PROGRAMMAZIONE

DEGLI

ELABORATORI PARALLELI

Supercomputing Tools for Science and Engineering

Pisa - 4-7 Dicembre 1989

Il presente corso è stato preparato da alcuni ricercatori del Reparto "Calcolo Parallelo" dell'Istituto CNUCE-CNR di Pisa, sulla base di informazioni contenute in libri e riviste specializzate, nonché in manuali tecnici, di cui si riporta l'elenco nel capitolo "Bibliografia". Al fine di consentire la massima divulgazione degli argomenti trattati, gli autori autorizzano chiunque ad utilizzare detto materiale con preghiera di citarne la fonte di provenienza.

A cura di Pasquale Lazzareschi, Ranieri Baraglia, Raffaele Perego
Reparto "Calcolo Parallelo"
CNUCE - Istituto del Consiglio Nazionale delle Ricerche

PARTE I

INTRODUZIONE

AL CALCOLO PARALLELO

A cura di Pasquale Lazzareschi
Reparto "Calcolo Parallelo"
CNUCE - Istituto del Consiglio Nazionale delle Ricerche

Esistono diverse forme di parallelismo:

- **Parallelismo interno della cpu**
- **Multiprogrammazione**
- **Elaborazione vettoriale**
- **Multitask**
- **. . .**

In questo contesto intendiamo per parallelismo l'elaborazione contemporanea di 2 o piú parti di uno stesso programma su una macchina multiprocessore.

Elaborazione parallela, motivazioni:

- Diminuire il tempo di permanenza in macchina (elapsed time) di lavori importanti.

- Possibilità di risolvere problemi che richiederebbero un tempo di cpu eccessivo per un singolo processore.

- Sfruttare le possibilità di una macchina multiprocessore quando non è possibile utilizzarle completamente con la multiprogrammazione.

- Limiti tecnologici.

- Eccessivo costo di un uniprocessore super veloce.

- Basso costo di processori con velocità ragionevole.

Elaborazione parallela, Svantaggi:

- Il tempo di cpu generalmente aumenta
- Nuovi compilatori e strumenti
- Nuovi algoritmi
- Programmazione piú difficile
- Debug piú difficile
- La diminuizione del tempo di permanenza in macchina avviene utilizzando dei processori che potrebbero essere utilizzati da altri job.

Elaborazione parallela, terminologia

Task

Una sezione logica discreta di lavoro computazionale

Task parallelo

Una sezione indipendente di lavoro computazionale. Può essere eseguito contemporaneamente ad altri task paralleli.

Un task parallelo può essere costituito da:

- Iterazioni diverse di un DO loop.
- Alcune sequenze di istruzioni.
- Subroutines.

Sincronizzazione

La coordinazione dei task paralleli.
Può essere:

- Attesa che alcuni o tutti i task abbiano raggiunto un certo punto prima di continuare
- Serializzare l'accesso in scrittura, a locazioni di memoria comuni, da parte di task paralleli

Elaborazione parallela, terminologia

Granularità

Dimensione dei task paralleli relativamente alla dimensione del programma

Coarse grain

I task sono grandi (tempo di cpu)

