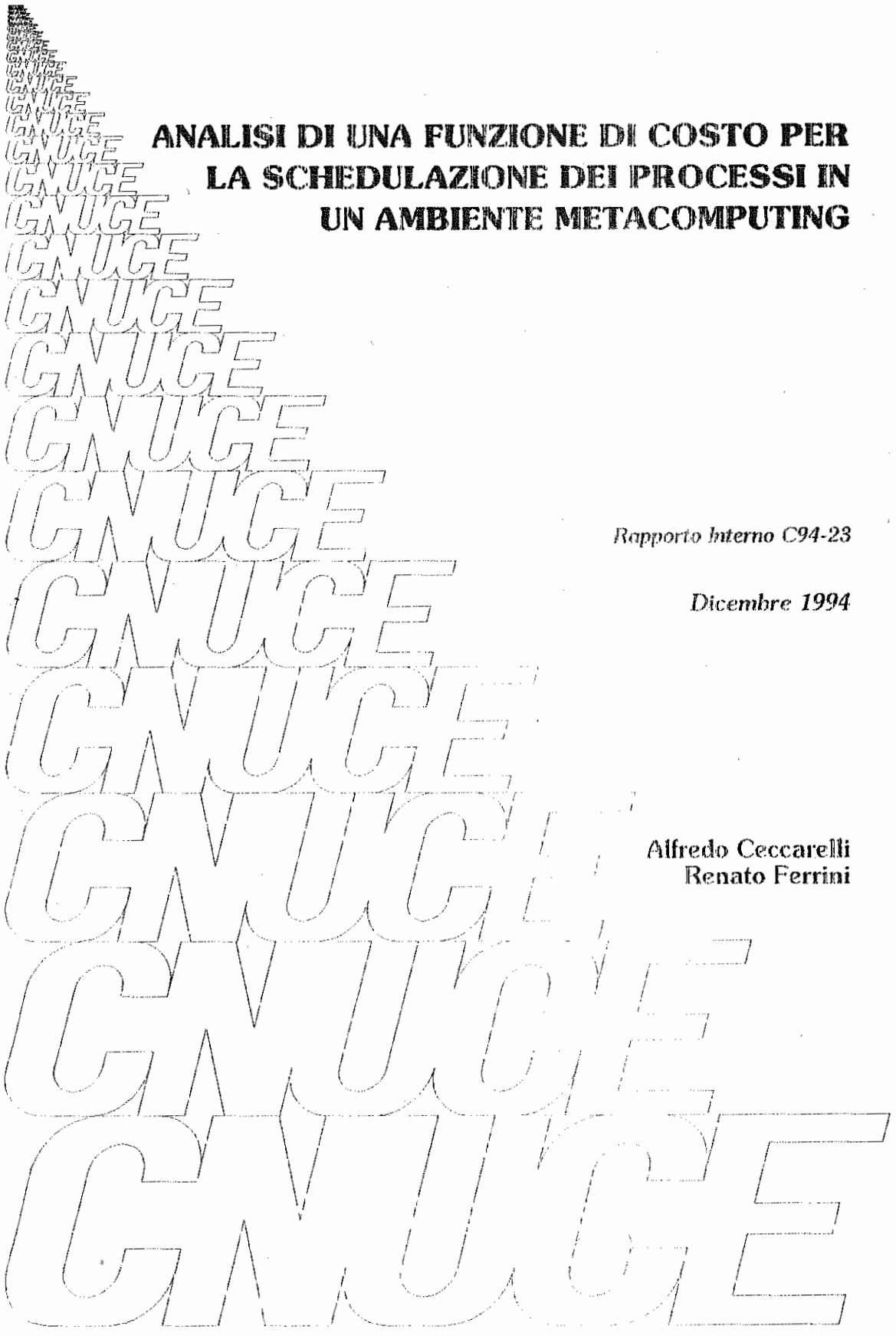
# ANALISI DI UNA FUNZIONE DI COSTO PER LA SCHEDULAZIONE DEI PROCESSI IN UN AMBIENTE METACOMPUTING




*Rapporto Interno C94-23*

*Dicembre 1994*

Alfredo Ceccarelli
Renato Ferrini

# Analisi di una funzione di costo per la schedulazione dei processi in un ambiente Metacomputing

**Alfredo Ceccarelli**

**Renato Ferrini**

**Rapporto Interno C94-23**

**Pisa, Dicembre 1994**

# Indice

# Introduzione

Negli ultimi anni la nascita di forme di parallelismo distribuito ha aperto la strada a nuove possibilità di calcolo, ma ha anche introdotto una serie di problematiche che devono essere affrontate e risolte per un pieno sfruttamento di tutte le risorse. La presenza contemporanea in uno stesso sistema di architetture eterogenee che basano i propri principi di funzionamento su paradigmi differenti impone una struttura dell'applicazione molto complessa e quindi di difficile gestione da parte di quegli utilizzatori che non hanno delle specifiche competenze informatiche. Occorre pertanto sviluppare da una parte gli strumenti che aiutino a svolgere questa attività in modo abbastanza semplice e dall'altra il software che consenta automaticamente un buon utilizzo delle macchine parallele presenti nel sistema distribuito.

In quest'ottica abbiamo affrontato uno dei principali problemi presente in un sistema di calcolo distribuito di tipo *metacomputing* e che è rappresentato dalla allocazione ottimale dei task sulle differenti architetture parallele. Il suddetto problema, che per altro esiste anche nelle altre forme di parallelismo e va sotto il nome di *mapping*, non è stato ancora affrontato in maniera sistematica e quindi non esiste in letteratura alcun riferimento che possa aiutare il programmatore ad eseguire l'allocazione in modo soddisfacente.

Questo documento riporta i risultati di uno studio che è stato svolto per definire un indice (o funzione di costo) che permetta di valutare l'adattabilità di un determinato codice ad una particolare architettura. Tale elemento potrà in futuro essere usato per determinare se una particolare allocazione dei task sulle macchine presenti nel *metacomputing* è ottimale o meno.

# Il Metacomputing ed il Mapping

## Il Network Computing ed il Cluster Computing

L'evoluzione tecnologica degli ultimi anni ha permesso di sviluppare forme di parallelismo nuove che si configurano come casi particolari dei *Sistemi di Calcolo Distribuito* (SCD). Si ricorda che un sistema di calcolo distribuito consiste di un insieme di nodi di elaborazione in grado di eseguire un proprio flusso di istruzioni, che non condividono alcuna memoria principale e cooperano scambiandosi messaggi attraverso una rete di comunicazione. Uno di questi nuovi sistemi di parallelismo è rappresentato dal *Network Computing* che indica l'utilizzo contemporaneo di una rete composta da elaboratori aventi caratteristiche architetturali e prestazioni diverse tra loro; si tratta quindi di un sistema di calcolo distribuito eterogeneo e di tipo *loosely-coupled.*

Il *Network Computing* costituisce una nuova risorsa per il calcolo parallelo ad un costo molto contenuto e con un alto livello di scalabilità, cioè con la possibilità di espandere la configurazione o con la possibilità di arricchire la configurazione con macchine più specializzate. L'altro lato della medaglia del *Network Computing* è legato a problemi connessi soprattutto alla sicurezza delle informazioni ed alla gestione di un parco macchine eterogeneo. Soprattutto il primo punto è particolarmente avvisato dagli utenti che spesso non vedono positivamente l'idea di dover condividere la propria workstation con altri che potrebbero aver accesso alle proprie informazioni private. L'altro problema è legato anche al fatto di dover installare e mantenere il software necessario per le comunicazioni tra le varie stazioni di lavoro.

