del sistema di calcolo viene considerata solo a livello dei costi di comunicazione per le reti e della differente performance delle macchine (possibilità di specificare limiti di carico differenti). Un altro punto debole è la cosiddetta *unit workload assumption* [BNG92] ossia il fatto di considerare i processi tutti uguali dal punto di vista del carico che comportano per un processore: con questa semplificazione un processo che impegna a fondo le risorse di calcolo (es.: risoluzione di sistemi lineari) ed un processo che le utilizza poco (es.: gestione terminale) sono considerati uguali dal punto di vista del carico indotto su un processore.

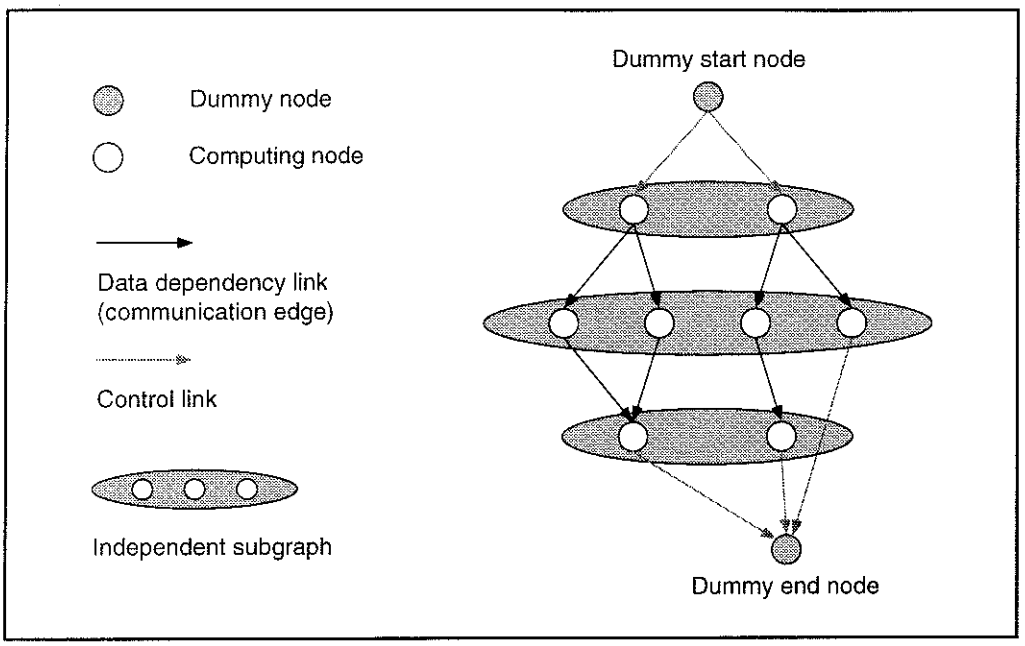


Figura 4: Esempio di come un grafo viene partizionato in sottografi indipendenti

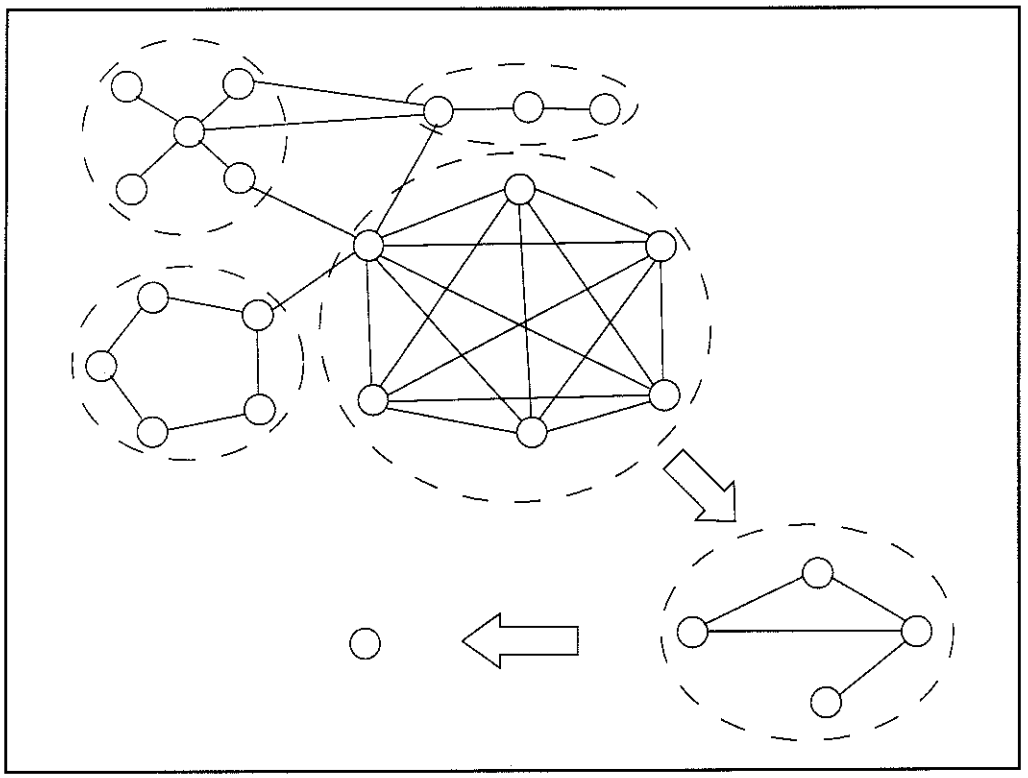


Figura 5: Clustering di un grafo

2 Un nuovo algoritmo di mapping

Negli ultimi anni sono stati realizzati svariati algoritmi di mapping, specifici per ambienti di calcolo eterogenei. Una caratteristica comune a molti di tali algoritmi è il modo in cui viene modellata la computazione. L'applicazione da *mappare* deve essere partizionata in moduli e fra questi moduli deve esistere un ordinamento parziale che vincoli l'ordine di esecuzione. Ciascun modulo può iniziare la sua esecuzione solo dopo che tutti i predecessori (nell'ordinamento parziale stabilito) hanno terminato la propria. Come conseguenza immediata di queste assunzioni, si ha che i dati possono viaggiare in una sola direzione, ossia un modulo può ricevere dati solo dai predecessori e inviarli solo ai successori. Da quanto detto, si comprende chiaramente come la migliore e più naturale rappresentazione per una computazione con i vincoli citati, sia quella di un grafo diretto aciclico (DAG).

Un DAG è un particolare grafo orientato in cui non esistano cicli. Ogni nodo del grafo rappresenta un modulo. Gli archi rappresentano le dipendenze sull'ordine di esecuzione dei moduli. Agli archi viene di solito associato un peso che indica la quantità di dati che ciascun modulo invia ad ogni successore. Questi valori vengono utilizzati per stimare i tempi di comunicazione.

Nonostante i vincoli che il modello impone, esistono diversi problemi di interesse che possono essere modellati per mezzo di un DAG. Il vantaggio è dato dal fatto che ciascun algoritmo di mapping riesce a sfruttare i vincoli per migliorare le proprie prestazioni: alcuni per arrivare ad una soluzione subottima in minor tempo, altri per ottenere una soluzione più vicina all'ottimo. Tuttavia, l'introduzione di questi vincoli, di solito, non è in grado di abbassare la complessità del problema fino a renderla polinomiale.

Esistono tuttavia diverse applicazioni che non possono essere modellate per mezzo di un DAG. Tra queste applicazioni, citiamo quelle in cui il flusso dei dati sia bidirezionale, ad esempio, simulazioni dell'evoluzione di galassie. Per il tipo di applicazioni aventi caratteristiche simili a quella appena menzionata, esistono alcuni algoritmi di mapping ma questi non tengono conto dell'eterogeneità architetturale delle risorse di calcolo.

Modificando opportunamente uno di tali algoritmi, è stato possibile adattarlo ad un ambiente eterogeneo. L'algoritmo realizzato rappresenta una soluzione del problema del mapping per una classe specifica di applicazioni. A nostro avviso, per ottenere i migliori risultati, è necessario disporre di più algoritmi di mapping, ciascuno adatto ad una classe di applicazioni. In tal modo è possibile sfruttare i vincoli e le caratteristiche che ogni specifica classe di applicazioni presenta.

3 Descrizione generale

L'algoritmo presentato ricalca fedelmente quello di Bowen et al. (§ 1.8 e § [BNG92]) salvo le piccole modifiche necessarie per tenere conto delle informazioni sull'eterogeneità dei processi e dei processori. In un primo passo, i grafi dei processi e dei processori vengono sottoposti a *clustering* gerarchico, in modo da raggruppare in *cluster* processi che comunicano frequentemente e processori topologicamente vicini. Il processo di *clustering* genera due alberi, uno per i processi ed uno per i processori. Nella seconda fase, l'albero dei processi viene *mappato* sull'albero dei processori, mantenendo il carico assegnato ad ogni processore entro un intervallo precedentemente specificato e mantenendo il valore di *affinity* (§ 3.2) sopra una soglia precedentemente stabilita.

Il limite superiore sul carico assegnato ad un processore modella la soglia che appare nelle curve throughput/carico per i sistemi di calcolo: il throughput cresce linearmente con il carico fino ad un valore di soglia. In seguito aumentando il carico, l'incremento di throughput è modesto e per valori di carico ancora maggiori, le prestazioni degradano. Questo fenomeno viene comunemente chiamato *thrashing* (fig. 6). Pertanto, mantenendo il carico entro questo valore di soglia, si riesce a massimizzare il throughput globale. D'altro canto, l'imposizione di un limite inferiore al carico, consente di evitare che qualche processore rimanga inoperoso o quasi. I vincoli sul carico, inoltre, permettono di distribuire il carico in modo da abbassare i tempi di completamento. Fissando un

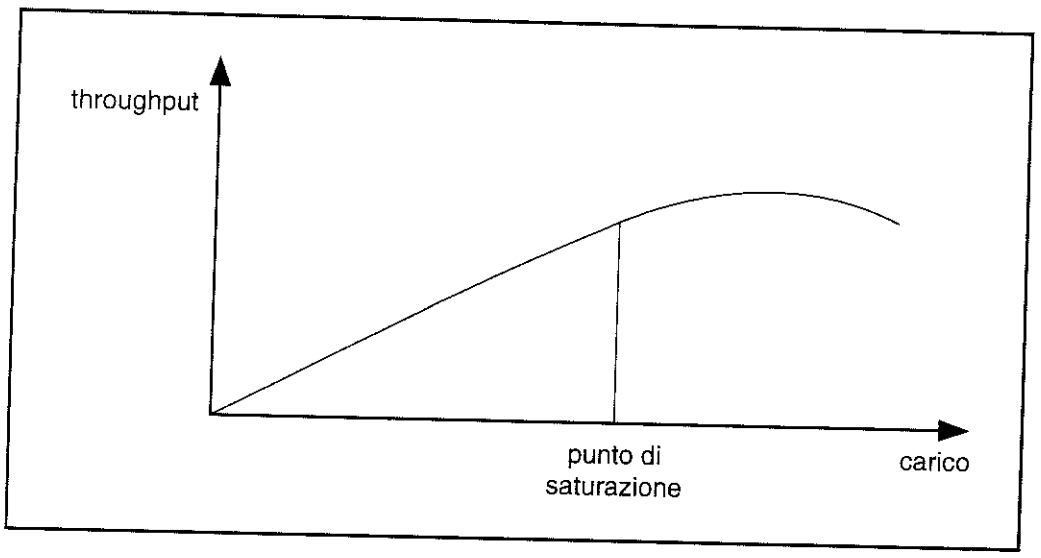


Figura 6: Una tipica curva del throughput in funzione del carico

valore di minimo pari a 1 per tutti i processori, si ottiene un assegnamento in cui nessun nodo è scarico, ottenendo un primo livello di bilanciamento del carico. Inoltre, ponendo i vincoli di massimo a valori non troppo elevati (ossia evitando che la somma dei valori di massimo carico ecceda di molte volte il numero complessivo di processi), si impedisce che un processore venga sovraccaricato e rallenti l'intera applicazione.

Il vincolo sulla *affinity* permette di modellare la differente affinità tra tipi di codice e classi architetturali. Tenendo questo valore sopra una certa soglia, si evita che processi di un certo tipo vengano assegnati a macchine che non sono in grado di eseguirli con una efficienza adeguata.

3.1 Grafo dei processi

Il grafo dei processi è stato arricchito con informazioni riguardanti il tipo di del codice. Ad ogni nodo, rappresentante un processo, è stato associato un valore che indica il tipo di codice (sequenziale, vettoriale, dataparallel, ecc.) del processo associato. Inoltre, per aumentare la flessibilità dell'algoritmo, per ogni nodo è possibile specificare la macchina a cui dovrà essere assegnato. Questa possibilità di forzare il mapping di alcuni processi può essere utile nel caso in cui esistano risorse disponibili su una sola macchina. Ad esempio, un processo che legge e scrive files dovrà essere assegnato alla macchina su cui tali files sono memorizzati.