Fine grain

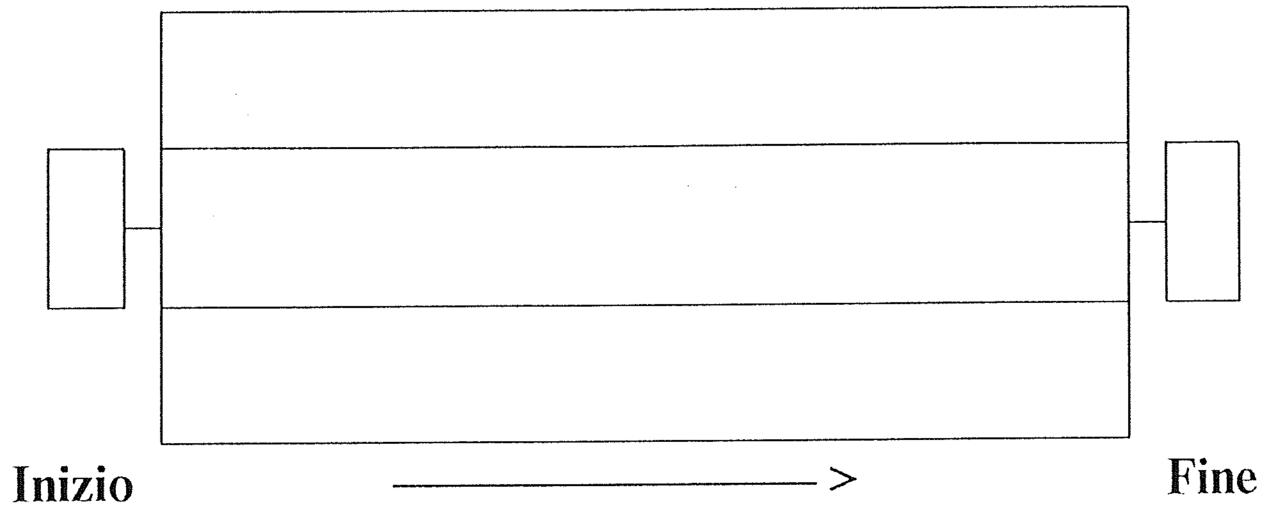
I task sono piccoli

Overhead

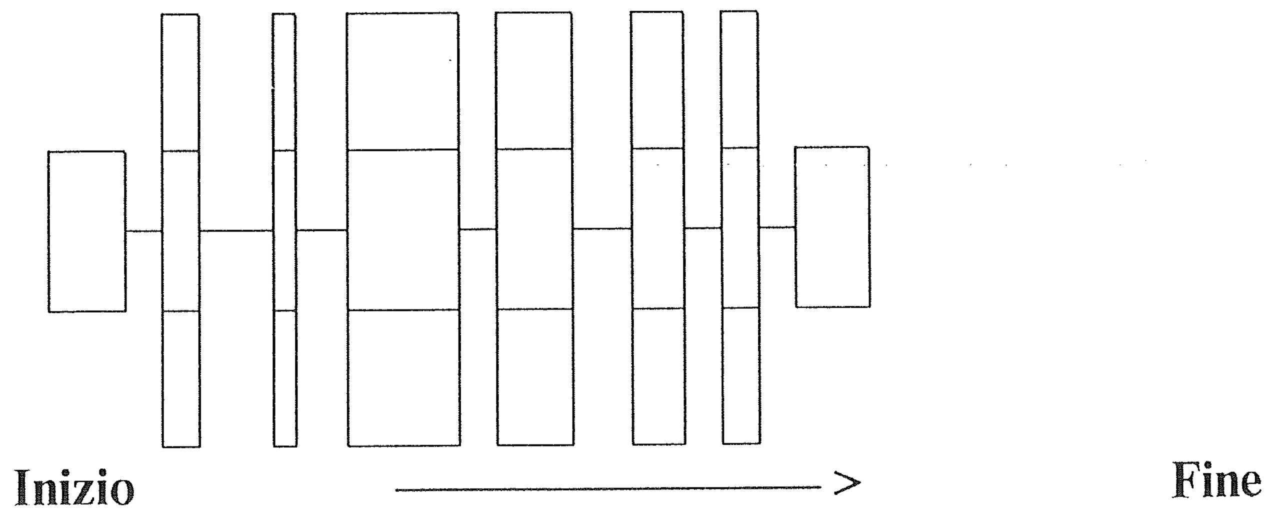
Tutto il lavoro aggiunto per il controllo del parallelismo.
(Sincronizzazione schedulazione e terminazione dai task)

Elaborazione parallela, Terminologia.

Coarse grain



Fine grain



MODELLI DI PARALLELISMO

Shared memory

I task paralleli hanno accesso alla stessa memoria

Message passing

I task paralleli comunicano trasmettendosi i dati attraverso una rete di comunicazione

Master slave

I task "slave" non possono comunicare direttamente tra loro ma solo con il master

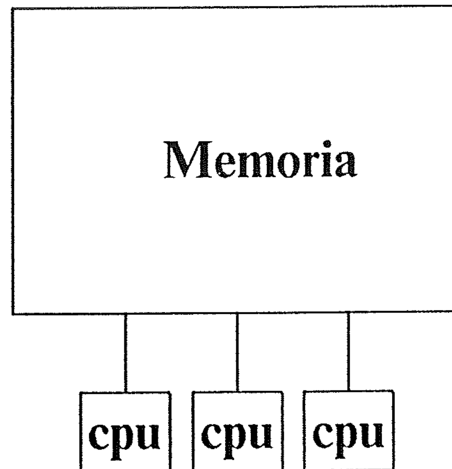
Peer to peer

Ogni task può comunicare con tutti gli altri

Modelli ibridi

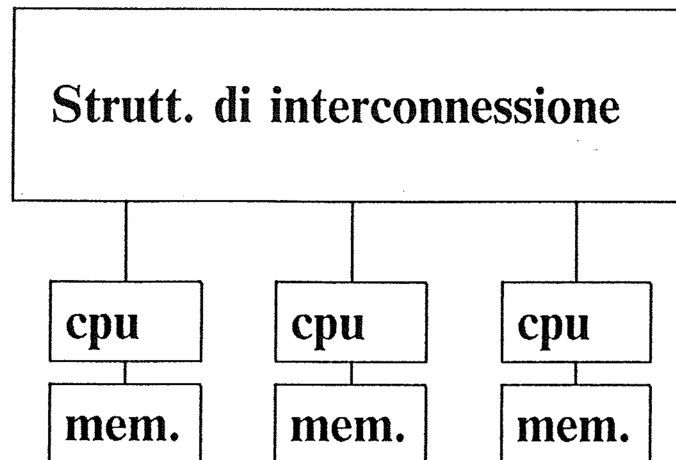
Una combinazione dei due precedenti

SHARED MEMORY



- I task comunicano per mezzo della memoria comune
- Overhead proporzionale alla sincronizzazione
- Tempo di accesso ai dati comuni trascurabile
- Una locazione di memoria comune non deve essere modificata da un task mentre un altro task parallelo la sta usando. (è necessario un meccanismo di "lock")

MESSAGE PASSING



- I task comunicano scambiandosi messaggi.