Ma allo stato attuale l'ostacolo maggiore che limita il raggiungimento delle massime prestazioni nel *Network*

*Computing* resta la rete di interconnessione, perchè la tecnologia relativa alle reti locali sia a bassa velocità (Ethernet) che a media velocità (FDDI, HiPPI) non garantisce un trasferimento di dati paragonabile a quello dei sistemi *tightly-coupled.* Pertanto il *Network Computing* si rivolge a modelli di parallelismo a grana grossa, perchè il numero dei nodi del sistema è generalmente basso, e con uno scambio di informazioni abbastanza limitato.

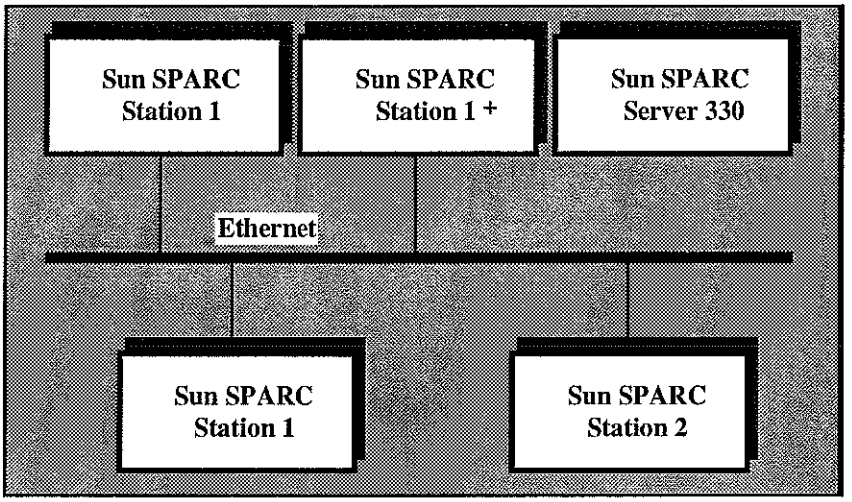



*Figura 1 - Cluster Computing ottenuto con rete Ethernet*

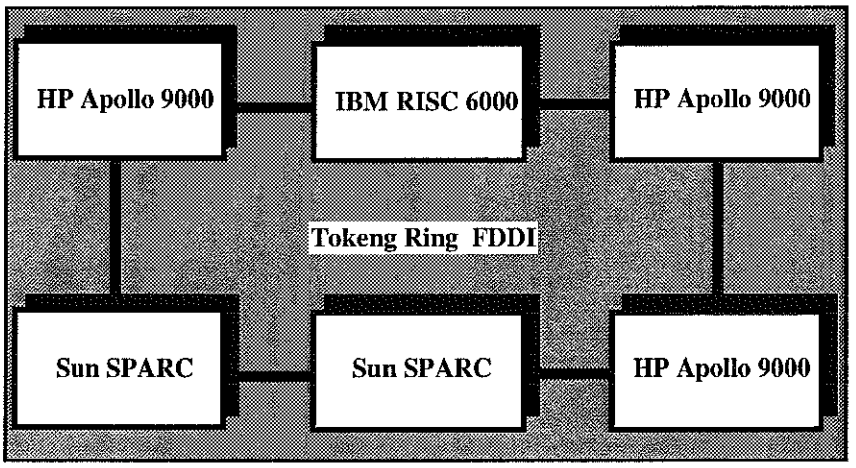



*Figura 2 - Cluster Computing ottenuto con rete FDDI*

Anche l'eterogeneità potrebbe comportare un aggravio di

lavoro in fase di programmazione, in quanto la rappresentazione dei dati sulle varie architetture potrebbe essere diversa e quindi si renderebbero necessari degli interventi per uniformare le informazioni.

Un caso particolare del *Network Computing* è rappresentato dal *Cluster Computing* che si ha quando tutti i nodi della rete sono costituiti da workstation. Nelle figure 1 e 2 sono riportati due esempi di *Cluster Computing* , in cui sono state usate le stazioni di lavoro più diffuse a livello commerciale connesse tra loro da due tipi di rete locale.

Il *Cluster Computing* può essere omogeneo sia di primo livello, quando è composto solamente da workstation della stessa casa costruttrice, che di secondo livello, quando le stazioni di lavoro sono identiche. Questa caratteristica elimina in parte alcuni inconvenienti visti per il *Network Computing* e rende questa forma di parallelismo realizzabile con meno sforzi.

## Il Metacomputing

Un altro caso di un *Network Computing* è rappresentato dal *Metacomputing* che costituisce un passo ulteriore sulla strada dell'eterogeneità. A differenza del *Cluster Computing*, dove delle comuni stazioni di lavoro vengono raggruppate per formare un sistema di una potenza di calcolo non indifferente, il *Metacomputing* è costituito da macchine specializzate per il calcolo parallelo, come architetture vettoriali o architetture ad ipercubo, o che svolgono delle funzioni particolari, come sistemi di acquisizione o memorizzazione di grosse quantità di dati. Con questa definizione, quindi, un *Cluster Computing* può essere una delle componenti di un sistema *Metacomputing*.
L'insieme delle macchine specializzate che compongono il *Metacomputing* non sono mai tutte concentrate nello stesso luogo ma sono distribuite su di un'area che a volte può essere di diverse centinaia di chilometri quadrati, per cui la rete di interconnessione costituisce un grosso ostacolo all'efficienza delle prestazioni computazionali. Il vantaggio, però, di disporre di più sistemi specializzati nel calcolo parallelo in un unico ambiente di lavoro consente di risolvere i problemi legati al fatto che molte applicazioni reali sono intrinsecamente eterogenee. Infatti nella realtà non tutte le componenti di un codice ottengono i tempi di esecuzione migliori su di una stessa architettura parallela. Quindi il profilo computazionale dell'applicazione non è mai omogeneo ma composto da più parti che meglio si adattano alle varie piattaforme

computazionali.