3.2 Grafo dei processori

Il grafo dei processori è stato modificato aggiungendo, ad ogni nodo, informazioni riguardanti la classe architetturale (sequenziale, SIMD, vettoriale, ecc.) del processore rappresentato ed un valore compreso tra 0 e 1 denominato *affinity*. Per ogni processore, il valore di *affinity* indica il grado di libertà che l'algoritmo di mapping ha nell'assegnare processi di vario tipo al processore in questione. Un valore pari a 0 indica che qualsiasi tipo di processo può essere assegnato. Un valore pari a 1 indica che possono essere assegnati solo i processi il cui tipo di codice corrisponde esattamente alla classe architetturale del processore.

3.3 Matrice di affinità

La matrice di affinità costituisce un ulteriore input dell'algoritmo di mapping proposto, oltre ai grafi dei processi e dei processori. Questa matrice associa ad ogni coppia (tipo di codice, classe architetturale) un valore di affinità compreso tra 0 e 1. In base a tale matrice è possibile scegliere, per un dato processo, la migliore architettura e, se non disponibile, cercare la migliore tra quelle a disposizione. Il modo migliore di determinare i valori di tale matrice è quello di effettuare una fase di analisi preventiva del codice (**code type profiling**) per determinare i tipi di codice presenti nelle applicazioni utilizzate. Effettuando un **benchmark analitico** è possibile stabilire le prestazioni delle varie architetture sui tipi di codice individuati e, in base a tali risultati, assegnare valori opportuni alla matrice di affinità [GY93].

4 Algoritmo di clustering

Gli algoritmi di clustering possono essere suddivisi in due categorie:

Algoritmi agglomerativi. Inizialmente, si considera il grafo come composto da N cluster di 1 elemento ciascuno. Ad ogni passo di clustering, vengono uniti i due cluster più simili. Dopo $N-1$ passi, si ottiene un solo cluster.

Algoritmi divisivi. Inizialmente, il grafo è considerato come un singolo cluster e, successivamente, viene diviso fino a quando i cluster ottenuti non hanno la dimensione desiderata.

L'algoritmo utilizzato (fig. 7) è un particolare algoritmo agglomerativo che usa i pesi sugli archi come criterio di similarità. L'input dell'algoritmo è costituito da un grafo pesato $G = (V, F, E)$, dove V è l'insieme dei nodi, F è l'insieme degli archi ed E è l'insieme dei pesi associati agli archi.

Il primo passo è di copiare G in un grafo di lavoro G'. Poiché l'algoritmo usato è di tipo agglomerativo, bisogna effettuare una serie di passi intermedi in cui vengono raggruppati in cluster i nodi con la maggiore affinità, determinata dal peso sugli archi. Durante ogni passo intermedio, ciascun nodo deve essere messo in un cluster. Una volta che un cluster intermedio è stato formato, tutti i suoi nodi tranne uno vengono rimossi dal grafo G'. Questo singolo nodo funge da rappresentante dei nodi rimossi, durante le successive iterazioni dell'algoritmo. Un passo intermedio termina quando tutti i nodi sono stati inseriti in un cluster e l'algoritmo termina quando V' contiene un solo nodo.

Un vettore (c) viene utilizzato per marcare i nodi che hanno subito il clustering. Prima dell'inizio di un passo intermedio, $c(v_i)$ viene posto a 0 per tutti i nodi in V'. Mentre i nodi vengono raggruppati in cluster, nel passo intermedio, alcuni $c(v_i)$ vengono posti a 1, mentre altri nodi vengono completamente rimossi da V'. Il passo intermedio termina quando tutti i nodi in V' hanno $c(v_i)$ pari a 1.

La prima operazione di un passo intermedio è di selezionare un nodo *pivot*. Questo nodo è quello adiacente all'arco di maggior costo, nel grafo dei processi, mentre è quello adiacente all'arco di costo minore nel grafo dei processori. Poiché esistono almeno due nodi che soddisfano alla condizione citata, vengono applicate le seguenti regole per ridurre il loro numero a 1:

- Scegli i nodi con il maggior numero di archi incidenti.
- Scegli il nodo di numero inferiore.

L'operazione successiva è di selezionare tutti i vicini del pivot che non appartengono già ad un cluster (funzione **Rank_Neighbors**). Questi nodi vengono ordinati in senso discendente secondo i pesi degli archi che li connettono al pivot. L'insieme di tutti i vicini ed il pivot vengono considerati per essere raggruppati in un cluster. Il criterio utilizzato per la scelta del pivot assicura che il sottografo che viene ora considerato per essere sottoposto a clustering, in effetti contiene l'arco di peso maggiore.

Algoritmo di clustering

Cluster(G)
  $G' = G$
  do while$|V'| > 1$
    $c(v') = 0 \; \forall \, v' \in V'$
    do while$\exists \, v' \in V'$ **tale che** $c(v') = 0$
      Clear C
      Scegli un pivot $v_p \in V'$ **tale che** $c(v_p) = 0$
        Scegli i nodi con il massimo e_{pj} **tale che** $v_j \in V' \; \wedge \; c(v_j) = 0$
        Scegli i nodi con il massimo numero di archi incidenti
        Scegli il nodo di numero più basso
      Rank_Neighbors(k, v_p)
      Aggiorna G'
        $c(v_p) := 1$
        $\forall \, v_i \in C \; , \; v_i \neq v_p$ **rimuovi** v_i **da** V'
        if $(v_i, v_j) \in F' \; \wedge \; v_i, v_j \in C$ **then rimuovi** (v_i, v_j) **da** F'
        $\forall \, (v_i, v_j) \in F'$ **tale che** $v_i \in C \; \wedge \; v_j \notin C \; \wedge \; v_i \neq v_p$ **do**
          Rimuovi (v_i, v_j) **da** F'
          if $(v_p, v_j) \notin F'$**then aggiungi** (v_p, v_j) **a** $F' \; ; \; e_{pj} := e_{ij}$
          if $(v_p, v_j) \in F'$**then** $e_{pj} := e_{pj} + e_{ij}$
      Registra v_p **e i nodi in** C **come appartenenti allo stesso cluster**
        Genera un nuovo nodo R **nell'albero**
        Rendi figli di R **tutti i nodi dell'albero corrispondenti ai nodi in** C
        Aggiorna R **in modo da rappresentare tutti i nodi figli**
    end
  end

Rank_Neighbors(k, v_p)
  $Q = \{e_{pj} \mid v_p, v_j \in F' \; \wedge \; c(w_j) = 0\}$
  Ordina Q **in senso discendente**
  Elimina i nodi che oltrepassano la soglia
    $Q' = \{Q(1), \ldots, Q(t)\}$ **dove:**
      $\dfrac{Q(i) - Q(i+1)}{Q(i)} < T \; \forall \, i = 1, \ldots, t-1 \; \wedge \; \dfrac{Q(t) - Q(t+1)}{Q(t)} \geq T$
    $C := C \cup V_q$ **dove** $V_q = \{v_j \mid e_{pj} \in Q'\}$
  if $k > 1$**then Rank_Neighbors**$(k-1, v_i) \quad \forall \, v_i \in V_q$

Figura 7: L'algoritmo di clustering

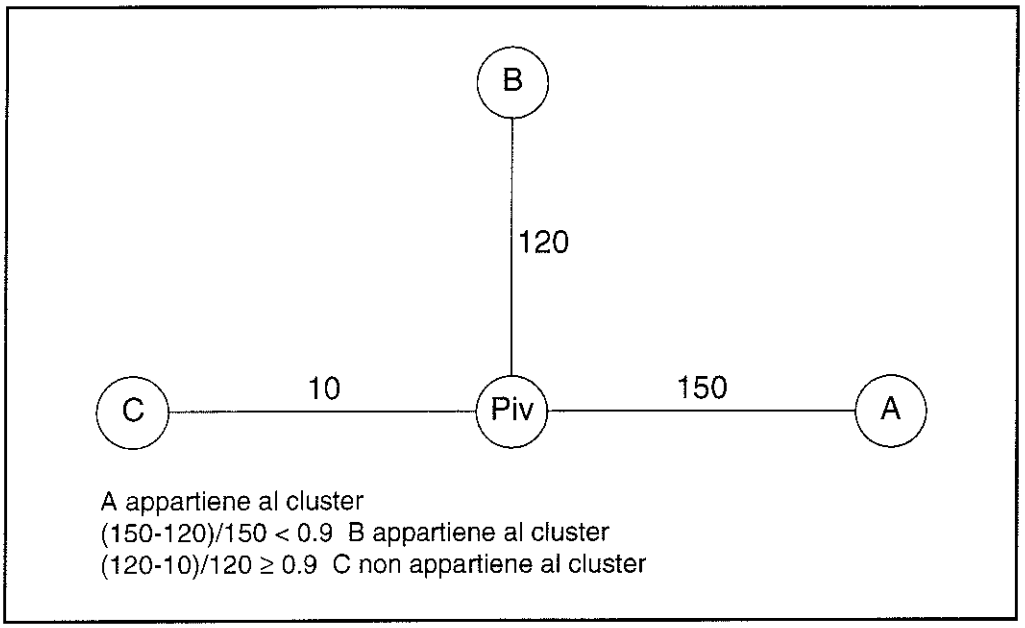


Figura 8: Esempio della funzione del parametro di soglia T

Un valore di soglia T viene utilizzato per selezionare i nodi candidati all'inclusione in un cluster. Questo parametro consente di ottenere cluster dove gli archi abbiano pesi quasi uguali. La funzione **Rank_Neighbors** viene chiamata ricorsivamente per includere i vicini dei vicini fino ad una profondità k. Un piccolo valore di k comporta la formazione di molti cluster di piccole dimensioni ed una maggiore profondità dell'albero risultante. Valori di k elevati danno luogo alla formazione di pochi grandi cluster, "appiattendo" l'albero generato. In figura 8 è mostrato un esempio dell'effetto del parametro di soglia T. Due vicini del pivot sono connessi da archi costosi mentre il terzo è connesso da un arco con costo relativamente basso. Utilizzando un valore di soglia pari a 0.9, il nodo C verrebbe scartato. In seguito vengono eseguiti gli aggiornamenti per il passo corrente dell'algoritmo. Il nodo pivot viene marcato ponendo $c(v_p)$ pari a 1. Per gli altri nodi del cluster, questa operazione non è necessaria, in quanto essi vengono completamente rimossi dall'insieme V'. Inoltre, tutti gli archi che connettono nodi interni al cluster vengono rimossi da F'. Poiché il pivot deve rappresentare tutti i nodi cancellati per il resto dell'algoritmo, sono necessari alcuni aggiustamenti. Tutti gli archi che vanno da nodi esterni al cluster a nodi interni cancellati devono essere sostituiti da archi che puntano al pivot. Ci sono due casi da considerare, come mostrato in figura 9. Nel primo caso, un nodo esterno al cluster (A) punta sia ad un nodo cancellato sia al pivot. L'arco (A, C) deve essere rimosso e il costo dell'arco $(A, pivot)$ deve essere incrementato per rappresentare entrambi gli archi. Nel secondo caso, un nodo esterno al cluster (B) non punta al pivot e pertanto un nuovo arco$(B, pivot)$ deve essere aggiunto per compensare la cancellazione di (B, D). Alla fine, viene costruito un nuovo nodo dell'albero e tutti i nodi contenuti in C diventano figli del nuovo nodo creato. Successivamente, il nodo dell'albero appena creato viene aggiornato in modo da rappresentare tutti i nodi figli.

5 Algoritmo di allocazione

L'algoritmo di allocazione è la parte che ha subito le modifiche maggiori rispetto all'algoritmo descritto in [BNG92]. Infatti, la necessità di considerare il parametro di *affinity*, ha richiesto la riscrittura della funzione **Select**, che si occupa di scegliere il processo da assegnare ad un dato

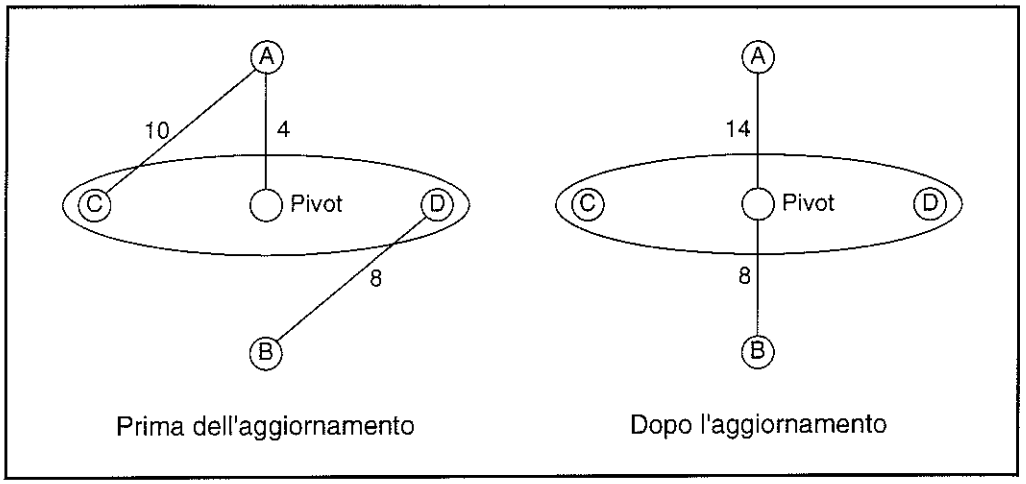


Figura 9: Esempio di aggiornamento di un grafo

processore.