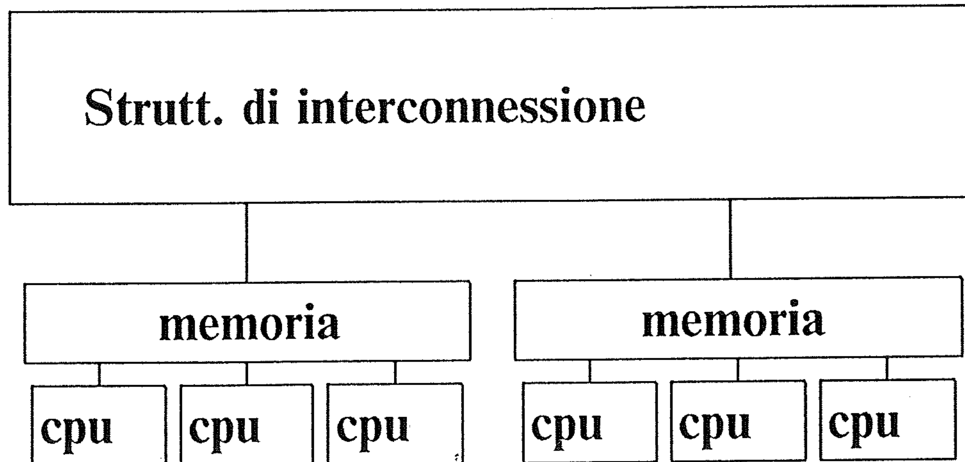
- Overhead proporzionale al numero e alla dimensione dei messaggi.

- Il tempo di trasmissione dei dati è maggiore del tempo di accesso alla memoria.