Nella figura 3 è mostrato graficamente il profilo dell'esecuzione sequenziale di una tipica applicazione del campo scientifico e/o ingegneristico, in cui vengono evidenziate le percentuali del tempo di calcolo che meglio si adattano alle varie piattaforme di elaborazione. Se una tale applicazione venisse eseguita su di un supercalcolatore vettoriale con 4 processori si avrebbero dei sensibili benefici per quanto riguarda la parte vettorizzabile del codice (*Vector*) che subirebbe una riduzione del 30%, mentre le restanti parti avrebbero una modesta diminuzione dato il limitato livello di parallelismo dell'architettura. In totale il tempo di esecuzione dell'applicazione verrebbe ridotto di una percentuale del 55% consentendo così di ottenere una velocità di esecuzione più che doppia rispetto al corrispondente tempo di esecuzione sequenziale.

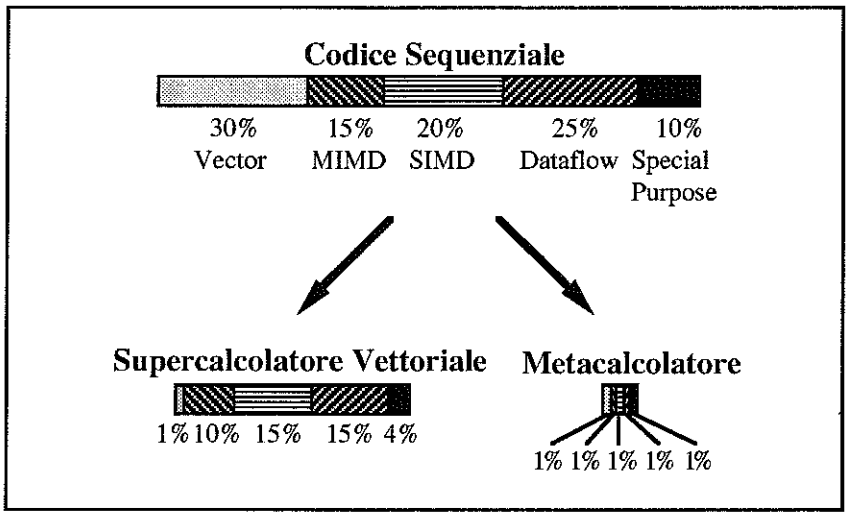



*Figura 3 - Profilo computazionale di un codice campione*

Nell'ipotesi, invece, di eseguire l'applicazione su di un metacalcolatore, dove è possibile allocare ogni componente sull'opportuna architettura, tutte le percentuali sarebbero ridotte al minimo raggiungendo delle prestazioni 20 volte superiori a quelle dell'esecuzione sequenziale. Il *Metacomputing* rappresenta quindi una nuova e promettente via per soddisfare le sempre più crescenti richieste di calcolo, in alternativa alla tecnica di incrementare in modo massiccio il numero dei processori su di una stessa macchina o di migliorare la tecnologia dei componenti hardware.

## Il *Mapping*

Nell'ambito del calcolo parallelo, ed in modo particolare nel *Metacomputing*, assume grande importanza l'allocazione statica, o dinamica, dei task sui vari processori presenti nel sistema e questa operazione è comunemente denominata *mapping*. In genere l'obiettivo è quello di trovare tra tutte le possibili combinazioni quella che minimizza (tempo globale di esecuzione, sbilanciamento del carico, ecc...) o massimizza (tolleranza ai guasti) alcuni parametri che sono ritenuti più importanti ai fini della produzione finale. Per agevolare questa operazione il programma parallelo viene configurato come un grafo, i cui nodi rappresentano i task e gli archi le comunicazioni tra i task, ed analogamente il sistema parallelo è codificato come un altro grafo, in cui i nodi rappresentano i processori e gli archi le connessioni bidirezionali tra i processori. Con questa esemplificazione il problema del *mapping* viene scomposto in due sottoproblemi:

1. *variazione topologica*
   In generale il grafo dei task ed il grafo dei processori non sono isomorfi, cioè non esiste alcuna simmetria tra le due strutture

2. *variazione di cardinalità*
   Il numero dei task ed il numero dei processori possono essere diversi; nella realtà il numero dei task è quasi sempre superiore del numero dei processi disponibili nel sistema

In letteratura è possibile trovare molti algoritmi o tecniche che aiutano a svolgere in modo più o meno efficiente il *mapping* e che possono essere raggruppati nelle seguenti categorie:

1. *Algoritmi di Branch & Bound*

2. *Algoritmi di Local Search*

3. *Algoritmi di Greedy*

Gli algoritmi di *Branch & Bound* trovano la soluzione ottimale, ma, avendo una complessità (numero di operazioni eseguite) che

cresce in modo esponenziale all'aumentare del numero dei task o dei processori, possono essere usati per problemi di dimensioni ridotte. Anche se sono state introdotte delle funzioni euristiche per diminuire la ricerca nell'albero delle soluzioni, riducendo di conseguenza il tempo di esecuzione, questi algoritmi risolvono casi di *mapping* in cui il numero dei task è inferiore a 20 ed il numero dei processori non supera le 10 unità.

Gli algoritmi di *Local Search* trovano una soluzione sub-ottimale ma hanno il vantaggio di limitare il tempo richiesto per la ricerca. Partendo da una combinazione di *mapping* ammissibile, questi algoritmi migliorano iterativamente la soluzione fino a quando viene soddisfatta una condizione che indica la bontà del risultato finale. In alcuni casi il procedimento termina quando si è raggiunta la massima cardinalità dei due grafi, cioè quando il maggior numero di archi del grafo dei task coincide con altrettanti archi del grafo dei processori, mentre in altri casi la condizione da soddisfare prevede di minimizzare la quantità delle comunicazioni. Con tale tecnica si riescono a risolvere problemi di *mapping* fino ad una dimensione di 512 task e 512 processori.

Gli algoritmi di *Greedy* sono quelli che richiedono una minore quantità di tempo di calcolo per svolgere il *mapping* in quanto, oltre a ricercare una soluzione sub-ottimale come gli algoritmi di *Local Search*, garantiscono una complessità di ordine polinomiale e ciò è dovuto al fatto che seguono un unico *path* che va dalla radice ad una foglia dell'albero di tutte le possibili soluzioni. Per operare la scelta migliore gli algoritmi di *Greedy* usano una funzione dei costi molto accurata che, applicata all'allocazione parziale svolta fino ad un certo momento, permette di valutare la strada migliore da intraprendere. Queste tecniche consentono di risolvere problemi di *mapping* abbastanza complessi in tempi ragionevoli.