Anche gli alberi (dei processi e dei processori) sono stati modificati per considerare le differenti tipologie di architetture e i vari di tipi di codice.

5.1 Alcune definizioni

5.1.1 Albero dei processi

Con r indicheremo un generico nodo dell'albero dei processi, mentre $n(r)$ indica il numero di figli del nodo r (0 se r è una foglia), e $D(r)$ indica l'insieme dei figli del nodo r ($D(r) = \{r_1, \ldots, r_{n(r)}\}$). $w(r)$ è un vettore che, per ogni tipo di codice, indica il numero di processi che compongono il nodo r. Ad esempio:

$w(r) =$

Seq	VLIW	DP	DF	Vector
5	0	1	0	0

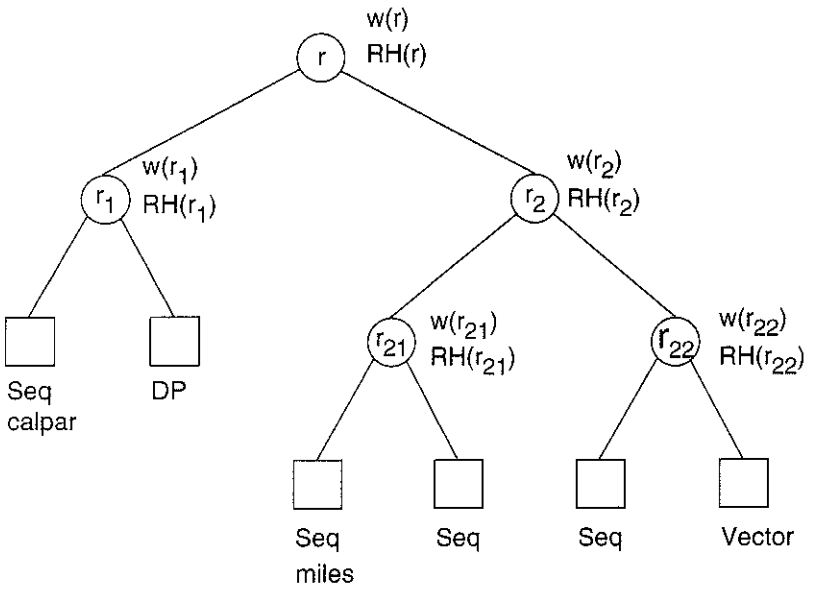
indica che il cluster r è composto da 5 processi di tipo Sequenziale e da 1 processo di tipo DataParallel. $RH(r)$ è un insieme che indica quali host sono necessari affinché il cluster di processi r possa essere eseguito. Ciò è utile nel caso in cui alcuni processi debbano accedere a risorse che sono situate solo su uno specifico host: in tal caso basterà associare ad ogni processo r' un insieme $RH(r')$ che indica il processore dove si trovano le risorse richieste da r'. In figura 10 è mostrato un esempio di albero dei processi.

5.1.2 Albero dei processori

In maniera analoga a quanto detto prima, con R, $n(R)$ e $D(R)$ indicheremo rispettivamente un generico nodo dell'albero dei processori, il numero di figli del nodo R e l'insieme dei figli del nodo R ($D(R) = \{R_1, \ldots, R_{n(R)}\}$). $m(R)$ e $M(R)$ sono due vettori che, per ogni tipo di architettura, indicano il minimo e il massimo carico (numero di processi) da assegnare al cluster di processori R. Ad esempio:

	WS	SIMD	VLIW
$m(R) =$	5	0	0
$M(R) =$	10	2	0

r = generico nodo dell'albero dei processi
$n(r)$ = numero di figli del nodo r
$w(r)$ = vettore dei processi rappresentati dal nodo r
(carico rappresentato da r)
$RH(r)$ = (Requested Hosts) insieme degli hosts richiesti da r
$D(r)$ = insieme dei figli di r ($D(r) = \{r_1, \ldots, r_{n(r)}\}$)



	Seq	VLIW	DP	DF	Vector
$w(r_1) =$	1	0	1	0	0
$w(r_{21}) =$	2	0	0	0	0
$w(r_{22}) =$	1	0	0	0	1
$w(r_2) =$	3	0	0	0	1
$w(r) =$	4	0	1	0	1

$RH(r_1) = \{\text{calpar}\}$
$RH(r_{21}) = \{\text{miles}\}$
$RH(r_{22}) = \emptyset$
$RH(r_2) = \{\text{miles}\}$
$RH(r) = \{\text{calpar,miles}\}$

Figura 10: Un esempio di albero dei processi

indica che il cluster di processori R deve eseguire da 5 a 10 processi su hardware di tipo WorkStation, da 0 a 2 processi su hardware di tipo SIMD e nessun processo su hardware VLIW. $AH(R)$ è un insieme che indica da quali host è formato il cluster di processori R: questo consente di esprimere i vincoli per quei processi che necessitano di risorse disponibili solo su alcuni host. $c(R)$ è un vettore che indica, durante la fase di allocazione e per ciascuna classe architetturale, il numero totale di processi correntemente assegnati al cluster R mentre $RA(R)$ è un insieme che indica i cluster di processi assegnati al cluster di processori R (esempio: $c(R') = [5, 0, 15]$, $RA(R') = \{r_1, r_3, r_4, r_5\}$). $V(R)$ indica la *violazione* del nodo R, ossia la differenza percentuale fra il carico minimo e quello correntemente assegnato durante la fase di allocazione. Nella versione originaria dell'algoritmo $V(R)$ è definito come segue:

$$V(R) = \frac{m(R) - c(R)}{m(R)}$$

Data questa definizione, segue che $V(R)$ gode della seguente proprietà:

$$V(R) > 0 \Leftrightarrow c(R) < m(R)$$

Nel nostro caso $c(R)$ e $m(R)$ non sono scalari ma vettori. Pertanto si è reso necessario determinare una nuova funzione per calcolare la violazione, che godesse della proprietà appena citata. Nelle definizioni che seguono, il modulo di un vettore ($|v|$) è definito come la somma delle sue componenti.

$$d_+ = \max(m(R) - c(R), 0)$$

$$d_- = \min(m(R) - c(R), 0)$$

$$V(R) = \begin{cases} \frac{|d_+|}{|m(R)|} & se\ |d_+| > 0 \\ \frac{|d_-|}{|m(R)|} & altrimenti \end{cases}$$

$Aff(R)$ è un valore assegnato a ciascun processore e indica la minima *affinity* ammissibile per il processore R. L'*affinity* è un valore che indica quanto buona è la corrispondenza fra la classe architetturale di R e il tipo di codice dei processi ad esso assegnati. Un esempio di albero dei processori è mostrato in figura 11.

5.1.3 Strutture ausiliarie

Nel corso dell'allocazione, i processori vengono suddivisi in due insiemi, S_v e S_a, definiti come segue:

$$\begin{aligned} S_v &= \{R \mid V(R) > 0\} \\ S_a &= \{R \mid V(R) \leq 0\} \end{aligned}$$

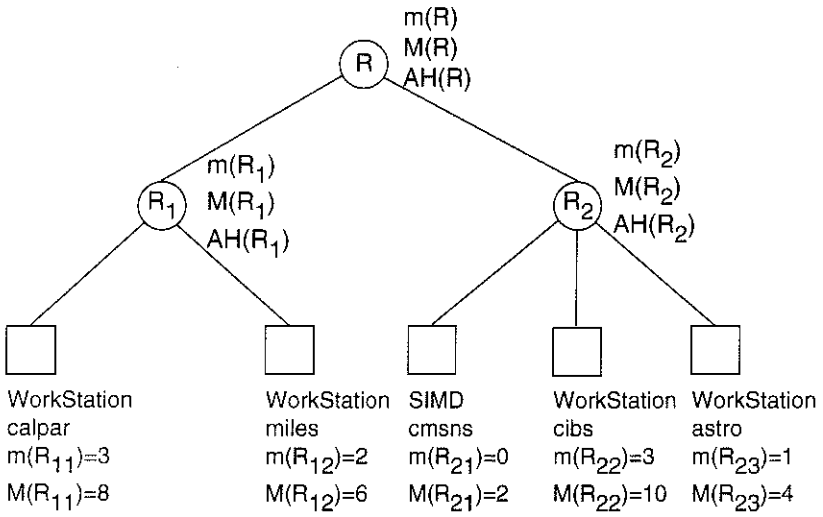
Si notino le seguenti relazioni:

$$0 \leq c(R) < m(R) \quad \Rightarrow \qquad 1 \geq V(R) > 0 \qquad \Rightarrow \quad R \in S_v$$

$$m(R) \leq c(R) \leq M(R) \quad \Rightarrow \quad 0 \geq V(R) \geq \frac{|m(R) - M(R)|}{|m(R)|} \quad \Rightarrow \quad R \in S_a$$

Ciò implica che l'insieme S_v contiene tutti i processori che non hanno ancora raggiunto il carico minimo e che quindi violano i vincoli. Quanto maggiore è la *violazione* di un processore, tanto più esso è *lontano* dal carico minimo. L'insieme S_v è ordinato in senso decrescente, cosicché, scegliendo sempre il primo elemento di S_v, il lavoro viene sempre assegnato al processore con la massima *violazione*. Quando un processore R ha raggiunto il carico minimo, $V(R)$ diventa negativo e quindi R viene rimosso da S_v per essere inserito in S_a.

R = generico nodo dell'albero dei processori
$n(R)$ = numero di figli del nodo R
$m(R)$ = vettore dei limiti inferiori al carico per il processore R
$M(R)$ = vettore dei limiti superiori al carico per il processore R
$B(R)$ = valore massimo della badness per il processore R
$AH(R)$ = (Available Hosts) nomi degli hosts che compongono il cluster R
$D(R)$ = insieme dei figli di R ($D(R) = \{R_1, \ldots, R_{n(R)}\}$)
$V(R)$ = Violazione del processore R
$c(R)$ = vettore dei carichi assegnati ad R
$RA(R)$ = insieme dei processi assegnati al processore R



	WorkStation	SIMD	Vector
$m(R_1) =$	5	0	0
$M(R_1) =$	14	0	0
$m(R_2) =$	4	0	0
$M(R_2) =$	14	2	0
$m(R) =$	9	0	0
$M(R) =$	28	2	0

$AH(R_1) = \{\text{calpar,miles}\}$
$AH(R_2) = \{\text{cmsns,cibs,astro}\}$
$AH(R) = \{\text{calpar,miles,cmsns,cibs,astro}\}$

Figura 11: Un esempio di albero dei processori

L'insieme S_a contiene i processori che hanno raggiunto il carico minimo ma non il massimo e quindi sono ancora disponibili per un assegnamento. L'insieme S_a è ordinato secondo valori crescenti del modulo dei $V(R)$. In questo modo, il primo elemento di S_a è il processore più *lontano* dal carico massimo consentito.

5.1.4 Matrice di affinità

Il valore di *affinity* viene calcolato per mezzo di una matrice che ad ogni coppia (Classe Architetturale, Tipo di Codice) fa corrispondere un valore proporzionale alla compatibilità fra codice e architettura. Un esempio di matrice potrebbe essere il seguente:

	Seq	Small Vector	Large Vector	DP	DF	VLIW
WorkStation	1	0.02	0.01	0.001	0.1	0.2
SIMD	0	0	0.5	1	0	0
Vector	0	1	1	0.2	0	0.1
MIMD-HCube	0	0	0	1	0	0
MIMD-Mesh	0	0	0	1	0	0.05

Imponendo un vincolo alla minima *affinity* ammessa su ogni processore, si impedisce all'algoritmo di assegnare processi di un certo tipo a processori inadatti ad eseguirli efficientemente; questo tipo di vincoli, pertanto, permette di esprimere le potenzialità di un'ambiente di calcolo eterogeneo.

5.2 L'algoritmo

L'algoritmo di allocazione (fig. 12) viene chiamato con i parametri (r, R, C), dove r è un nodo dell'albero dei processi, R è un nodo dell'albero dei processori e C è la matrice di affinità. Il primo passo dell'algoritmo è di verificare se R è una foglia dell'albero dei processori. Assegnare i figli di r ai figli di R genera il mapping. Se R è una foglia, allora tutti i figli di r sono già stati assegnati ad R. Pertanto, visto che l'algoritmo è implementato in maniera ricorsiva, l'unica operazione da compiere è di restituire il controllo alla procedura chiamante.

Prima di iniziare il ciclo principale dell'algoritmo, è necessario effettuare una serie di inizializzazioni. Per cominciare, $V(R_i)$ viene posto a 1 per tutti i processori; ciò implica che il carico assegnato è 0. Gli insiemi S_v e S_a vengono aggiornati di conseguenza. In seguito, P_r viene inizializzato in modo da contenere tutti i figli del nodo r, ossia tutti i processi da assegnare. Infine, gli insiemi $RA(R_i)$, contenenti gli assegnamenti parziali, vengono inizializzati opportunamente.