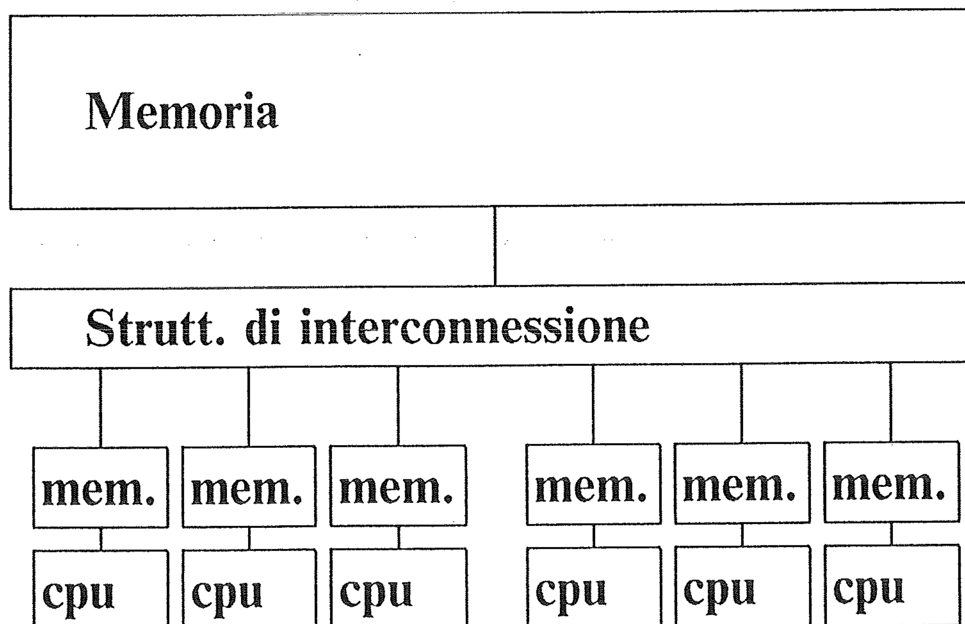
MODELLI IBRIDI

Varie possibilità

Esempio 1

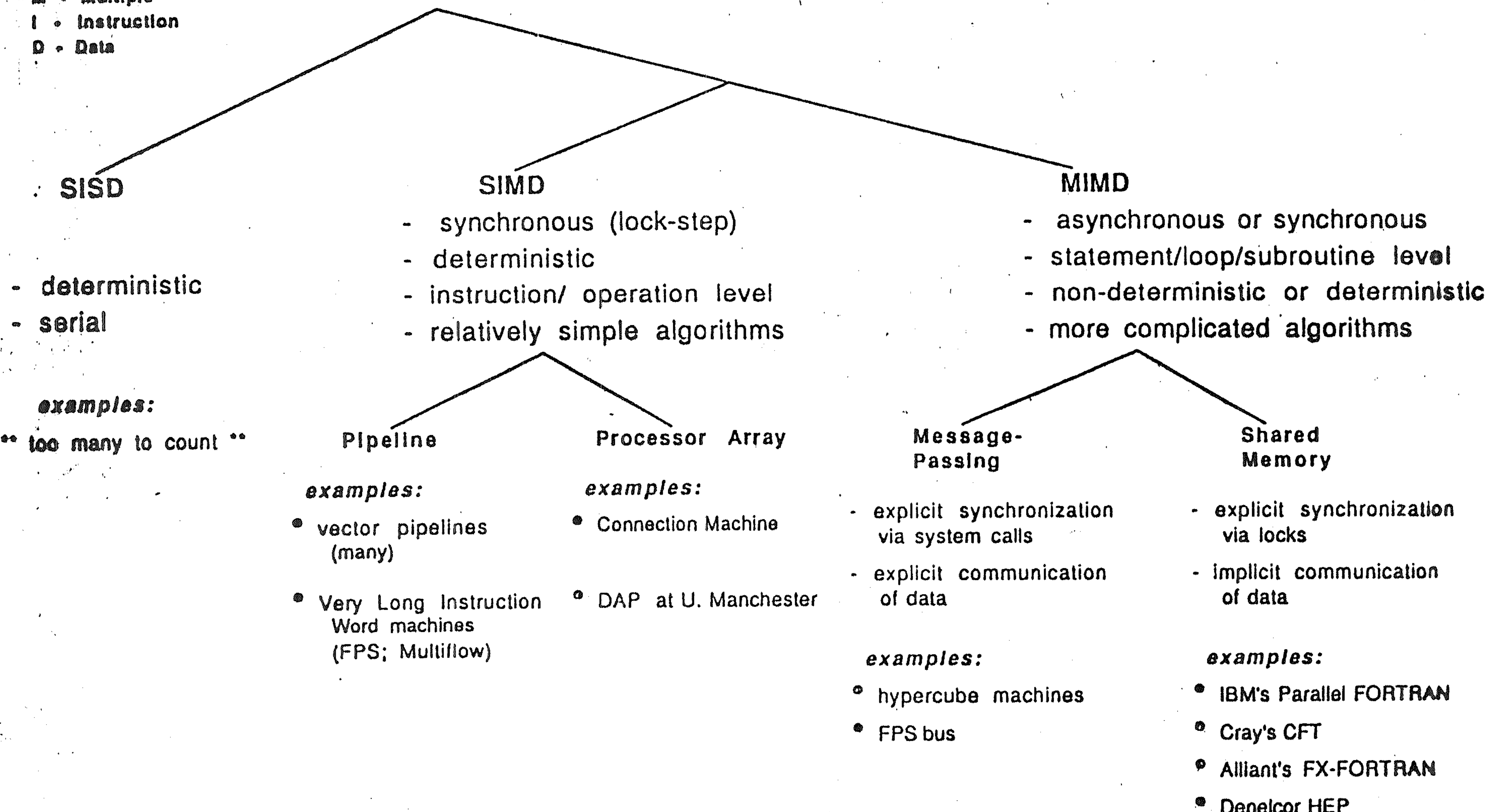


Esempio 2



Serial and Parallel Processing Schemes

(Flynn's 1966 classification)



Elaborazione parallela.

Esempio di tempi (programma cpu bound)

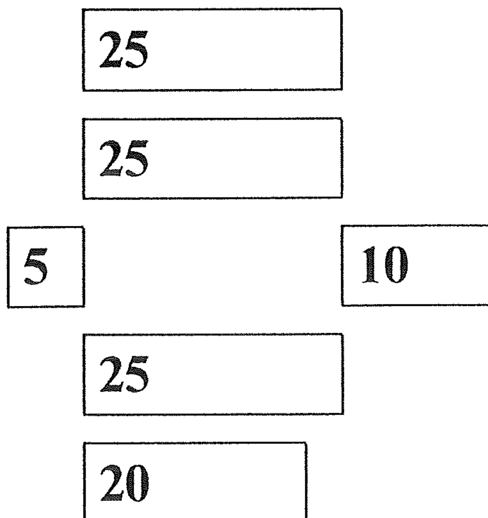
Scalare

Tempo CPU = 100
Tempo di permanenza in macchina = 100

Vettoriale

Tempo CPU = 50
Tempo di permanenza in macchina = 50

Parallelo



Tempo CPU = 110
Tempo di permanenza in macchina = 40

Elaborazione parallela.