# Allocazione di un'applicazione nel Metacomputing

## Il mapping nel Metacomputing

Allo stato attuale il problema del *mapping* è stato ristretto ad una sola macchina parallela o a sistemi distribuiti di processori eterogenei che comunque possono essere assimilati ad un'unica macchina parallela. Sotto queste ipotesi è possibile schematizzare l'architettura di target sotto la forma di un grafo con le caratteristiche viste nel precedente capitolo. Nel *metacomputing* il problema si complica in quanto si ha un insieme di macchine parallele ognuna delle quali può essere rappresentata con un grafo, ottenendo così un insieme di grafi aventi tutti una diversa cardinalità e morfologia. Pertanto allocare il grafo dei task su di una struttura dei processori molto complessa comporta una complessità di un ordine di grandezza superiore.

Una semplificazione al problema può essere ottenuta sfruttando gli algoritmi già esistenti, che in pratica permettono di allocare un'applicazione su di una qualsiasi macchina parallela, e suddividendo il *mapping* in tre fasi:

1. *Conoscenza delle caratteristiche computazionali delle varie parti dell'applicazione*
2. *Allocazione delle suddette parti sulle architetture parallele più idonee che sono presenti nel metacomputing*
3. *Mapping di ogni parte sull'architettura parallela assegnata usando un algoritmo già esistente*

Con questa organizzazione il problema più gravoso è la soluzione del primo punto perchè richiede la ripartizione di tutta l'applicazione in *macro-processi* con caratteristiche computazionali abbastanza omogenee e la cui esecuzione sequenziale è ravvicinata. Svolta tale operazione il *mapping* diventa più semplice e consiste in una prima fase nel soddisfare le restrizioni imposte dal sistema, come ad esempio l'uso di un particolare software che è disponibile solo su di una macchina, e successivamente nel valutare, in base alle caratteristiche computazionali dei macro-processi, su quale processore si ottengono le prestazioni migliori.

Per agevolare quest'ultima fase è stato utilizzato un metodo che permette di stimare il tempo di esecuzione su di una particolare architettura. Questo obiettivo è stato ottenuto mediante la combinazione di due elementi: la definizione dei parametri computazionali del task e la conoscenza, ottenuta con misure sperimentali, delle capacità elaborative di una macchina.

Il primo punto ha richiesto l'implementazione di uno strumento, disponibile sotto il sistema UNIX, che ha consentito di ottenere le principali caratteristiche di un codice scritto nel linguaggio Fortran 77. Gli elementi caratterizzanti di una applicazione sono stati ricondotti a 4:

- la richiesta del tempo di calcolo
- l'uso della memoria centrale
- il numero delle operazioni di I/O
- la comunicazioni tra i processi

Per ognuno di essi sono stati poi individuati quegli elementi della programmazione Fortran che potessero fornire una stima più o meno esatta del loro consumo. Quindi il tempo di calcolo è stato assimilato alle quattro operazioni aritmetiche su variabili *floating-point* ed al numero delle chiamate a routine della libreria matematica, la memoria centrale con il numero di riferimenti a costanti e variabili scalari o vettoriali, sia intere che reali, le operazioni di I/O con il numero dei bytes trasferiti ed infine la comunicazione con il numero dei bytes inviati o ricevuti da un processo. Tutte queste informazioni sono presenti nella programmazione Fortran e sono facilmente ricavabili da una scansione del codice.

## Le caratteristiche computazionali

L'algoritmo dello strumento, scritto nel linguaggio C, si basa sulla ricerca di opportune parole chiave del linguaggio Fortran 77, che permettono di quantificare le quattro caratteristiche computazionali cercate. In base a quanto detto il risultato finale, che viene stampato dopo l'esame di tutto il codice, contiene quindi le seguenti informazioni:

1. numero delle operazioni di somma tra numeri reali
2. numero delle operazioni di sottrazione tra numeri reali
3. numero delle operazioni di moltiplicazione tra numeri reali
4. numero delle operazioni di divisione tra numeri reali
5. numero di chiamate alle routine della libreria matematica del Fortran
6. numero di accessi alla memoria centrale
7. numero di byte trasferiti in operazioni di Lettura/Scrittura su memoria esterna
8. numero di byte trasferiti nella comunicazione tra processi

La finalità di stimare il tempo di esecuzione di un task su di una particolare architettura parallela, allo scopo di avere un'indicazione precisa del grado di adattabilità del codice alle caratteristiche computazionali della macchina, ha reso necessario misurare il tempo unitario richiesto per lo svolgimento di ognuna delle otto operazioni sopra elencate.

## Misura del tempo di esecuzione

Le informazioni che caratterizzano le prestazioni di un sistema sono abbastanza difficili da reperire poichè, come spesso accade, le case costruttrici non forniscono gli elementi per risalire al dato cercato. Quindi, per capire il funzionamento di una macchina non resta che ricorrere a dati sperimentali mediante alcune prove pratiche.

La metodologia di misura viene così ad assumere una notevole importanza nella rilevazione dei tempi sopra specificati. Infatti, se non si tenesse conto di eventuali fenomeni indotti i tempi rilevati potrebbero essere alterati con conseguente propagazione dell'errore sul risultato finale. Inoltre, dovendo misurare delle operazioni che richiedono tempi dell'ordine dei nanosecondi, è necessario operare con strumenti molto accurati per realizzare una precisione almeno del millisecondo.

Il criterio seguito per misurare il tempo di calcolo delle operazioni aritmetiche tra due numeri reali in doppia precisione è schematizzato nel codice di figura 4. L'esempio riportato si riferisce all'operazione di somma ma è facilmente estendibile alle altre operazioni aritmetiche.

```
Tstart = clock()
         for ( i=0; i<n; ++i) {
              a = a + b;
         }
Tstop = clock()
Tempo = Tstop - Tstart
Tsomma = Tempo/n
```

*Figura 4 - Codice per la rilevazione dei tempi di calcolo delle operazioni aritmetiche*

La routine *clock()*, che si trova nella libreria di corredo della macchina fornisce il tempo di calcolo usato e viene richiamata immediatamente prima e dopo l'evento da misurare. La differenza dei due tempi rilevati e la successiva operazione di divisione della variabile *Tempo* per $n$ fornisce il tempo di calcolo richiesto da una singola operazione.

L'uso di una operazione ricorsiva su scalari ( $a = a + b$ ) consente di ottenere alcuni importanti vantaggi:

1. Il metodo usato consente di determinare il tempo di calcolo scorporato del tempo di accesso alle variabili perchè dopo un primo accesso alla memoria per leggere i valori delle variabili $a$ e $b$, le successive operazioni di addizione avvengono tra registri. Così facendo si hanno solo 2 accessi iniziali alla memoria il cui tempo diventa trascurabile rispetto al tempo necessario per eseguire $n$ operazioni di somma. Se nell'espressione le variabili usate fossero del tipo vettori o matrici, il tempo di calcolo dell'operazione comprenderebbe anche il tempo di

accesso alla memoria, poichè l'accesso ad ogni elemento del vettore richiederebbe una lettura di memoria.