All'istruzione successiva, comincia la fase di assegnamento vero e proprio. Il costrutto tbf repeat until fa ripetere una serie di operazioni fino a che tutti i processi (figli di r) sono stati assegnati ($P_r = \emptyset$), oppure tutti i processori hanno raggiunto il carico minimo ($S_v = \emptyset$). Prima si sceglie il processore con la massima violazione e, in seguito, la funzione **Select** sceglie uno dei rimanenti figli di r da assegnare al processore scelto. Una volta effettuata la selezione, restano solo da fare gli aggiornamenti alle strutture dati. Il carico corrente del processore selezionato ($c(R_i)$) viene aggiornato per riflettere il lavoro assegnato (*load*); conseguentemente viene calcolato il nuovo valore della violazione ($V(R_i)$). Se $V(R_i)$ è negativo, allora il processore ha raggiunto il carico minimo e pertanto R_i viene rimosso da S_v. In aggiunta, se $c(R_i) < M(R_i)$, R_i viene aggiunto a S_a. Infine, r_j viene aggiunto all'insieme $RA(R_i)$.

Se S_v si svuota prima di P_r, allora tutti i processori hanno raggiunto il carico minimo ma ci sono ancora processi che devono essere assegnati. Il successivo blocco **repeat until** è identico a quello appena descritto con l'unica differenza che i processori vengono scelti dall'insieme S_a anziché S_v.

Il passo successivo modifica la struttura dell'albero dei processi (fig. 13). Viene creato un insieme di nodi intermedi (r'_i) fra il nodo r ed i suoi figli. Il numero di nodi creato è pari al numero di figli del nodo R. Per ogni nuovo figlio r'_i di r i nodi assegnati al processore R_i (contenuti in

Algoritmo di allocazione

Allocate(r, R, C)
if R è una foglia**then return**
$V(R_i) := 1 \; \forall \, R_i \in D(R)$
$S_v := \{R_i \mid V(R_i) = 1 \; \wedge \; R_i \in D(R)\}$
$S_a := \emptyset$
$c(R_i) := 0 \; \forall \, R_i \in D(R)$
$P_r := \{r_j \mid r_j \in D(r)\}$
$RA(R_i) := \emptyset \; \forall \, R_i \in D(R)$
repeat until $P_r = \emptyset$**or** $S_v = \emptyset$
Sia R_i il figlio di R con $V(R_i)$ massimo
$r_j, load =$ Select(r, R, R_i, P_r, C)
Effettua gli aggiornamenti
$c(R_i) := load$
$V(R_i) :=$ Calc_Violation$(m(R_i), c(R_i))$
if $V(R_i) \leq 0$**then**
Rimuovi R_i da S_v
if $c(R_i) < M(R_i)$**then** aggiungi R_i a S_a
Rimuovi r_j da P_r
Inserisci r_j in $RA(R_i)$
if $P_r \neq \emptyset$**then repeat until** $P_r = \emptyset$
Sia R_i il figlio di R con minimo $|V(R_i)|$ in S_a
$r_j, load =$ Select(r, R, R_i, P_r, C)
Effettua gli aggiornamenti
$c(R_i) := load$
$V(R_i) :=$ Calc_Violation$(m(R_i), c(R_i))$
if $c(R_i) = M(R_i)$**then** rimuovi R_i da S_a
Rimuovi r_j da P_r
Inserisci r_j in $RA(R_i)$
Per ogni $R_i \in D(R)$, crea un nuovo nodo r_i come figlio di r
e rendi figli di r_i tutti i nodi in $RA(R_i)$
Allocate$(r_i, R_i) \; \forall \, i = 1, \ldots, n(r) = n(R)$

Figura 12: L'algoritmo di allocazione

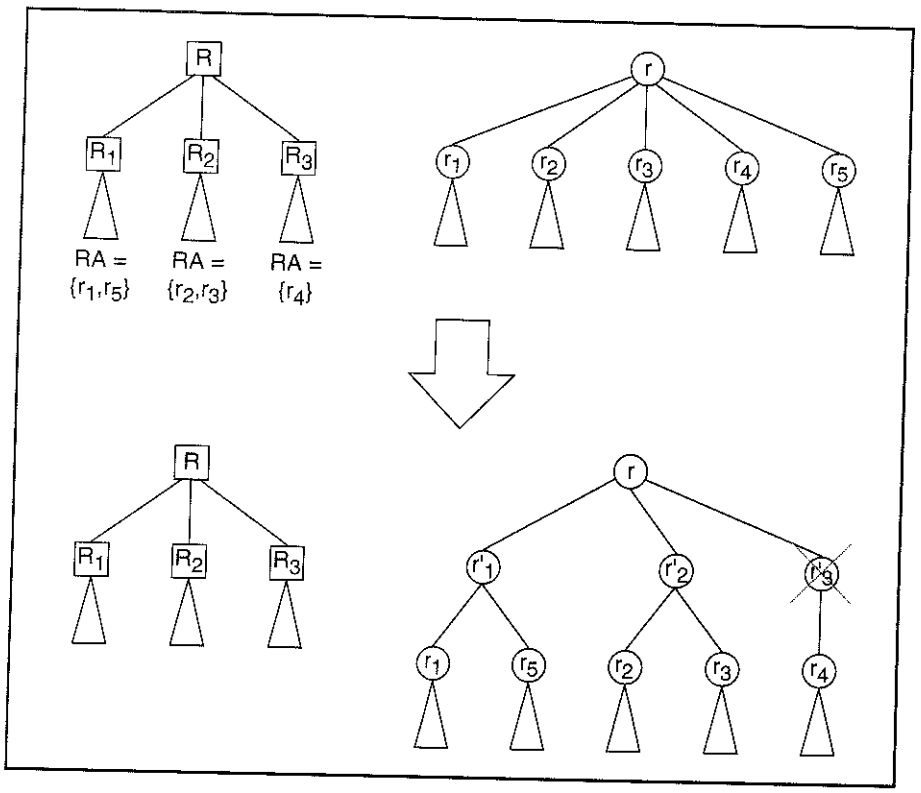


Figura 13: Esempio di come la struttura dell'albero dei processi venga modificata durante la fase di allocazione

$RA(R_i)$) vengono resi figli di r'_i stesso. Se solo un processo è assegnato ad un particolare processore, il nodo intermedio viene scartato. Il passo finale consiste nel richiamare ricorsivamente l'algoritmo. A questo punto, i figli di r sono stati assegnati ai figli di R in modo che r e R hanno lo stesso numero di figli. Per ognuno di questi assegnamenti, viene effettuata una chiamata ricorsiva alla procedura **Allocate** con parametri (r_i, R_i, C), dove gli r_i sono i nodi creati al passo precedente.

5.3 La funzione Select

Nel corso dell'allocazione, il prossimo processore a cui assegnare il carico viene determinato in maniera molto semplice: viene scelto il processore che è più lontano (percentualmente) dal carico minimo imposto. Più difficile invece è la scelta di un cluster di processi che si adatti al meglio al processore scelto; non solo esso deve rispettare i vincoli sul processore corrente ma non deve provocare, in seguito, un fallimento dell'algoritmo dovuto alla impossibilità di rispettare i vincoli. Ciò è ottenuto verificando se il processo candidato r_j rispetta il vincolo:

$$\sum_{r' \in P_r - \{r_j\}} |w(r')| \geq \sum_{R' \in D(R) \wedge c(R') < m(R')} (|m(R') - c(R')|) \qquad (1)$$

Il lato sinistro di tale diseguaglianza rappresenta il lavoro non ancora assegnato mentre il lato destro indica la quantità di lavoro ancora necessaria affinché tutti i processori raggiungano il carico minimo imposto. Nella versione originaria dell'algoritmo, tale controllo garantisce che, se una soluzione esiste, l'algoritmo è in grado di trovarla. Poiché la nostra versione dell'algoritmo ha un vincolo addizionale (quello sulla *affinity*) questo controllo non è sufficiente a garantire l'ottenimento di una soluzione ogniqualvolta ne esista una. Tuttavia, il controllo (1) viene comunque effettuato,

Funzione Select

Select(r, R, R_i, P_r, C)
  do while true
    $AP := \{r_j \in P_r : |w(r_j)| \leq |M(R_i)| - |c(R_i)| \ \wedge \ RH(r_j) \subseteq AH(R_i)\}$
    if $AP = \emptyset$**then**
      Sia $\overline{r} \in P_r$ con massimo $|w(\overline{r})|$
      Pop$(\overline{r})$
      Aggiorna l'albero dei processi
    else
      do while $|AP| > 0$
        Sia $r_j \in AP$ con massimo $w(r_j)$
        Rimuovi r_j da AP
        $wtot := w(r_j) + \sum_{r' \in RA(R_i)} w(r')$
        $w_{rem} := \sum_{r' \in P_r - \{r_j\}} |w(r')|$
        $w_{need} := \sum_{R' \in D(R) \ \wedge \ c(R') < m(R')} (|m(R') - c(R')|)$
        $aff :=$ Calc_Affinity$(wtot, M(R_i), C)$
        if $aff \geq Aff(R_i)$**and** $w_{rem} \geq w_{need}$**then**
          $load :=$ Calc_Load$(wtot, M(R_i), C)$
          return$(r_j, load)$
      Sia $\overline{r} \in P_r$ con massimo $|w(\overline{r})|$
      Pop$(\overline{r})$
      Aggiorna l'albero dei processi

Figura 14: La funzione Select

in modo da ridurre il numero dei casi in cui l'algoritmo fallisce. Vediamo ora una descrizione più dettagliata (fig. 14) dei passi dell'algoritmo. Si entra in un ciclo infinito da cui si esce non appena si è trovata una soluzione accettabile per il processore corrente R_i. L'insieme AP contiene i processi che possono essere assegnati ad R_i: infatti essi devono appartenere a P_r (ossia l'insieme dei processi non ancora assegnati), il loro peso deve essere inferiore al residuo di carico massimo ammissibile per R_i ($|w(r_j)| \leq |M(R_i)| - |c(R_i)|$) ed inoltre gli hosts richiesti devono essere disponibili sul cluster corrente ($RH(r_j) \subseteq AH(R_i)$). Se l'insieme AP è vuoto, si sceglie il cluster di processi più pesante e si effettua un *Pop*, ossia i figli di $\overline{r}$ diventano figli del padre di $\overline{r}$ e $\overline{r}$ stesso viene cancellato. L'effetto risultante è di scomporre il cluster di processi più pesante nelle sue componenti (fig. 15). Se invece AP è non vuoto, allora lo si scandisce fino a trovare un cluster di processi che, oltre a verificare i vincoli anzidetti, rispetti le limitazioni sulla affinity per il cluster corrente R_i e il vincolo (1). I cluster di processi in AP vengono esaminati a partire dal più pesante: preso un cluster di processi r_j, lo si prova ad assegnare al cluster di processori corrente R_i. Se l'assegnamento non viola i vincoli, allora r_j viene scelto per essere definitivamente assegnato a R_i, altrimenti, si procede nella scansione di AP. Il vincolo (1) viene verificato assicurandosi che w_{rem} (il numero complessivo dei processi rimanenti dopo aver tolto r_j) sia non inferiore a w_{need} (il numero totale di processi necessari per soddisfare i vincoli di minimo carico). Il vincolo sulla affinity viene verificato servendosi della funzione ausiliaria **Calc_Affinity**. Come si può notare dal codice in figura 14, tale funzione viene richiamata con parametro attuale $wtot$, che comprende sia i processi già assegnati precedentemente a R_i, sia i processi rappresentati da r_j.

Se, dopo aver scandito tutto l'insieme AP, nessun cluster di processi risulta adatto, si procede

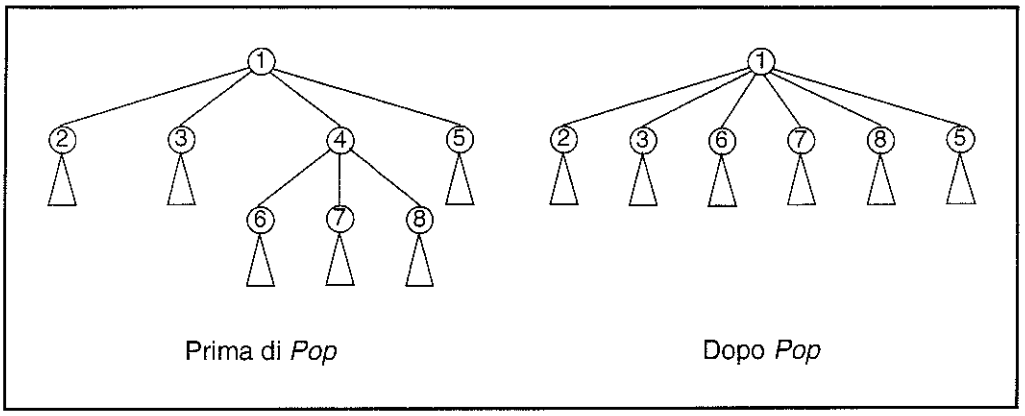


Figura 15: Esempio di *Pop* di un nodo

ad una operazione di *Pop* sul cluster di processi più pesante e il ciclo ricomincia.