Parallel speedup

$$\text{Speedup} = \frac{\text{Tempo di permanenza in macchina prog. seriale}}{\text{Tempo di permanenza in macchina prog. parallelizzato}}$$

Elaborazione parallela.

Limiti del Parallel speedup

Tempo di cpu di un programma seriale

Seriale	Parallelizzabile

0 10 100

Se *p* è la percentuale di codice che potrebbe essere eseguita in parallelo.

La percentuale di codice che rimane seriale dopo la parallelizzazione sarà *100-p*

100-p	p

Se *n* è il numero di processori disponibili il tempo di cpu della parte di codice parallelizzabile sarà suddiviso sugli *n* processori

Il tempo di esecuzione del programma parallelizzato sarà:

$$100 - p + \frac{p}{n}$$

Elaborazione parallela.

Limiti del Parallel speedup

Il massimo speedup ottenibile sarà limitato dalla parte di codice che rimane seriale

Su *n* processori il massimo speedup ottenibile è dato da:

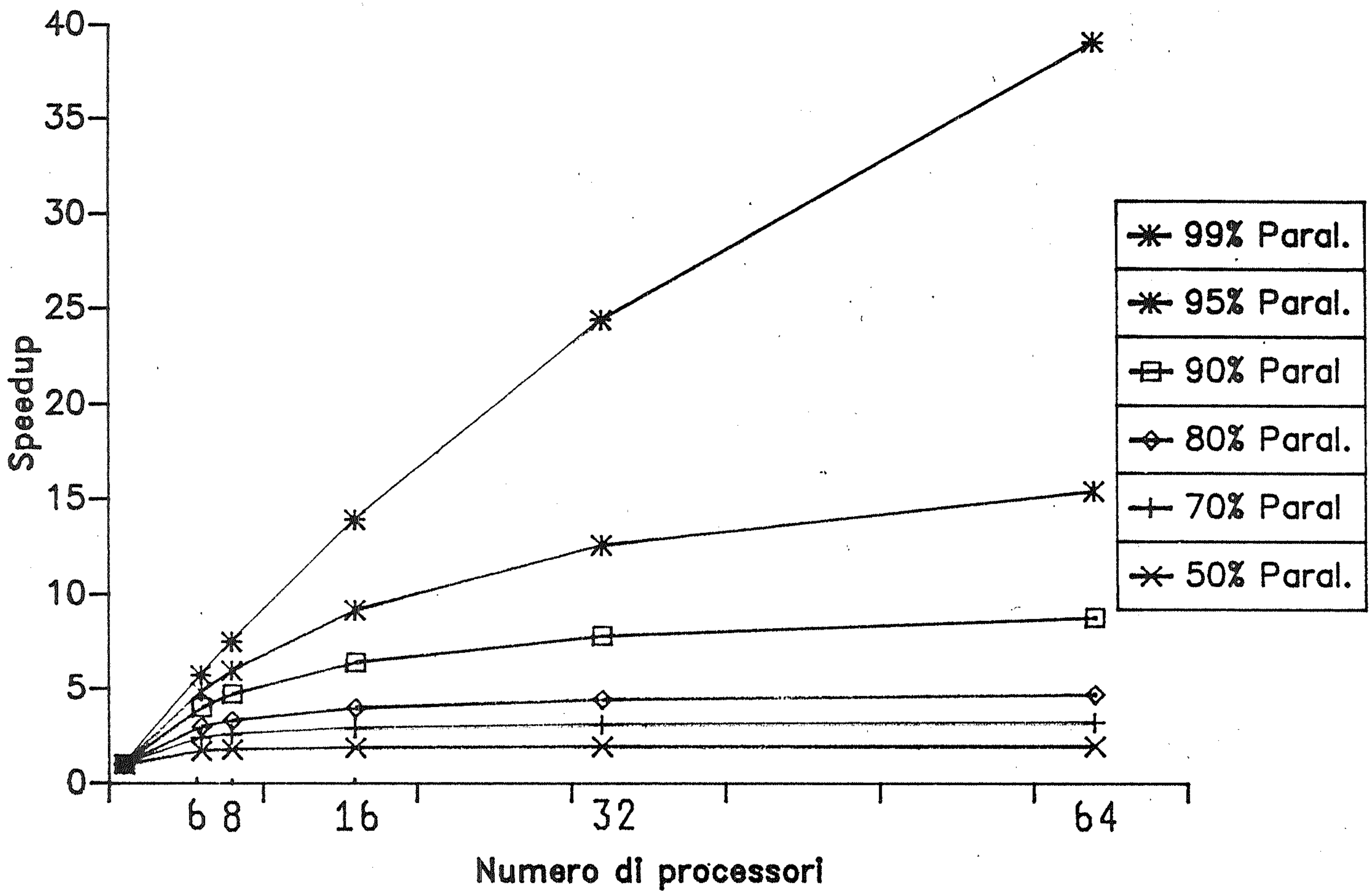
$$\text{Speedup} = \frac{\text{tempo seriale}}{\text{tempo parallelo}} = \frac{100}{100 - p + \frac{p}{n}}$$

Se indichiamo con *f* la frazione di codice parallelizzabile il massimo speedup è dato da:

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{n}} = \frac{n}{n(1 - f) + f}$$

Questa relazione è nota come legge di Amdahl

Parallel Speedup



Elaborazione parallela.