2. Il compilatore è obbligato ad eseguire l'operazione tante volte quante sono specificate dall'indice del FOR. Ciò per evitare che, essendo le variabili *a* e *b* di tipo scalare, e quindi indipendenti dalle iterazioni del ciclo, alcuni compilatori siano indotti a portare l'operazione fuori dal ciclo per eseguirla una volta soltanto. Se ciò accadesse si avrebbe un grosso errore nella rilevazione del tempo cercato.

3. La possibilità di rilevare il tempo di calcolo su uno spettro di valori degli operandi molto ampio consente di ottenere una durata media dell'operazione più significativa e costituisce un ulteriore vantaggio della espressione ricorsiva. Infatti, il tempo di esecuzione di un'operazione aritmetica varia al variare del contenuto delle variabili interessate; se dunque i valori interessati sono ogni volta diversi tra loro si ottiene un valore medio del tempo di calcolo più attendibile.

Ricordiamo inoltre che la routine *clockO* fornisce il tempo espresso in *microsecondi* e dunque, per non incorrere in errori dovuti a problemi di precisione, è necessario che il tempo da misurare sia almeno della durata di alcune decine di *millisecondi.* Per questa ragione la misura viene eseguita con *n=1.000.000.*

```
for (i=0; i<1000; ++i) {
          v[i] = 1.5 + i ;
          }
Tstart = clock()
for ( i1=0; i1<n; ++i1) {
          for (i2=0; i2<1000; ++i2) {
          w[i2] = 0. ;
          w[i2] = v[(n-1) - i2] ;
          }
}
Tstop = clock()
Tacc. = (Tstop - Tstart)/2n
```

*Figura 5 - Codice per la rilevazione dei tempi di accesso alla memoria*

Un altro accorgimento si è reso necessario in quanto il codice usato per la determinazione del tempo di calcolo comprende

anche le operazioni per la gestione dell'istruzione *FOR* che richiedono una certa quantità di tempo. Il valore del tempo di *FOR* è stato scorporato da quello totale tramite una prova successiva senza alcuna istruzione nel ciclo, consentendo così di ottenere il tempo effettivo dell'operazione di somma come differenza dei due valori precedenti.

Il tempo di accesso alla memoria centrale costituisce la seconda importante informazione che caratterizza le prestazioni di una macchina. Come nelle misure precedenti, anche in questo caso, il metodo di misura ha una grande importanza per l'attendibilità dei valori rilevati. Il criterio adottato, esplicitato nel codice di figura 5, consiste nella continua inversione del vettore $V$ precedentemente inizializzato.

Il tempo di accesso alla memoria viene rilevato attraverso $n$ assegnazioni tra due vettori. L'uso di elementi sempre diversi dà la certezza che venga sempre eseguito l'accesso in memoria allo specifico elemento. In questo modo sono evitati eventuali accorgimenti di ottimizzazione da parte del compilatore che porterebbero a ridurre il numero di accessi alla memoria con conseguente imprecisione del tempo rilevato.

| **Operazione** | **NCUBE2** | **RISC** | **SP/2** | **SUN** | **HP** |
|---|---|---|---|---|---|
| Addizione | 0.52 | 0.20 | 0.06 | 1.17 | 0.20 |
| Sottrazione | 0.60 | 0.10 | 0.06 | 1.00 | 0.20 |
| Prodotto | 0.76 | 0.10 | 0.06 | 1.17 | 0.30 |
| Divisione | 1.76 | 0.70 | 0.32 | 1.50 | 0.40 |
| Accessi in Memoria | 0.53 | 0.15 | 0.10 | 0.66 | 0.15 |

*Tabella 1 - Tempo di calcolo in microsecondi richiesto per ogni singola operazione.*

Anche in questo caso, per non incorrere in errori dovuti a problemi di precisione, è necessario ripetere l'operazione più volte in modo da ottenere dei tempi finali dell'ordine di alcune decine di millisecondi.

## Risultati ottenuti su alcune macchine

I codici per la misura dei tempi sono stati eseguiti sulle seguenti macchine: NCUBE2, RISC 6000/320H, SP/2, SUN, HP 9000/720 che costituiscono l'ambiente di sperimentazione Metacomputing presente al CNUCE. I risultati ottenuti, espressi in *microsecondi*, sono riportati nella tabella 1.

Come detto all'inizio, il tempo di calcolo di un task comprende anche il tempo di esecuzione delle routine della libreria matematica che sono richiamate all'interno del processo. Il codice di prova per effettuare queste rilevazioni è indicato nella figura 6.

```
vett[0] = 1.1;
for (i1=1; i1<1000; ++i1) {
vett[i1] = vett[i1-1] + 1.1 };
Tstart = clock()
for ( i1=0; i1<n; ++i1) {
          for (i2=0; i2<1000; ++i2) {
          abs (vett [i2]) };
}
Tstop = clock()
TRoutine = Tstart - Tstop
```

*Figura 6 - Codice per la rilevazione dei tempi delle routine di sistema.*

Sono stati rilevati i tempi di alcune routine più usate come valore assoluto, radice quadrata, logaritmo naturale ed in base 10, seno, coseno, ecc... i cui risultati, espressi in microsecondi, sono indicati nella tabella 2. Il tempo di esecuzione di ciascuna routine è stato ottenuto con lo stessa metodologia seguita per la determinazione del tempo di calcolo delle operazioni aritmetiche e del tempo di accesso alla memoria.

| Routine | NCUBE 2 | RISC | SP/2 | SUN | HP |
|---|---|---|---|---|---|
| ABS | 3.32 | 1.00 | 0.34 | 1.16 | 0.80 |
| SQRT | 3.82 | 2.20 | 0.48 | 12.17 | 0.40 |
| LOG10 | 19.37 | 4.90 | 1.33 | 16.49 | 2.20 |
| LOGn | 19.48 | 3.10 | 0.72 | 6.92 | 1.90 |
| SIN | 17.16 | 2.10 | 0.66 | 9.67 | 2.00 |
| COS | 17.36 | 2.00 | 0.64 | 9.83 | 2.00 |

*Tabella 2 - Tempo di calcolo stimato, in microsecondi, di alcune routine della libreria matematica.*

## Esempio di uso dello strumento

Nel Rapporto Interno *C94-21* viene descritto uno strumento che consente di valutare le caratteristiche computazionali di una applicazione scritta nel linguaggio Fortran 77. Per verificare la validità dei risultati ottenuti si è preferito testare lo strumento su una routine reale piuttosto che su un caso fittizio implementato opportunamente. La routine, che viene riportata nell'appendice A, fa parte di una grossa applicazione e calcola le soluzioni di una equazione differenziale mediante il metodo delle differenze finite. Poichè il numero di volte che viene eseguita è molto alto, essa costituisce un processo che deve essere allocato con molta attenzione per ridurre il tempo totale di esecuzione.

I risultati parziali, forniti dal suddetto strumento e relativi, per ogni classe di operazione, sia ad ogni ciclo di DO presente nella routine che ai risultati totali, sono riportati nell'appendice B.