5.4 La funzione Calc_Affinity

La funzione **Calc_Affinity** (fig. 16) riceve in ingresso, oltre alla matrice di affinità, tre vettori, m, M e w. I vettori m ed M contengono rispettivamente i vincoli sul minimo e massimo carico, mentre il vettore w contiene il peso del cluster di processi da assegnare. Il valore restituito rappresenta l'affinity risultante dall'assegnare il carico rappresentato da w ad un cluster di processori con vincoli sul carico pari a m e M. Il problema da risolvere è che i vincoli sul carico sono espressi in base alle classi architetturali che compongono il cluster di processori mentre il peso dei cluster di processi viene espresso in base ai tipi di codice. È pertanto necessaria una "trasformazione" che permetta di passare da tipi di codice a classi architetturali. Questa "trasformazione" viene effettuata servendosi della matrice di affinità ed utilizzando un criterio greedy.

La fase di inizializzazione della funzione **Calc_Affinity** consiste nel copiare i vettori m, M e w in tre vettori di lavoro m', M' e w' e di porre il valore della variabile $affinity$ pari a 1. Il calcolo dell'affinity viene effettuato in quattro passi distinti.

Nei primi due passi si cerca di assegnare i processi in modo da soddisfare i vincoli di minimo carico, mentre nei due successivi, i processi rimasti vengono assegnati evitando di superare i vincoli di massimo carico. Esaminiamo in dettaglio i vari passi.

Nel primo passo, si considerano singolarmente i vari tipi di codice. Chiamato *code* il tipo di codice che ci accingiamo a considerare, sia *bestarch* la migliore architettura per *code*, ossia la classe architetturale che presenta il massimo valore di affinity in corrispondenza del tipo di codice in esame (*code*). In seguito, viene ricavato dalla matrice di affinità il valore di affinity per la coppia (*bestarch*, *code*). A questo punto viene calcolato il massimo numero di processi di tipo *code* che devono essere assegnati a processori di tipo *bestarch*; tale valore (chiamato *maxload*) è pari al minimo tra il numero di processi disponibili di tipo *code* ($w'[code]$) ed il minimo numero di processi richiesto per la classe architetturale *bestarch* ($m'[bestarch]$). Le istruzioni successive si occupano di decrementare opportunamente i valori di m', M' e w' e di aggiornare il valore corrente della affinity, moltiplicando il vecchio valore per quello trovato all'iterazione corrente. In sostanza, nel primo passo si cerca di assegnare i processi ai processori che meglio si prestano ad eseguirli, avendo come obiettivo la saturazione dei vincoli di carico minimo.

Se dopo il primo passo il vettore w' ha componenti diverse da zero, allora vuol dire che sono rimasti ancora processi da assegnare. Come nel passo precedente, nel secondo passo della funzione **Calc_Affinity** i processi rimanenti vengono assegnati con l'obiettivo di soddisfare i vincoli di minimo carico. In questo passo però, non è possibile scegliere, per un dato tipo di codice, la

classe architetturale con massima affinity, in quanto tale scelta è già stata effettuata nel passo precedente. In questo passo, per ciascun tipo di codice *code* vengono prese in considerazione tutte le classi architetturali a disposizione, partendo dalla seconda migliore, fino alla peggiore. Le rimanenti operazioni ricalcano quelle viste nel primo passo.

In definitiva, i primi due passi si occupano di assegnare i processi in modo da soddisfare i vincoli sul minimo carico, scegliendo prima le architetture con massima affinità e poi quelle con affinità via via minore.

I due passi successivi sono analoghi a quelli già visti, con la differenza che l'obiettivo non è la saturazione dei vincoli di minimo ma, bensì, il non superamento dei limiti di massimo carico.

5.5 La funzione Calc_Load

Dopo aver assegnato un cluster di processi r_j ad un cluster di processori R_i, occorre calcolare il nuovo valore del vettore contenente i carichi per il cluster R_i, ossia $c(R_i)$. In realtà questo problema è stato già affrontato all'interno della funzione **Calc_Affinity** e pertanto è possibile risolverlo in maniera analoga, utilizzando una funzione denominata **Calc_Load**. I parametri richiesti in ingresso sono il vettore dei processi (w), i vettori contenenti i vincoli di minimo e massimo carico (m e M) e la matrice di affinità (C). Il valore restituito in uscita è un vettore che indica il numero di processi assegnati ad ogni classe architetturale.

Una schematizzazione della funzione **Calc_Load** viene mostrata in figura 17. Come si può notare, la struttura della funzione è simile a quella descritta nella sezione precedente. I valori di carico che vengono via via calcolati (*maxload*) vengono accumulati nel vettore *load* e alla fine tale vettore viene restituito alla procedura chiamante.

6 Un esempio di funzionamento

Per meglio comprendere il funzionamento dell'algoritmo presentato, riportiamo un semplice esempio utile a chiarire i concetti illustrati precedentemente. In figura 18 vengono illustrati l'albero dei processi e l'albero dei processori oggetto del nostro esempio. Le sigle che compaiono sotto le foglie dell'albero dei processi ne indicano il tipo di codice. La tabella successiva indica i pesi dei nodi interni. Per quanto riguarda l'albero dei processori mostrato, va detto che le notazioni riguardanti i vincoli di massimo e minimo poste sotto le foglie (ad esempio $m = 1$, $M = 3, \ldots$) non sono del tutto corrette, in quanto m e M sono vettori e non scalari. Però, poiché ciascuna foglia è composta da un'unica classe architetturale, abbiamo abbreviato la notazione $m = [1\ 0\ 0]$ con $m = 1$, specificando altresì la classe architetturale a cui il vincolo si riferisce.

La matrice di affinità che utilizzeremo per l'esempio è la seguente:

	Seq	DP	Large Vector	Small Vector
WorkStation	1	0.05	0.1	0.2
SIMD	0.05	1	0.8	0.3
Vector	0.1	0.2	1	1

Consideriamo l'algoritmo di allocazione: dopo aver effettuato le inizializzazioni, avremo che:

$$S_v = \{R_1, R_2\}, \quad S_a = \emptyset$$

$$P_r = \{r_1, r_2, r_3\}$$

$$c(R_i) = 0 per\ i = 1, 2$$

$$V(R_i) = 1 per\ i = 1, 2$$

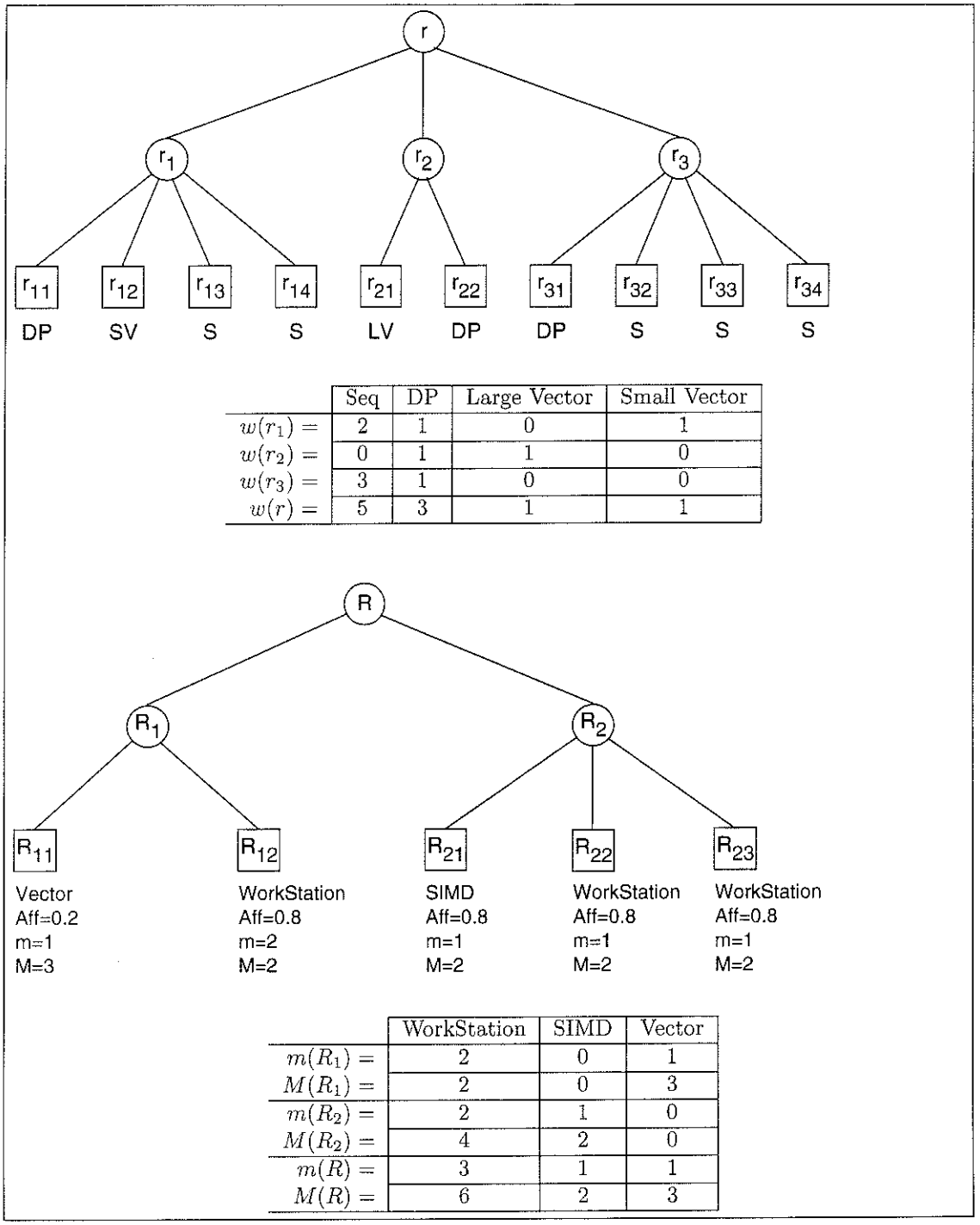
$$RA(R_i) = \emptyset per\ i = 1, 2$$

Calc_Affinity (w, m, M, C)
$m' := m$
$M' := M$
$w' := w$
$affinity := 1$
foreach *code***do**
Sia *bestarch* la miglior architettura per *code*
$aff\_factor := C[bestarch][code]$
$maxload := min(m'[bestarch], w'[code])$
$m'[bestarch] := m'[bestarch] - maxload$
$M'[bestarch] := M'[bestarch] - maxload$
$w'[code] := w'[code] - maxload$
$affinity := affinity * (aff\_factor)^{maxload}$
foreach *code***do**
foreach *arch***do**
$aff\_factor := C[arch][code]$
$maxload := min(m'[arch], w'[code])$
$m'[arch] := m'[arch] - maxload$
$M'[arch] := M'[arch] - maxload$
$w'[code] := w'[code] - maxload$
$affinity := affinity * (aff\_factor)^{maxload}$
foreach *code***do**
Sia *bestarch* la miglior architettura per *code*
$aff\_factor := C[bestarch][code]$
$maxload := min(M'[bestarch], w'[code])$
$M'[bestarch] := M'[bestarch] - maxload$
$w'[code] := w'[code] - maxload$
$affinity := affinity * (aff\_factor)^{maxload}$
foreach *code***do**
foreach *arch***do**
$aff\_factor := C[arch][code]$
$maxload := min(M'[arch], w'[code])$
$M'[arch] := M'[arch] - maxload$
$w'[code] := w'[code] - maxload$
$affinity := affinity * (aff\_factor)^{maxload}$
return $(affinity)$

Figura 16: La funzione **Calc_Affinity**

```
Calc_Load(w, m, M, C)
    w' := w
    m' := m
    M' := M
    load := 0
    foreach code do
        Sia bestarch la miglior architettura per code
        maxload := min(m'[bestarch], w'[code])
        w'[code] := w'[code] − maxload
        m'[bestarch] := m'[bestarch] − maxload
        M'[bestarch] := M'[bestarch] − maxload
        load[bestarch] := load[bestarch] + maxload
    foreach code do
        foreach arch do
            maxload := min(m'[arch], w'[code])
            w'[code] := w'[code] − maxload
            m'[arch] := m'[arch] − maxload
            M'[arch] := M'[arch] − maxload
            load[arch] := load[arch] + maxload
    foreach code do
        Sia bestarch la miglior architettura per code
        maxload := min(M'[bestarch], w'[code])
        w'[code] := w'[code] − maxload
        M'[bestarch] := M'[bestarch] − maxload
        load[bestarch] := load[bestarch] + maxload
    foreach code do
        foreach arch do
            maxload := min(M'[arch], w'[code])
            w'[code] := w'[code] − maxload
            M'[arch] := M'[arch] − maxload
            load[arch] := load[arch] + maxload
    return(load)
```

Figura 17: La funzione **Calc_Load**



	Seq	DP	Large Vector	Small Vector
$w(r_1) =$	2	1	0	1
$w(r_2) =$	0	1	1	0
$w(r_3) =$	3	1	0	0
$w(r) =$	5	3	1	1

	WorkStation	SIMD	Vector
$m(R_1) =$	2	0	1
$M(R_1) =$	2	0	3
$m(R_2) =$	2	1	0
$M(R_2) =$	4	2	0
$m(R) =$	3	1	1
$M(R) =$	6	2	3

Figura 18: Alberi utilizzati per l'esempio

A questo punto si entra nel primo blocco **repeat until** e si sceglie il cluster di processori con la massima violazione, supponiamo sia R_1 (in realtà la violazione è pari a 1 sia per R_1 che per R_2). Scelto il cluster di processori, bisogna scegliere il cluster di processi, utilizzando la funzione **Select**. In questo caso la **Select** viene chiamata con parametri r, R, R_1, P_r e C (la nostra matrice di affinità).