Conoscendo la percentuale di codice parallelizzabile presente in un programma seriale, la legge di Amdahl permette di calcolare il massimo speedup ottenibile.

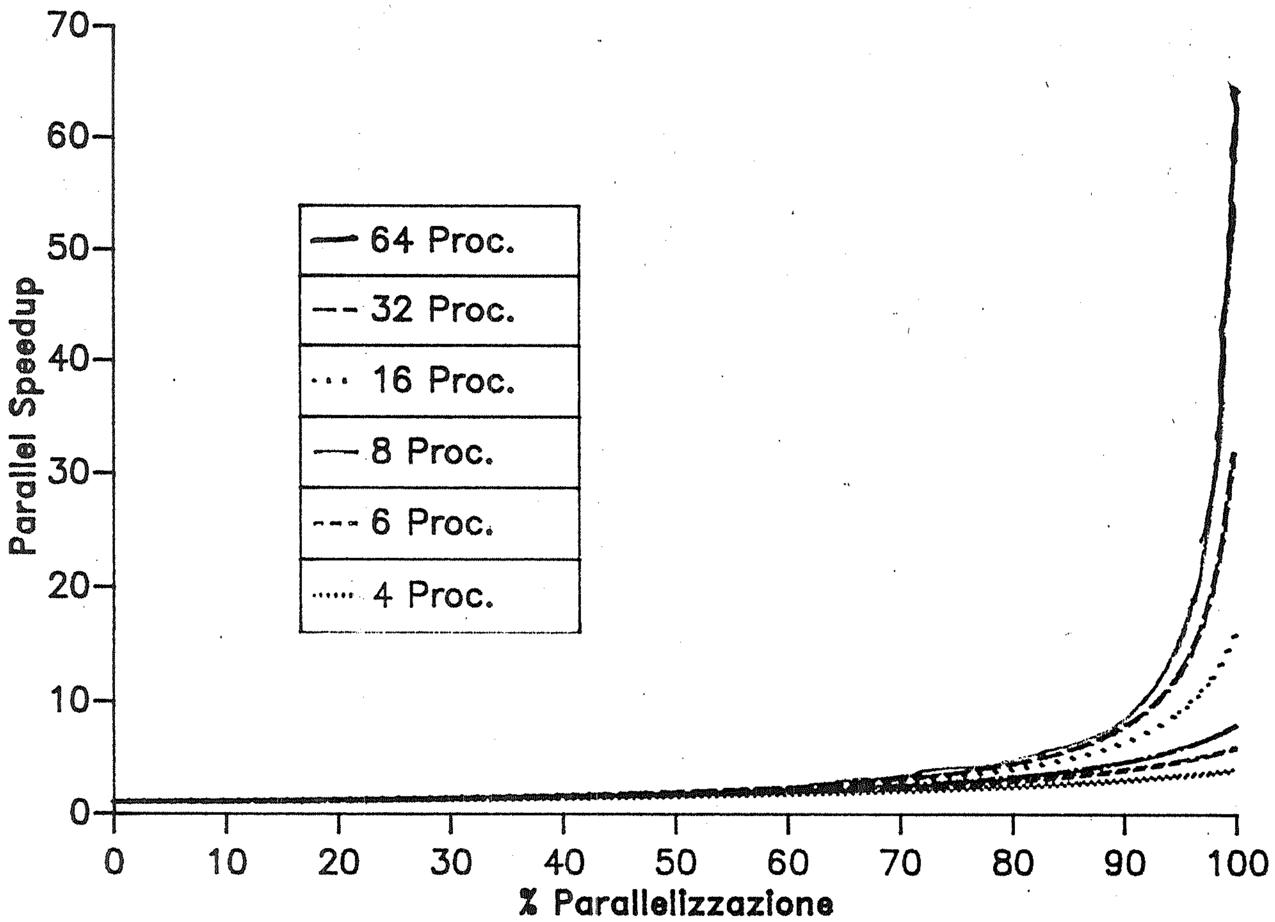
La legge di Amdahl non tiene conto degli overhead dovuti alla implementazione del parallelismo per cui lo speedup reale sarà sempre inferiore a quello calcolato.

Massimo speedup calcolato al variare della percentuale di codice parallelizzabile.