Determinato il numero ed il tipo di operazioni che

caratterizzano la routine *COOLEY* ed essendo noto il tempo di esecuzione di una singola operazione ricavabile dalla tabelle 1 e 2, si può ottenere il tempo di esecuzione di ciascun gruppo di operazione per ogni macchina facente parte dell'ambiente di sperimentazione.

| **Tipo di Operazione** | **NCUBE 2** | **RISC** | **SP/2** | **SUN** | **HP** |
|---|---|---|---|---|---|
| Addizione | 0.005 | 0.002 | 0.0006 | 0.012 | 0.002 |
| Sottrazione | 0.024 | 0.004 | 0.0024 | 0.040 | 0.008 |
| Prodotto | 0.038 | 0.005 | 0.0030 | 0.058 | 0.015 |
| Divisione | 17.70 | 7.04 | 3.220 | 15.09 | 4.024 |
| Accessi in memoria | 16.14 | 4.57 | 3.046 | 20.10 | 4.57 |
| ABS | 0.033 | 0.010 | 0.003 | 0.011 | 0.008 |
| SQRT | 0.0001 | 0.00006 | 0.00001 | 0.00001 | 0.00001 |
| I/O | 0.00076 | 0.00080 | 0.0002 | 0.00216 | 0.00008 |
| TOTALE | 33.9408 | 11.6318 | 6.2752 | 36.0232 | 8.6271 |

*Tabella 3 - Stima del tempo di esecuzione, in secondi, della routine COOLEY.*

Così, se la routine fosse eseguita sulla macchina *nCube2*, il tempo di calcolo totale relativo alle operazioni di addizione, ottenuto moltiplicando il numero di addizioni della routine per il tempo di esecuzione speso da una singola operazione, sarebbe 0.005

sec. Anche il tempo speso per le operazioni di moltiplicazione, calcolato nello stesso modo, sarebbe 0.038 sec. mentre il tempo per gli accessi in memoria risulterebbe uguale a 16.14 sec.. e così via. Mentre il dato relativo all'I/O è ottenuto moltiplicando il numero di byte trasferiti in una o più operazioni di lettura-scrittura su disco, per il tempo necessario a trasferire un singolo byte.

I risultati ottenuti su ogni macchina, che sono riportati nella tabella 3, inducono ad alcune considerazioni. Ad esempio nel confronto dei tempi relativi alle macchine RISC e HP il calcolo delle operazioni di divisione richiede un tempo complessivo pari a 7.04 sec. sulla prima e di 4.024 sec. spesi sulla seconda. Mentre per gli accessi in memoria che costituiscono la seconda più importante voce del tempo di calcolo della routine, hanno lo stesso valore globale. La routine potrà perciò, essere eseguita in minor tempo sulla macchina HP rispetto ad una RISC.

E' importante osservare che, qualora il numero delle operazioni di moltiplicazione avesse un peso maggiore, impiegando questa operazione un significativo minor tempo di esecuzione sulla macchina RISC, potrebbe risultare più conveniente eseguire il task su questa macchina.

Una seconda osservazione deriva dal valore totale di esecuzione della routine sulla macchina nCube2. I valori indicati nella colonna nCube indicano che l'esecuzione della routine su questa macchina, sarebbe in ogni caso penalizzante rispetto alle altre macchine. E' altresì opportuno rilevare che si potrebbe avere una più efficace esecuzione qualora la routine comunicasse con altri processi. In questo caso la rete di comunicazione di nCube costituirebbe senz'altro un notevole vantaggio rispetto alla più lenta rete di comunicazione Ethernet che collega le varie workstation. La conseguenza sarebbe una drastica diminuzione del tempo di esecuzione della routine nonostante i valori superiori spesi per il calcolo.

Comunque, molto importante risulta il dato finale relativo al tempo di esecuzione della routine sulle varie macchine. Ecco così che abbiamo raggiunto il nostro scopo e cioè conoscere, seppure in modo approssimato, il tempo totale speso da un task sulle macchine che fanno parte dell'ambiente *Metacomputing*. In base a questo valore è possibile schedulare il task su quella macchina che meglio si addice alle sue caratteristiche computazionali.

Nel caso specifico, la routine in esame è candidata per essere eseguita sulla macchina SP/2.

## Conclusioni

L'uso di nuove forme di parallelismo, come il *Network Computing* od il *Metacomputing*, ha consentito di disporre di grosse quantità di calcolo con relativa facilità o di sfruttare pienamente il livello di parallelismo di un'applicazione. D'altra parte ha introdotto una serie di nuovi problemi riconducibili soprattutto alla eterogeneità dell'ambiente di lavoro. Uno di questi problemi è rappresentato dalla schedulazione ottimale dei task di cui si compone un'applicazione sulle varie macchine che costituiscono l'ambiente parallelo.

In questo rapporto tecnico è stata definita una funzione di costo che fornisce un indice della durata del tempo di esecuzione di ciascun task su ogni architettura. Partendo infatti da uno strumento con cui si ricavano le principali caratteristiche computazionali di un codice scritto nel linguaggio Fortran 77, è stato possibile risalire al tempo di esecuzione mediante la creazione di una tabella dei tempi unitari di ciascun tipo di operazione. Questi valori sono stati ricavati sperimentalmente su un campione di macchine che sono di larga diffusione commerciale e che quindi hanno un'alta probabilità di essere una componente di un *Network Computing* o di un *Metacomputing*.

In ogni caso, al fine di consentire ad un qualsiasi utente di crearsi la tabella dei valori unitari con altri tipi o modelli di macchine, è stata descritta dettagliatamente la metodologia di misura che deve essere usata per rilevare i tempi desiderati. Il procedimento da seguire è stato reso molto semplice e di facile uso; infatti consiste essenzialmente nell'eseguire, sotto il sistema UNIX, una serie di piccoli programmi scritti in linguaggio C che permettono di rilevare i tempi unitari di esecuzione dei vari tipi di operazione. Ovviamente sono stati introdotti alcuni accorgimenti per evitare di incorrere in errori durante la fase di misurazione.

La tabella dei valori unitari consente di evidenziare le caratteristiche computazionali delle differenti architetture e questa conoscenza potrebbe ritornare utile anche per altri tipi di lavori informatici che non sono strettamente collegati al calcolo parallelo. Nell'ambito invece del calcolo parallelo la tabella consente di risalire ai tempi di esecuzione e quindi rappresenta una funzione di costo che può essere utilizzata in un algoritmo di *mapping*.