Entrati nella funzione **Select** bisogna determinare il contenuto dell'insieme AP. Come si noterà non sono stati imposti vincoli sugli host e pertanto è necessario semplicemente verificare che $|w(r_j)| \leq |M(R_1)|$ (si noti che in questo momento $|c(R_1)| = 0$). Poiché

$$|M(R_1)| = 5$$

$$|w(r_1)| = |w(r_3)| = 4$$

$$|w(r_2)| = 2$$

si ha che $AP = \{r_1, r_2, r_3\}$. Poiché AP è non vuoto, si entra nel ciclo **do while** e si sceglie il cluster di processi di peso massimo contenuto in AP. Supponiamo che venga scelto r_1. Dopo aver rimosso r_1 da AP si calcolano le tre quantità $wtot$ (il peso complessivo dei processi già assegnati a R_1 più il cluster candidato r_1), w_{rem} (il numero complessivo dei processi ancora rimasti da assegnare) e w_{need} (il numero di processi necessari a soddisfare i vincoli di minimo carico).

$$wtot = w(r_1) = [2\ 1\ 0\ 1] essendo\ RA(R_1) = \emptyset$$

$$w_{rem} = |w(r_2)| + |w(r_3)| = 2 + 4 = 6$$

$$w_{need} = |m(R_2)| = 3 essendo\ c(R_2) = 0$$

Essendo $w_{rem} \geq w_{need}$, è necessario solo verificare il vincolo sull'affinità. Il vincolo di affinity per le foglie dell'albero dei processori viene specificato come input dell'algoritmo; per un nodo interno R si è scelto di porre

$$Aff(R) = min(Aff(R_1), \ldots, Aff(R_{n(R)})) \quad R_1, \ldots, R_{n(R)} \in D(R)$$

ossia la minima affinity dei figli di R. Pertanto $Aff(R_1) = 0.2$.

Nella figura 19 viene mostrata la sequenza di passi necessaria per calcolare l'affinity risultante dall'assegnamento di r_1 a R_1. I valori iniziali delle variabili sono mostrati nella prima colonna. Si considera $w'[Seq]$ e si cerca la miglior architettura per il tipo di codice "Sequenziale"; dalla matrice di affinità si vede che l'architettura desiderata è "WorkStation". A questo punto va calcolato il valore di $maxload$:

$$maxload = \min(w'[Seq], m(R_1)[WS]) = \min(2, 2) = 2$$

m'	2, 0, 1	0, 0, 1	0, 0, 0	
M'	2, 0, 3	0, 0, 3	0, 0, 2	0, 0, 3
w'	2, 1, 0, 1	0, 1, 0, 1	0, 1, 0, 0	0, 0, 0, 0
Aff	1	1	1	0.2
		Fase 1		Fase 4

Figura 19: Sequenza di passi nella funzione **Calc_Affinity**

In seguito vengono effettuati gli aggiornamenti (mostrati nella seconda colonna) e si passa al tipo di codice successivo, ossia "DataParallel". La miglior architettura per tale tipo di codice è "SIMD" ma come si può notare il valore di minimo per "SIMD" è pari a 0 e pertanto si passa al tipo

di codice successivo. Per il tipo di codice "Large Vector" non esistono processi (il corrispondente valore di $w'(R_1)$ è nullo) e quindi non ci sono modifiche da fare. L'ultimo tipo di codice da esaminare è "Small Vector". In questo caso la miglior architettura è "Vector" e $maxload = 1$. Effettuati gli aggiornamenti, la situazione risultante è quella mostrata nella terza colonna. La prima fase della funzione **Calc_Affinity** è conclusa.

Poiché i vincoli di minimo sono stati soddisfatti, nessuna operazione deve essere compiuta nella seconda fase. Resta ancora una unità di carico da assegnare (quella di tipo "DataParallel"). Tale carico non può essere assegnato nella terza fase in quanto non è possibile effettuare una scelta ottima (dato che $M(R_1)[SIMD] = 0$) e pertanto bisogna ricorrere ad una scelta non ottima da effettuare nella quarta fase. Scartata la classe "SIMD", la seconda miglior scelta per il tipo "DataParallel" è "Vector", con un fattore di affinità pari a 0.2. Dopo aver effettuato quest'ultimo assegnamento, la situazione risultante è mostrata nell'ultima colonna della figura 19.

Tornando alla funzione **Select**, notiamo che il valore di affinity appena calcolato è uguale al minimo valore ammissibile ($Aff(R_1)$) e pertanto il cluster di processi r_1 può essere assegnato al cluster di processori R_1.

Resta da calcolare il carico $c(R_1)$. Notiamo che la funzione **Calc_Load** ha una struttura simile alla funzione **Calc_Affinity**. Pertanto, la sequenza di passi effettuata (fig. 20) è simile a quella mostrata in figura 19.

m'	2, 0, 1	0, 0, 1	0, 0, 0	
M'	2, 0, 3	0, 0, 3	0, 0, 2	0, 0, 3
w'	2, 1, 0, 1	0, 1, 0, 1	0, 1, 0, 0	0, 0, 0, 0
load	0, 0, 0	2, 0, 0	2, 0, 1	2, 0, 2
	Fase 1			Fase 4

Figura 20: Sequenza di passi nella funzione **Calc_Load**

Tornati nella funzione di allocazione, resta da calcolare il nuovo valore della violazione per R_1. Si ha che:

$$d_+ = \max(m(R_1) - c(R_1), 0) = \max([2\ 0\ 1] - [2\ 0\ 2], [0\ 0\ 0]) = [0\ 0\ 0]$$

$$d_- = \min(m(R_1) - c(R_1), 0) = \min([2\ 0\ 1] - [2\ 0\ 2], [0\ 0\ 0]) = [0\ 0\ -1]$$

Poiché $|d_+| = 0$ il valore della violazione è determinato da:

$$V(R) = \frac{|d_-|}{|m(R_1)|} = \frac{-1}{3}$$

Essendo la violazione minore di zero, R_1 ha raggiunto il carico minimo e pertanto viene rimosso da S_v per essere inserito in S_a. Una volta aggiornati P_r e $RA(R_1)$, il ciclo ricomincia.

7 Un esempio di terminazione senza soluzione

Per comprendere i limiti del nostro algoritmo illustriamo un esempio per il quale l'algoritmo sviluppato termina senza una soluzione, anche se il problema ammette almeno una soluzione. In figura 22 sono mostrati gli alberi dei processi e dei processori. Come matrice di affinità può essere utilizzata quella del precedente esempio.

Tralasciamo tutti i dettagli relativi alle inizializzazioni e consideriamo cosa accade se il primo cluster di processori candidato a ricevere lavoro è R_1. In tal caso, sia r_1 che r_2 appartengono ad AP. Poiché il cluster di peso massimo in AP è r_1, esso verrà assegnato a R_1. Tale assegnazione è possibile in quanto il carico residuo (r_2) è sufficiente a saturare i vincoli di minimo di R_2 e l'affinity

descritta. I test sono stati condotti con grafi di 30, 60, 120, 240 e 480 nodi. Per ogni particolare dimensione, sono stati generati 100 grafi. Tali grafi sono stati costruiti scegliendo casualmente i singoli sottografi della libreria fino ad ottenere il numero di nodi desiderato. Ciascun sottografo è connesso al successivo con un arco di peso 1 e l'ultimo sottografo è connesso al primo, formando una struttura ad anello.

Poiché i sottografi sono connessi in maniera circolare, è possibile calcolare un limite inferiore della soluzione ottima. Supponiamo che i processi abbiano tutti lo stesso tipo di codice, che non esistano vincoli sulla affinity, che il numero di processori sia minore o uguale al numero di sottografi del grafo dei processi e che gli archi del grafo dei processori abbiano tutti costo 1. Supponendo inoltre che i processori appartengano alla stessa classe architetturale e che i vincoli sul carico (m e M) siano pari a 2 e ∞, la soluzione ottima sarà quella di allocare almeno un sottografo ad ogni processore ed il costo di comunicazione risultante sarà pari a N, il numero di processori.

9 Prestazioni

Il lavoro di testing dell'algoritmo di mapping è stato suddiviso in due fasi distinte. Nella prima fase, abbiamo eseguito una serie di test "omogenei", utilizzando processi con uguale tipo di codice e processori della stessa classe architetturale. In questo modo abbiamo potuto verificare che il nostro algoritmo fornisse risultati simili a quelli dell'algoritmo originario di Bowen et al. La seconda serie di test è stata condotta utilizzando grafi di processi e di processori entrambi eterogenei, per verificare le prestazioni del nostro algoritmo in tali situazioni.

9.1 Prove su ambiente omogeneo

I grafi di processi utilizzati in questa serie di test sono composti da 30, 60, 120, 240 e 480 nodi, mentre i grafi dei processori sono formati da 2, 4, 8 e 16 nodi, connessi da archi di costo unitario.

Per una data coppia (grafo di processi, grafo di processori) definiamo un *carico medio*:

$$carico\ medio = \frac{numero\ di\ processi}{numero\ di\ processori}$$

Definiamo inoltre un insieme di varianze (10%, 25%, 50% e 98%) che ci permettono di fissare il minimo e massimo carico per un processore (m, M) a partire dal carico medio:

$$m = carico\ medio - carico\ medio \times varianza$$

$$M = carico\ medio + carico\ medio \times varianza$$

Tutti i test sono stati effettuati su una HP9000/720 con sistema operativo HP-UX 9.01. I valori riportati, sia i tempi sia i costi di comunicazione, rappresentano la media aritmetica su 100 prove.

Il primo test è stato effettuato per misurare i tempi di completamento dell'algoritmo di mapping al variare della dimensione del problema. Fissati un carico medio pari a 15 processi per processore ed una varianza del 10%, sono state effettuate quattro serie di test, con 30, 60, 120 e 240 processi. Dal grafico (fig. 23) si vede come l'algoritmo sia utilizzabile anche per grafi abbastanza grandi. I tempi riportati comprendono le fasi di clustering dei grafi dei processi e dei processori e la successiva fase di allocazione, mentre non è stato incluso il tempo necessario a leggere la descrizione dei grafi dal disco. I grafici successivi (fig. 24–27) mostrano come varia il costo della soluzione trovata utilizzando diversi valori di varianza per il carico. Lo stesso test è stato effettuato con quattro diversi valori del carico medio (3.75, 7.5, 15 e 30 processi per processore). La prima osservazione da fare è che al crescere del carico medio, diminuisce il costo della soluzione euristica. Ciò è dovuto al fatto che, con carichi medi elevati, (7.5, 15 e 30) è possibile assegnare interi sottografi a ciascun processore, evitando il *popping* su archi di costo elevato. Viceversa, con carico medio pari a 3.75,