% Par.	CPU	Speedup
40	6	1.5
50	6	1.7
60	6	2.0
70	6	2.4
80	6	3.0
90	6	4.0
95	6	4.8
99	6	5.7

Tenendo conto degli overhead (espresso come percentuale del tempo di cpu) si può calcolare lo speedup che possiamo aspettarci.

Parallel Speedup



Elaborazione parallela in ambiente multiprogrammato

Lo speedup reale può essere anche molto diverso da quello ci si aspetta e può variare da esecuzione a esecuzione.

I vari task di un programma parallelo devono essere allocati, per l'esecuzione sulle CPU reali.

Questa allocazione viene fatta dal sistema operativo. Il quale deve servire anche tutti gli altri lavori contemporaneamente in esecuzione.

Su una macchina completamente carica il programma parallelo viene piú o meno serializzato.

Un programma parallelo ha un vantaggio rispetto al corrispondente seriale:

- Se il carico del sistema diminuisce (qualche CPU inutilizzata) il programma parallelo può utilizzarla.

Elaborazione parallela.

Efficienza di un programma parallelo

Definita come: (in percentuale)

$$\text{Efficienza} = \frac{\text{speedup}}{\text{numero di processori}} \times 100$$

Speedup ed efficienza al variare del numero di processori:

% Par.	CPU	Speedup	% Efficienza
99	2	1.9	99
99	4	3.9	97
99	8	7.4	93
99	16	13.9	87
99	64	39.3	61
99	128	56.3	44
99	256	72.1	28
99	512	83.8	16
99	1024	91.2	8

Confronto tra vettoriale parallelo

Differenza importante

- La vettorizzazione riduce il tempo di CPU
  a spese di una parte di hardware che se non utilizzata rimane inattiva.

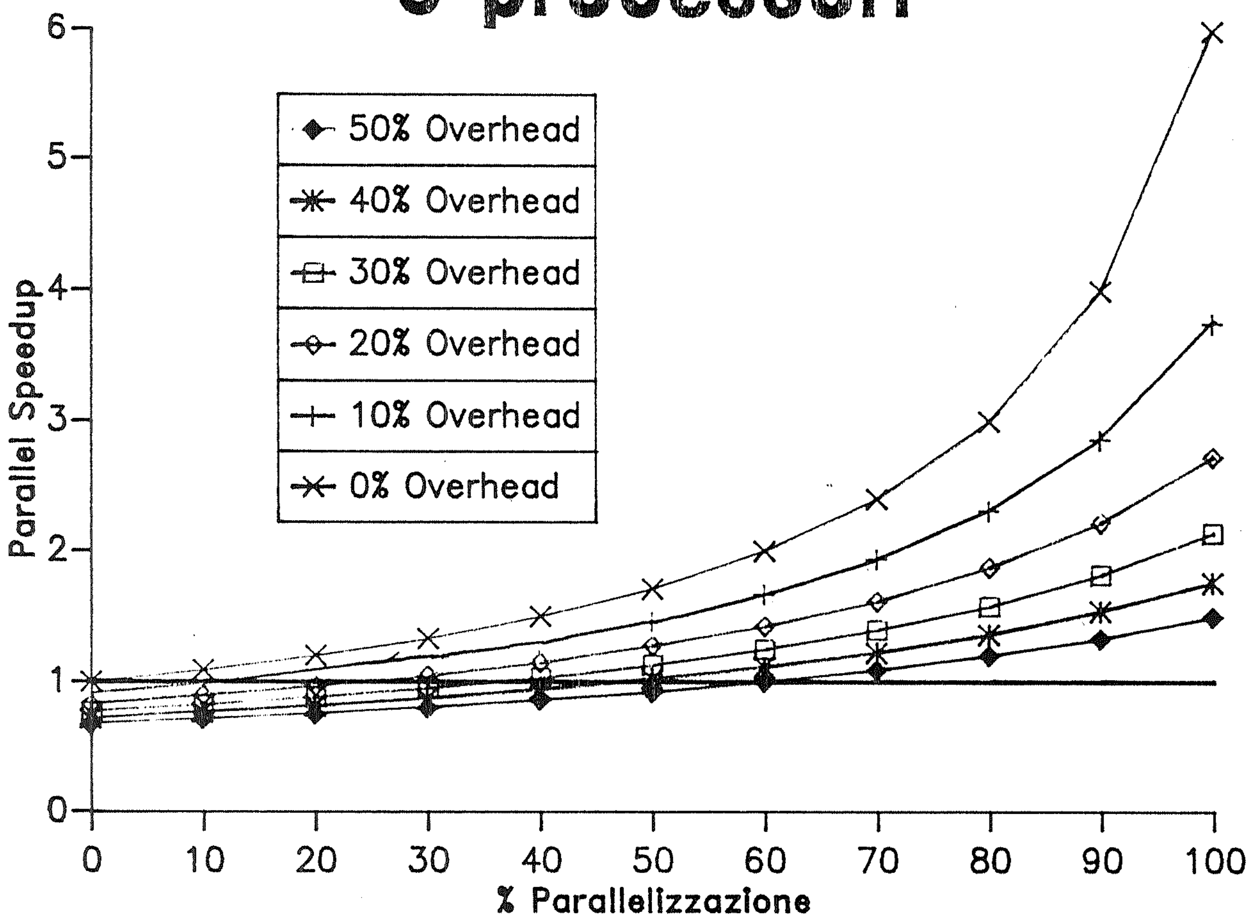
- Il parallelo riduce il tempo di permanenza in macchina del programma a spese di processori che potrebbero essere usati da altri programmi.
  Il tempo di CPU aumenta.