# Appendice A

```
      SUBROUTINE COOLEY(KV,E,N)
      IMPLICIT REAL*8(A-H,O-Z)
*  MAXGRID:  MAXIMUM  NUMBER  OF  POINTS  IN  THE  POTENTIAL
GRID.
      PARAMETER (MAXGRID=1000)
*
      DIMENSION Y(3)
      COMMON/VVV2/ ERROR,H,RMAX,MAXIT
      COMMON/CCC1/ POT(MAXGRID),P(MAXGRID),XX(2*MAXGRID)
C
      H2=H*H
      HV=H2/12.D0
      TEST=-1.D0
      DE=0.D0
C
C
C
      DO 79 IT=1,MAXIT
C
C       INTEGRATION INWARD
C
      P(N)=1.D-35
      GN=POT(N)-E
      GI=POT(N-1)-E
      IF(GI.LT.0.D0) GO TO 117
      P(N-1)=P(N)*DEXP(RMAX*DSQRT(GN)-(RMAX-H)*DSQRT(GI))
      Y(1)=(1.D0-HV*GN)*P(N)
      Y(2)=(1.D0-HV*GI)*P(N-1)
      M=N-2
   29 Y(3)=2.D0*Y(2)-Y(1)+H2*GI*P(M+1)
      GI=POT(M)-E
      P(M)=Y(3)/(1.D0-HV*GI)
      APM=DABS(P(M))
      IF(APM.LE.DABS(P(M+1))) GO TO 41
      IF(APM.LT.1.D+60 .AND. APM.GT.1D-60) GO TO 35
      PM=P(M)
      DO 32 ITM=M,N
   32 P(ITM)=P(ITM)/PM
      Y(2)=Y(2)/PM
      Y(3)=Y(3)/PM
   35 IF(M.LE.2) GO TO 41
      Y(1)=Y(2)
      Y(2)=Y(3)
```

```
      M=M-1
      GO TO 29
   41 PM=P(M)
      MSAVE=M
      YIN=Y(2)/PM
      DO 42 J=M,N
   42 P(J)=P(J)/PM
C
C       INTEGRATION OUTWARD
C
      P(1)=1.D-55
      Y(1)=0.D0
      GI=POT(1)-E
      Y(2)=(1.-HV*GI)*P(1)
      DO 52 I=2,M
      Y(3)=2.D0*Y(2)-Y(1)+H2*GI*P(I-1)
      GI=POT(I)-E
      P(I)=Y(3)/(1.D0-HV*GI)
      API=DABS(P(I))
      IF(API.LT.1.D+60 .AND. API.GT.1.D-60) GO TO 50
      PM=P(I)
      DO 45 ITM=1,M
   45 P(ITM)=P(ITM)/PM
      Y(2)=Y(2)/PM
      Y(3)=Y(3)/PM
   50 Y(1)=Y(2)
      Y(2)=Y(3)
   52 CONTINUE
      PM=P(M)
      IF(PM.EQ.0.D0) GO TO 65
      YOUT=Y(1)/PM
      YM=Y(3)/PM
      DO 58 J=1,M
   58 P(J)=P(J)/PM
C
C
      DF=0.D0
      DO 61 J=1,N
   61 DF=DF-P(J)*P(J)
      F=(-YOUT-YIN+2.D0*YM)/H2+POT(M)-E
      DOLD=DE
      GO TO 69
   65 F=999999D+29
      DF=-F
      DE=DABS(0.0001*E)
      GO TO 74
   69 DE=-F/DF
   74 EOLD=E
      E=E+DE
      TEST=DMAX1(DABS(DOLD)-DABS(DE),TEST)
      IF(TEST.LT.0.D0) GO TO 79
      IF(DABS(DE).LT.ERROR) GO TO 82
C
   79 CONTINUE
C
   82 KV=0
```

```
      NL=N-2
      DO 96 J=3,NL
      IF (P(J)) 88,87,87
   87 IF (P(J-1)) 89,96,96
   88 IF (P(J-1)) 96,93,91
   89 IF (P(J+1)) 96,90,90
   90 IF (P(J-2)) 95,96,91
   91 IF (P(J+1)) 92,96,96
   92 IF (P(J-2)) 96,95,95
   93 IF (P(J+1)) 94,96,96
   94 IF (P(J-2)) 96,96,95
   95 KV=KV+1
   96 CONTINUE
      SN=DSQRT(-H*DF)
      DO 99 J=1,N
   99 P(J)=P(J)/SN
C
      RETURN
C
C
  117 WRITE(6,11111)  IT,E,POT(N-1)
11111 FORMAT('COOLEY--COOLEY--COOLEY'/,
     >' ITERATION',I3,5X,E12.4,'  GREATER THAN  ',E12.4)
      CALL ABORT
      RETURN
      END
```

| | | | |
|---|---|---|---|
| DO Label | 52 | (n. di iterazioni presunte | 1000) |
| | n. di addizioni | | 1000 |
| | n. di sottrazioni | | 3000 |
| | n. di moltiplicazioni | | 4000 |
| | n. di divisioni | | 1003000 |
| | n. di chiamate di libreria | | 1000 |
| | n. di accessi in memoria | | 3033000 |
| | n. di byte di I/O | | 0 |
| | n. di byte di comunicazione | | 0 |

| | | | |
|---|---|---|---|
| DO Label | 58 | (n. di iterazioni presunte | 1000) |
| | n. di addizioni | | 0 |
| | n. di sottrazioni | | 0 |
| | n. di moltiplicazioni | | 0 |
| | n. di divisioni | | 1000 |
| | n. di chiamate di libreria | | 0 |
| | n. di accessi in memoria | | 3000 |
| | n. di byte di I/O | | 0 |
| | n. di byte di comunicazione | | 0 |

| | | | |
|---|---|---|---|
| DO Label | 61 | (n. di iterazioni presunte | 1000) |
| | n. di addizioni | | 0 |
| | n. di sottrazioni | | 1000 |
| | n. di moltiplicazioni | | 1000 |
| | n. di divisioni | | 0 |
| | n. di chiamate di libreria | | 0 |
| | n. di accessi in memoria | | 4000 |
| | n. di byte di I/O | | 0 |
| | n. di byte di comunicazione | | 0 |

| | | | |
|---|---|---|---|
| DO Label | 79 | (n. di iterazioni presunte | 10) |
| | n. di addizioni | | 100300 |
| | n. di sottrazioni | | 40170 |
| | n. di moltiplicazioni | | 50140 |
| | n. di divisioni | | 10060080 |
| | n. di chiamate di libreria | | 10090 |
| | n. di accessi in memoria | | 304601280 |
| | n. di byte di I/O | | 0 |
| | n. di byte di comunicazione | | 0 |

DO Label 96 (n. di iterazioni presunte 1000)

| | |
|---|---|
| n. di addizioni | 0 |
| n. di sottrazioni | 0 |
| n. di moltiplicazioni | 0 |
| n. di divisioni | 0 |
| n. di chiamate di libreria | 0 |
| n. di accessi in memoria | 12.000 |
| n. di byte di I/O | 0 |
| n. di byte di comunicazione | 0 |