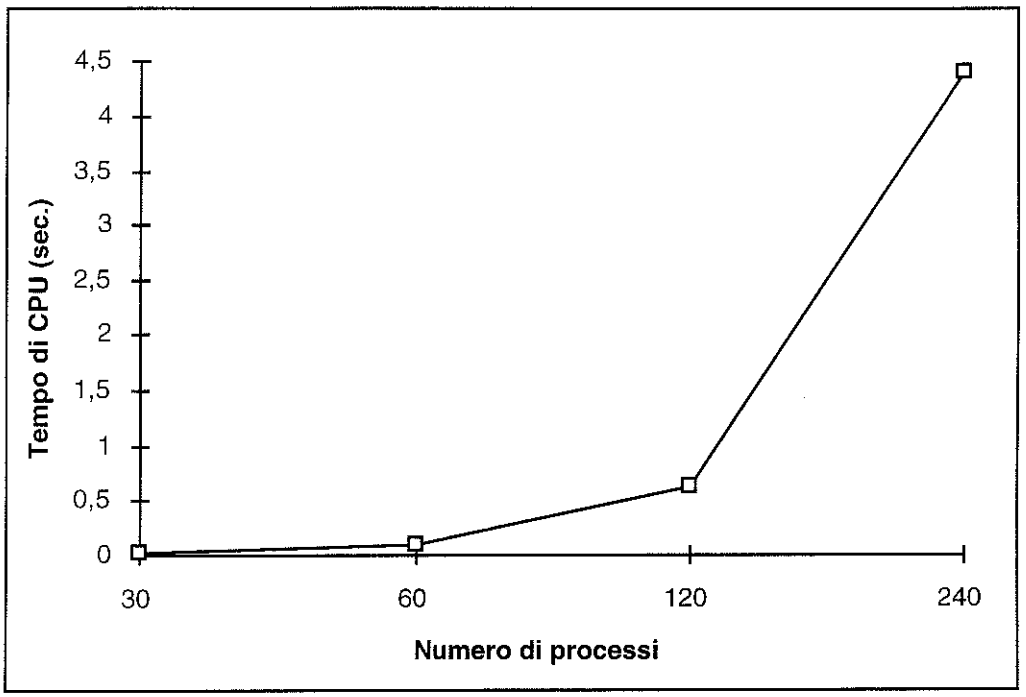


Figura 23: Analisi del tempo di calcolo

gran parte dei sottografi (composti da 2–8 nodi) deve essere distribuita su più processori, facendo si che archi dal costo elevato (quelli interni ai sottografi) contribuiscano al costo complessivo della soluzione. Analizziamo ora in dettaglio i singoli grafici. Nel grafico di figura 24, come ci si poteva aspettare, il costo della soluzione ottima cresce al diminuire della varianza. Infatti, diminuendo la varianza, l'algoritmo si trova a dover operare con vincoli sempre più stretti, con un conseguente aumento della necessità di effettuare il *popping*. Nei restanti 3 casi (fig. 25–27) si nota un andamento differente rispetto al primo caso. I risultati peggiori si hanno con varianza del 10%. Ci si potrebbe aspettare che una varianza del 98% dia i migliori risultati. Invece, il risultato migliore si ottiene con una varianza del 50%. Avendo una varianza molto larga, l'algoritmo tenterà di allocare i processi a grandi blocchi a ciascun processore, fino a quando non sarà costretto a ripartire un piccolo numero di processi su molti processori. In altri termini, l'algoritmo è costretto ad effettuare il *popping* in prossimità delle foglie dell'albero, dove sono presenti archi pesanti. Invece, con varianze più strette (50% e 25%), il *popping* deve essere effettuato prima, a causa dei limiti superiori più bassi. Ciò implica che il *popping* viene effettuato a livelli alti dell'albero, che contengono cluster di processi che comunicano poco frequentemente (archi dal costo inferiore). I grafici che seguono (fig. 28 e 29) mettono in relazione i costi di comunicazione con il numero di processori; tali costi vengono inoltre confrontati con i rispettivi lower bound. In entrambi i casi la varianza per i vincoli sul carico è pari al 50%. Nel primo caso (fig. 28) un piccolo grafo di processi di 60 nodi viene mappato su 2, 4, 8 e 16 processori. Come prevedibile, i costi crescono al crescere del numero di processori, pur mantenendosi molto vicini al lower bound. Fa eccezione il caso con 16 processori; in tale situazione, infatti, il carico medio è talmente basso (3.75) che molti sottografi devono essere suddivisi su più processori, facendo lievitare il costo della soluzione ottenuta. Nel secondo caso (fig. 29) il carico medio viene mantenuto costante. In tale situazione l'algoritmo riesce a produrre risultati molto prossimi al lower bound.