DO Label 99 (n. di iterazioni presunte 1000)

| | |
|---|---|
| n. di addizioni | 0 |
| n. di sottrazioni | 0 |
| n. di moltiplicazioni | 0 |
| n. di divisioni | 1000 |
| n. di chiamate di libreria | 0 |
| n. di accessi in memoria | 3000 |
| n. di byte di I/O | 0 |
| n. di byte di comunicazione | 0 |

Operazioni Totali SUBROUTINE COOLEY

| | |
|---|---|
| n. di addizioni | 10030 |
| n. di sottrazioni | 40172 |
| n. di moltiplicazioni | 50142 |
| n. di divisioni | 10060091 |
| n. di chiamate di libreria | 10092 |
| n. di accessi in memoria | 30460156 |
| n. di byte di I/O | 8012 |
| n. di byte di comunicazione | 0 |

# Bibliografia

V.A.F. Almeida, I.M.M. Vasconcelos, J.N.C. Arabe, *The effect of Heterogeneity on the Performance of Multiprogrammed Parallel System*, Technical Report RT014/91, Universidade Federal de Minas Gerais, Instituto de Ciencias Exatas, 1991

S. Antonelli, F. Baiardi, S. Pelegatti, M. Vanneschi, "A Static Approach to Process Mapping in Massively Parallel Systems", *Proceedings of IFIP Working Conference. on Parallel Processing*, 1988

S. Antonelli, F. Baiardi, S. Pelegatti, M. Vanneschi, "Communication Cost and Process Mapping in Massively Parallel System", *Technical Report TR 12/89*, 1989

S. Arunkumar, T. Chockalingam, "Randomized Heuristics for the Mapping Problem", *Journal of High Speed Computing,* vol. 4, n. 4, 1992, pp. 289-299

T. Baba, Y. Iwamoto, T. Yoshinaga, "A Network-Topology Independent Task Allocation Strategy for Parallel Computers", *Proceedings of Supercomputing,* 1990

F. Berman, L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures", *Journal of Parallel and Distributed Computing,* vol. 4, n. 5, 1987, pp. 439-458

S.H. Bokhari, "On the Mapping Problem", *IEEE Transactions on Computers,* vol. 30, n. 3, 1981, pp. 207-214

H.E. Bal, J.G. Steiner, A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys*, vol. 21, n. 3, 1989, pp. 260-322

R. Baraglia, D. Laforenza, R. Perego, *Cluster Computing: la programmazione parallela di reti di workstation*, Rapporto interno CNR C93-11, Pisa, 1993

A. Beguelin, J. Dongarra, V. Sunderam, A. Geist, R. Manchek, *Paradigms and Tools for Heterogeneous Network Computing*, Supercomputing '92, 1992

A. Benzoni, V. Sunderam, R. van de Geijn, *Matrix Factorization on a RISC Workstation Network*, High Performance Computing II, 1991

E.D. Brooks III, T. Nash, K.H. Winkler, *The role of Computational Cluster*, Supercomputing '92, 1992

N. Carriero, D. Gelenter, "How to Write a Parallel Program: a Guide to the Perplexed", *ACM Computing Surveys*, vol. 21, 1989, pp. 323-358

A. Ceccarelli, R. Ferrini, *Metodologia di valutazione delle prestazioni di un codice Fortran in un ambiente Network Computing*, Rapporto interno CNR C94-21, Pisa, 1994

G. Erbacci, A. Sarti, *Programmazione parallela e cluster di RISC 6000,* Notizie dal Cineca n.14, 1992

M.D. Ercegovac, *Heterogeneity Supercomputer Architectures*, Parallel Computing 7, 1988

G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, D.W. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall, 1988

R.F. Freund, S. Conwell, *Superconcurrency: a Form of Distributed Heterogeneous Supercomputing*, Supercomputing Review, 1990

Al Geist, J. Dongarra, *PVM3 User's Guide and Reference Manual,* Oak Ridge National Laboratory, 1993

K. Herbst, *The Dawning of the Age of Network Supercomputing,* Supercomputing Review, 1991

*The HP Cluster Computing Program*, Hewlett Packard Co., 1992

*The IBM RISC System/6000 processor,* IBM Journal of Research and Development, vol. 34, n. 1, 1990

*The IBM SP2 Command and Technical Reference,* IBM Manual SC23-3867, 1993

B. W. Kernighan, D. M. Ritchie, *Linguaggio C seconda edizione,* Gruppo Editoriale Jackson, 1989

A. Kolawa, *Writing Programs for Distributed (Networked) Computers*, Supercomputing '92, 1992

S.Y. Lee, J.K. Aggarwal, "A Mapping Strategy for Parallel Processing", *IEEE Transactions on Computers*, vol. G-36, n. 4, 1987

V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems", *IEEE Transactions on Computers*, vol. 37, n. 11, 1988, pp. 1384-1397

P.Y.R. Ma, E.Y.S. Lee, M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems", *IEEE Transactions on Computers*, vol. 31, n. 1, 1982, pp. 41-47

D. Menascè, V. Almeida, "Cost-Performance Analysis of Heterogeneity in Supercomputer Architectures", *Proceedings of Supercomputing '90*, IEEE Computer Society Press, 1990

D. Menascè, V. Almeida, *Heterogeneous Supercomputing: is it cost-effective?*, Supercomputing Review, 1991

*nCUBE 2 Series Supercomputer, Technical Overview*, NCUBE Co, 1989

*nCUBE 2 Programmer's Guide, release 3.0*, NCUBE Co, 1992

P. Sadayappan, F. Ercal, J. Ramanujam, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube", *Parallel Computing*, vol.13, n. 1, 1990, pp. 1-16

C.C. Shen, W.H. Tsai, "A Graph Matching Approache to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion", *IEEE Transactions on Computers*, vol. 34, n. 3, 1985, pp. 197-203

M. Schneider, *Tying the Knot Between Serial and Massively Parallel Supercomputing:Pittsburgh's Not-so-Odd Couple*, Supercomputing Review, 1991

J.B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks", *Journal of Parallel and Distributed Computing*, vol. 4, n. 2, 1987, pp. 342-362

L. Smarr, C.E. Catlett, "Metacomputing", *Communication of the ACM*, vol. 35, n. 6, 1992

*Sparc Compiler Fortran Reference Manual, V. 2.0*, Sun Microsystems Inc, 1992

*Sparc Compiler Fortran User's Guide, V. 2.0*, Sun Microsystems Inc, 1992

*Sun Fortran Reference Guide, release 1.4,* Sun Microsystems Inc.,*1991*

*Sun Fortran User's Guide, release 1.4,* Sun Microsystems Inc., *1991*

V.S. Sunderam, "Heterogeneous Network-based Concurrent Computing Environments", *Future Generation Computer Systems,* vol. 8, 1992, pp. 191-203