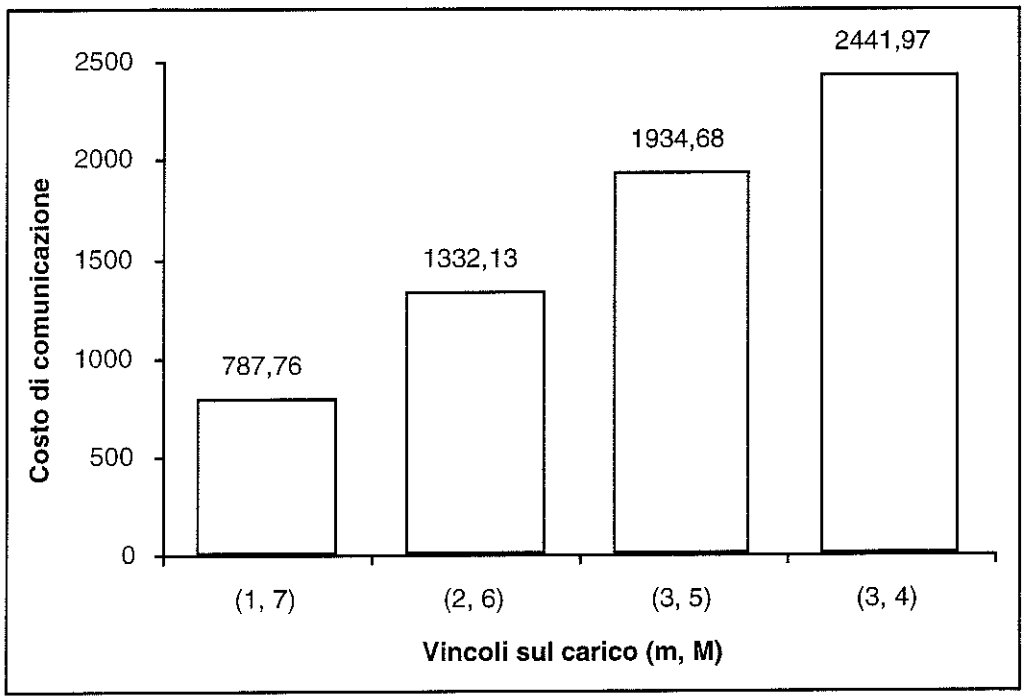


Figura 24: Sensibilità ai vincoli sul carico con carico medio pari a 3.75

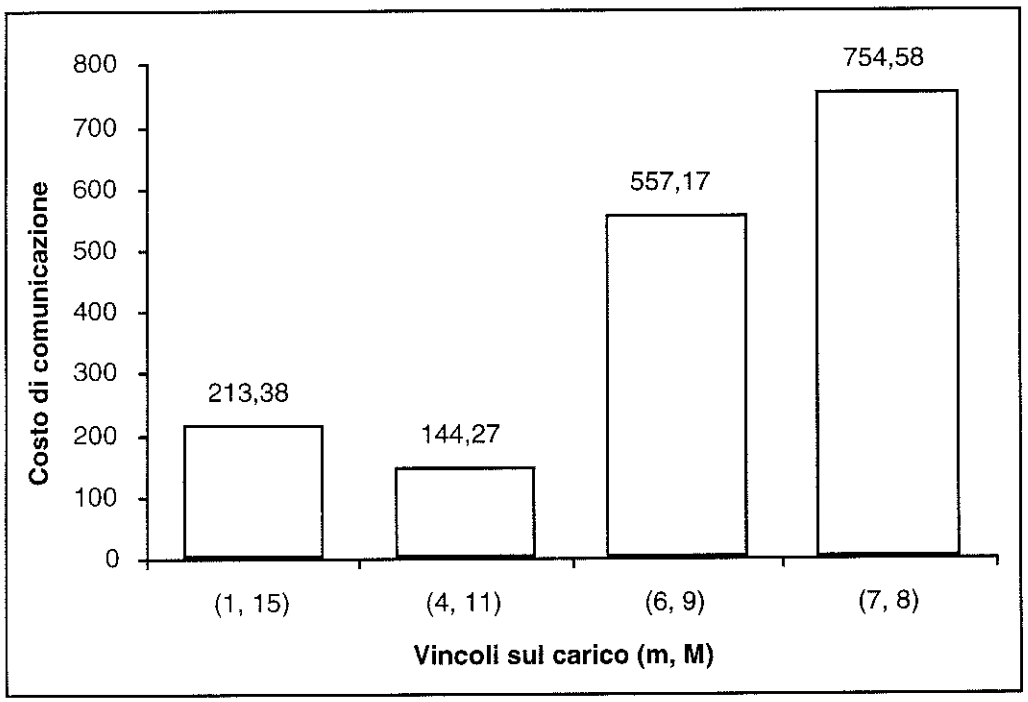


Figura 25: Sensibilità ai vincoli sul carico con carico medio pari a 7.5

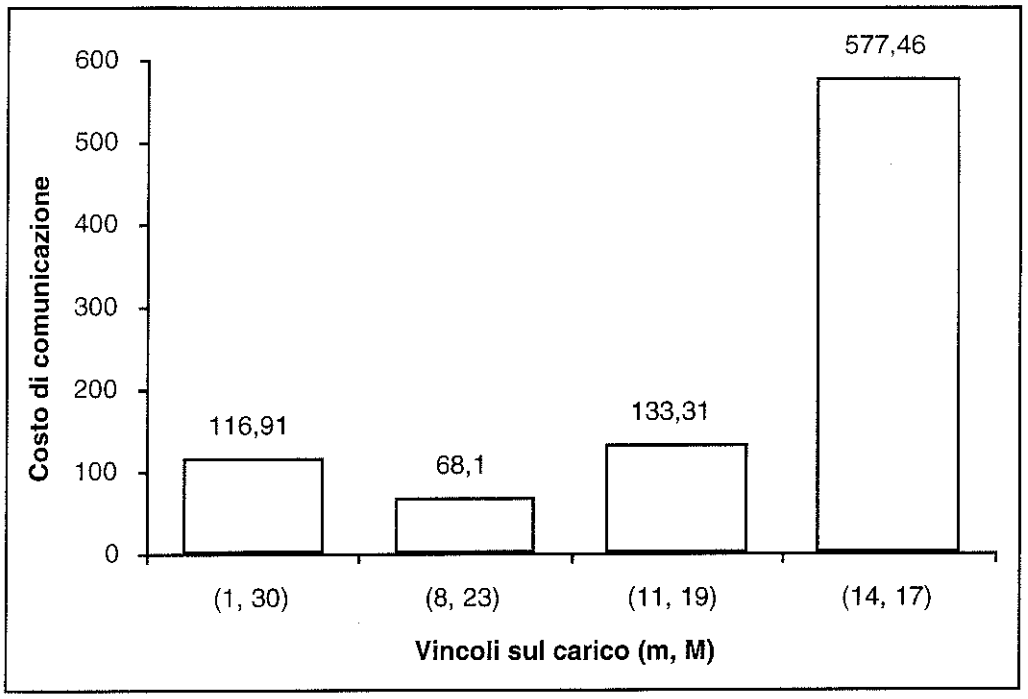


Figura 26: Sensibilità ai vincoli sul carico con carico medio pari a 15

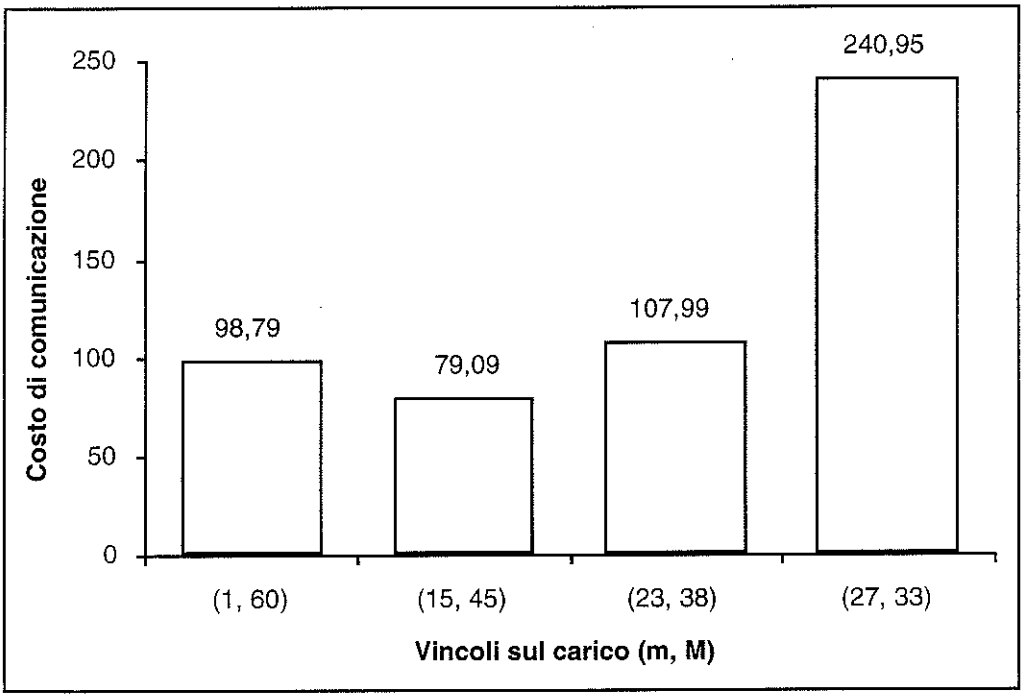


Figura 27: Sensibilità ai vincoli sul carico con carico medio pari a 30

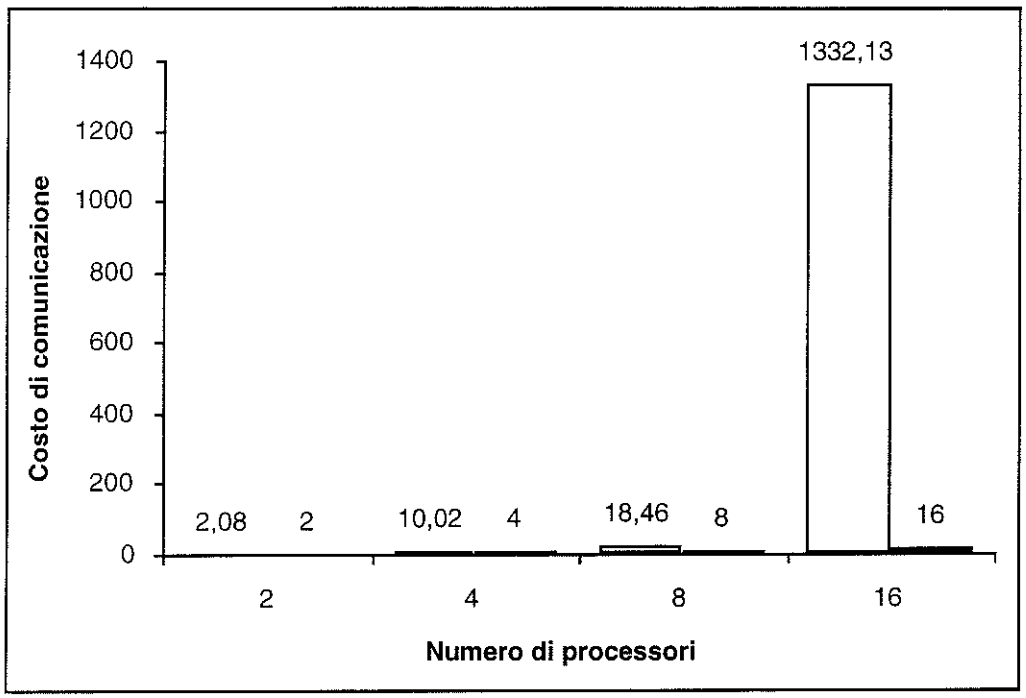


Figura 28: Costi di comunicazione per 60 processi su 2, 4, 8 e 16 processori

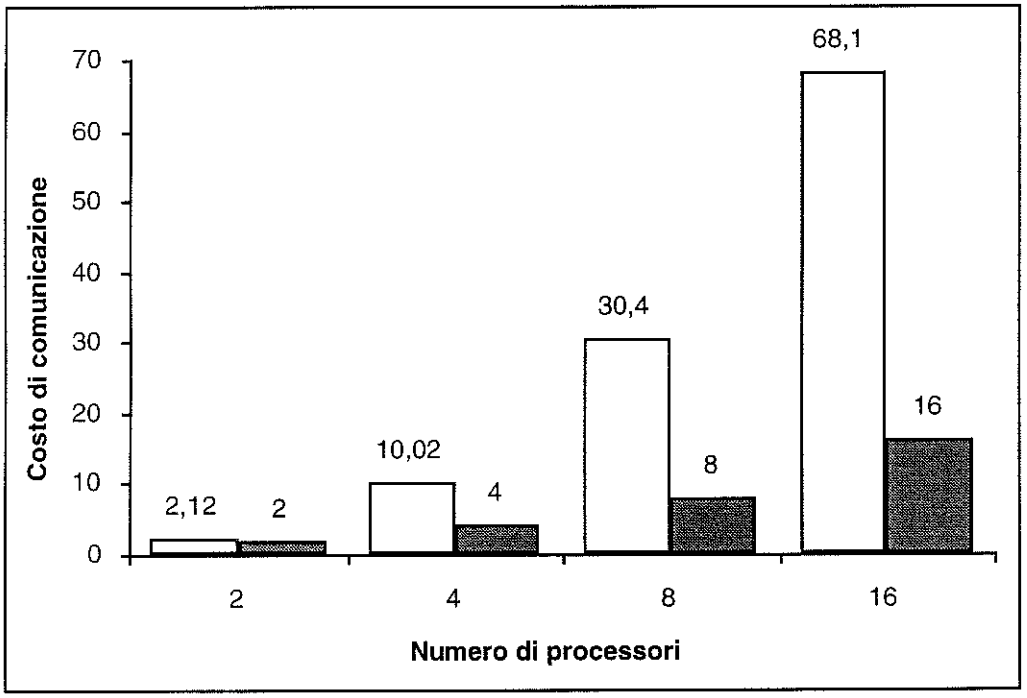


Figura 29: Costi di comunicazione con 15 processi per processore

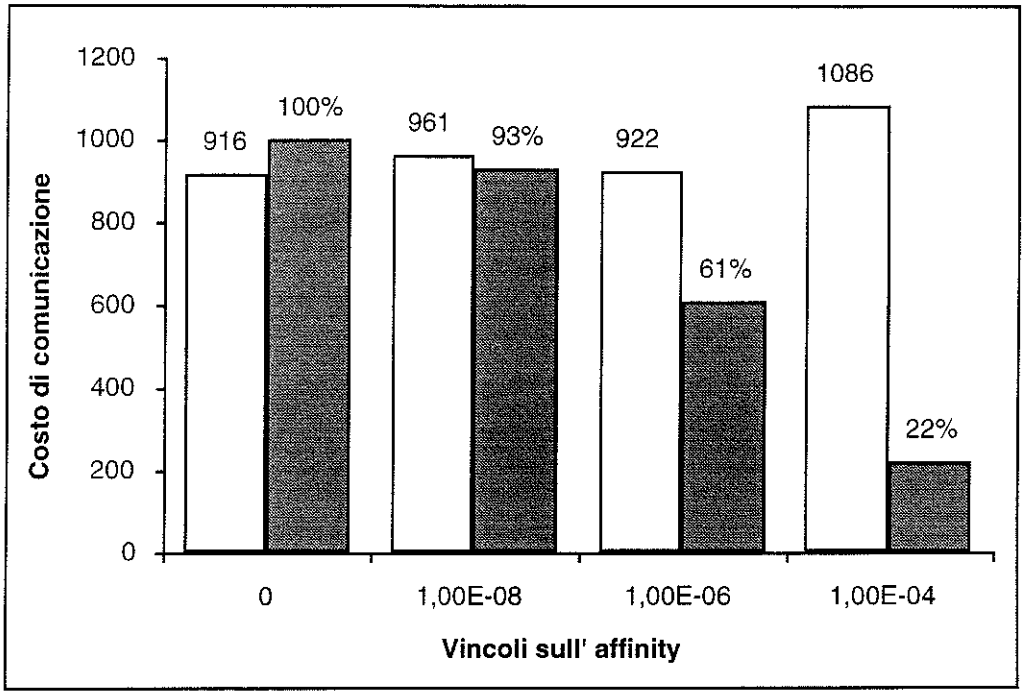


Figura 31: Costi di comunicazione in funzione dell'affinity (40 processi)

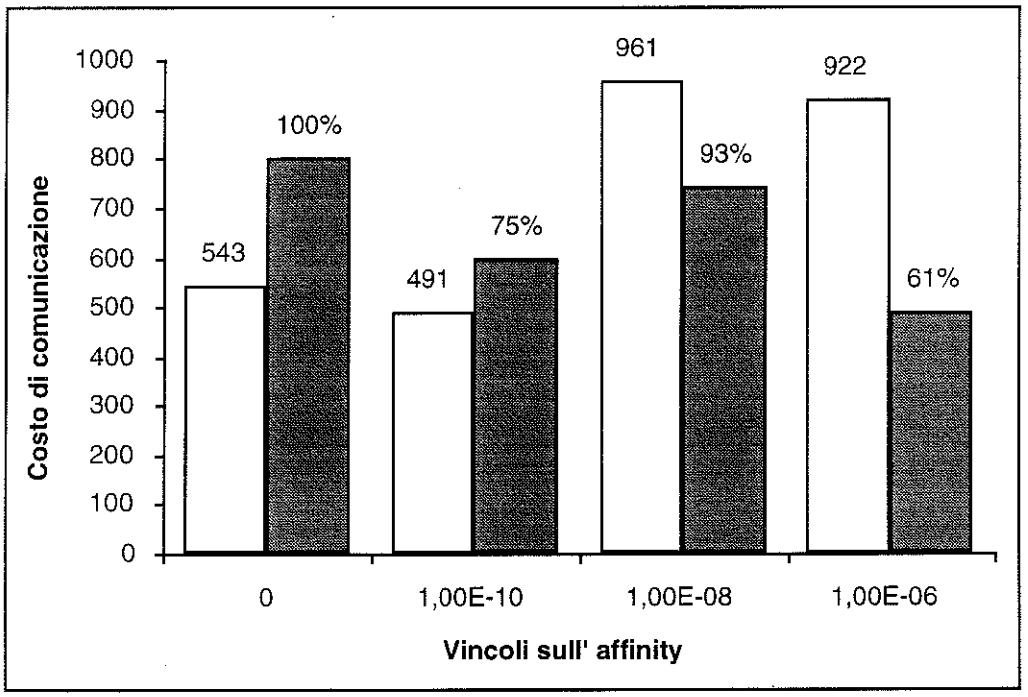


Figura 32: Costi di comunicazione in funzione dell'affinity (60 processi)

10 Limitazioni

L'algoritmo di mapping proposto soffre di alcune limitazioni. La prima limitazione, presente anche nell'algoritmo di Bowen et al., è la *unit workload assumption*, ossia il fatto di considerare il costo computazionale di ciascun processo pari a 1. In questo modo, la misura del carico assegnato ad un processore, effettuata contando il numero di processi, non è realistica. Infatti, a parità di numero di processi assegnati, il carico effettivo assegnato ad un processore può variare ampiamente a seconda delle funzioni svolte dai processi. Basta considerare il carico determinato da n processi che gestiscono altrettanti terminali e quello determinato da n processi che risolvono altrettanti sistemi lineari. Il vincolo in questione può essere rimosso al costo di riscrivere la funzione **Calc_Affinity** e la funzione **Calc_Load**. Con la *unit workload assumption* è possibile calcolare i valori di *affinity* e del carico indotto su un processore considerando i processi "in blocchi". Pertanto, i valori di *affinity* e di carico possono essere determinati con poche e semplici operazioni. Ciò è possibile in quanto il peso di un cluster di processi corrisponde al numero di processi che lo compone. Senza questa semplificazione, è necessario considerare i processi uno alla volta e determinare i valori di *affinity* e di carico esaminando un processo per volta e ripetendo per ciascuno tutti i controlli. Poiché la *unit workload assumption* non modifica l'essenza dell'algoritmo ma ne riduce solo l'accuratezza, abbiamo evitato di complicare inutilmente l'algoritmo stesso. Per gli scopi delle prove da effettuare, questa semplificazione non pregiudica la significatività dei risultati. In situazioni di impiego reale, il vincolo può essere rimosso, in modo da aumentare l'accuratezza dell'algoritmo.

La seconda limitazione consiste nel fatto che non sempre l'algoritmo riesce a trovare una soluzione accettabile; come già menzionato, esistono casi in cui l'algoritmo ideato termina senza fornire una soluzione. Questo problema deriva dall'introduzione del vincolo sulla *affinity*. L'assegnamento dei processi ai processori viene effettuato attraverso "raffinamenti successivi". All'inizio un grande cluster di processi viene assegnato a un grande cluster di processori; successivamente, ciascun componente del cluster di processi viene assegnato a ciascun componente del cluster di processori, in maniera ricorsiva. Nell'algoritmo originario era possibile prevedere se un assegnamento effettuato ad un certo livello provocava il fallimento nelle fasi successive ed era pertanto possibile prevenire tali situazioni. Nel nostro algoritmo, al contrario, non è possibile effettuare queste previsioni e pertanto esiste la possibilità di terminare senza una soluzione. Tuttavia, dalle prove effettuate è emerso che l'algoritmo ha sempre trovato una soluzione nel caso omogeneo e le percentuali di successo dei test eterogenei sono abbastanza alte.

Riferimenti

[Bok81] Shadid H. Bokhari. On the mapping problem. *Journal of Parallel and Distributed Computing*, c-30(3):207–214, March 1981.

[SM94] S. Selvakumar and C. Siva Ram Murthy. Static task allocation of concurrent programs for distributed computing systems with processor and resource heterogeneity. *Parallel Computing*, 20:835–851, 1994.

[Fre89] Richard F. Freund. Optimal selection theory for superconcurrency. In *Proceedings of Supercomputing '89*, pages 699–703, 1989.

[CEKS93] Song Chen, Mary M. Eshaghian, Ashfaq Khokhar, and Muhammad E. Shaaban. A selection theory and methodology for heterogeneous supercomputing. In *Proceedings of the Seventh International Parallel Processing Symposium Workshop on Heterogeneous Computing*, pages 15–22, April 1993.

[Ber90] Alan Albert Bertossi. *Strutture Algoritmi Complessità*. ECIG, 1990.

[DEP94] J. C. DeSouza Batista, Mary M. Eshaghian, A. C. Parker, S. Prakash, and Y. C. Wu. A sub-optimal assignment of application tasks onto heterogeneous systems. In *Proceedings*

of the Eighth International Parallel Processing Symposium Workshop on Heterogeneous Processing, April 1994.

[LP92] Chokchai Leangsuksun and Jerry Potter. Problem representation for an automatic mapping algorithm on heterogeneous processing environments. In *Proceedings of the Sixth International Parallel Processing Symposium Workshop on Heterogeneous Computing*, March 1992.

[LP94] Chokchai Leangsuksun and Jerry Potter. Designs and experiments on heterogeneous mapping heuristics. In *Proceedings of the Eighth International Parallel Processing Symposium Workshop on Heterogeneous Computing*, April 1994.

[BNG92] Nicholas S. Bowen, Christos N. Nikolaou, and Arif Ghafoor. On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *IEEE Transactions on Computers*, 41(3):257–273, March 1992.

[GY93] Arif Ghafoor and Jaehyung Yang. A distributed heterogeneous supercomputing management system. *IEEE Computer*, 26(6):78–86, June 1993.