Vettorizzare sempre

Controllare che la parallelizzazione non inibisca o renda inefficiente la vettorizzazione.

Nella elaborazione parallela di solito i vettori si dividono in tante parti quanti sono i processori disponibili.

Fare attenzione che la lunghezza dei vettori rimanga sufficiente per la vettorizzazione.



Parallel Speedup
6 processori
50% Overhead
40% Overhead
30% Overhead
20% Overhead
10% Overhead
0% Overhead
Parallel Speedup
0
1
2
3
4
5
6
0
10
20
30
40
50
60
70
80
90
100
% Parallelizzazione

PARTE II

PROGRAMMAZIONE

DEGLI

ELABORATORI PARALLELI

A

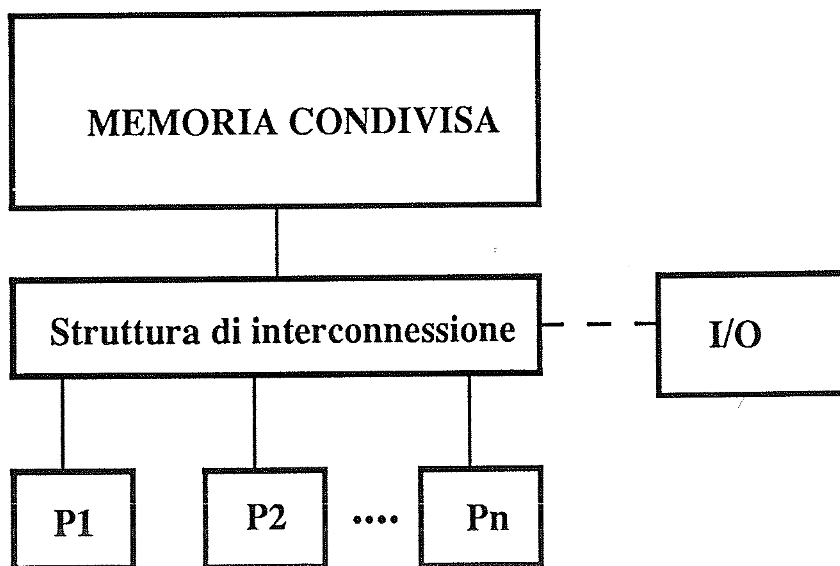
MEMORIA SHARED

---

a cura di: Ranieri Baraglia
Reparto "Calcolo Parallelo"
CNUCE - Istituto del Consiglio Nazionale delle Ricerche

PARALLEL PROCESSING - introduzione

Modello di architettura a memoria globale o condivisa:



I task in esecuzione parallela comunicano accedendo a variabili comuni in memoria condivisa.

Classificazione secondo Flynn:

MIMD (multiple istructions multiple data)

PARALLEL PROCESSING - strumenti

Non esistono strumenti per la totale parallelizzazione automatica di un programma.

Esistono strumenti per la parallelizzazione automatica dei soli DO-loop.

Sono, generalmente compilatori ottenuti con estensioni alle corrispondenti versioni sequenziali (es. IBM PF e CRAY CFT77)

Permettono due tipi di parallelismo:

- ***implicito***

  individuato automaticamente dal compilatore

- ***esplicito***

  richiesto con primitive di parallelizzazione.

PARALLELISMO IMPLICITO - inibitori

Inibitori alla parallelizzazione automatica di DO-loop:

- salti fuori da loop (GO TO)
- operazioni di I/O
- chiamate a routine (CALL, FUNCTION)
- dipendenze tra dati
- variabili induttive

PARALLELISMO IMPLICITO - dipendenze

Esiste una dipendenza sui dati quando l'ordine di esecuzione delle operazioni, istruzioni o iterazioni di un loop determina il risultato dell'elaborazione.

Dipendenze sono accettabili all'interno di un task ma non tra task paralleli.

I task paralleli debbono essere indipendenti.

Le dipendenze e le variabili induttive non costituiscono sempre un inibitore alla parallelizzazione.

Dipendenze tra dati:

- vera
- antidipendenza
- di output.

PARALLELISMO IMPLICITO - dipendenze

Dipendenza vera

S1 accede una locazione di memoria in scrittura e S2 accede la stessa locazione in lettura.

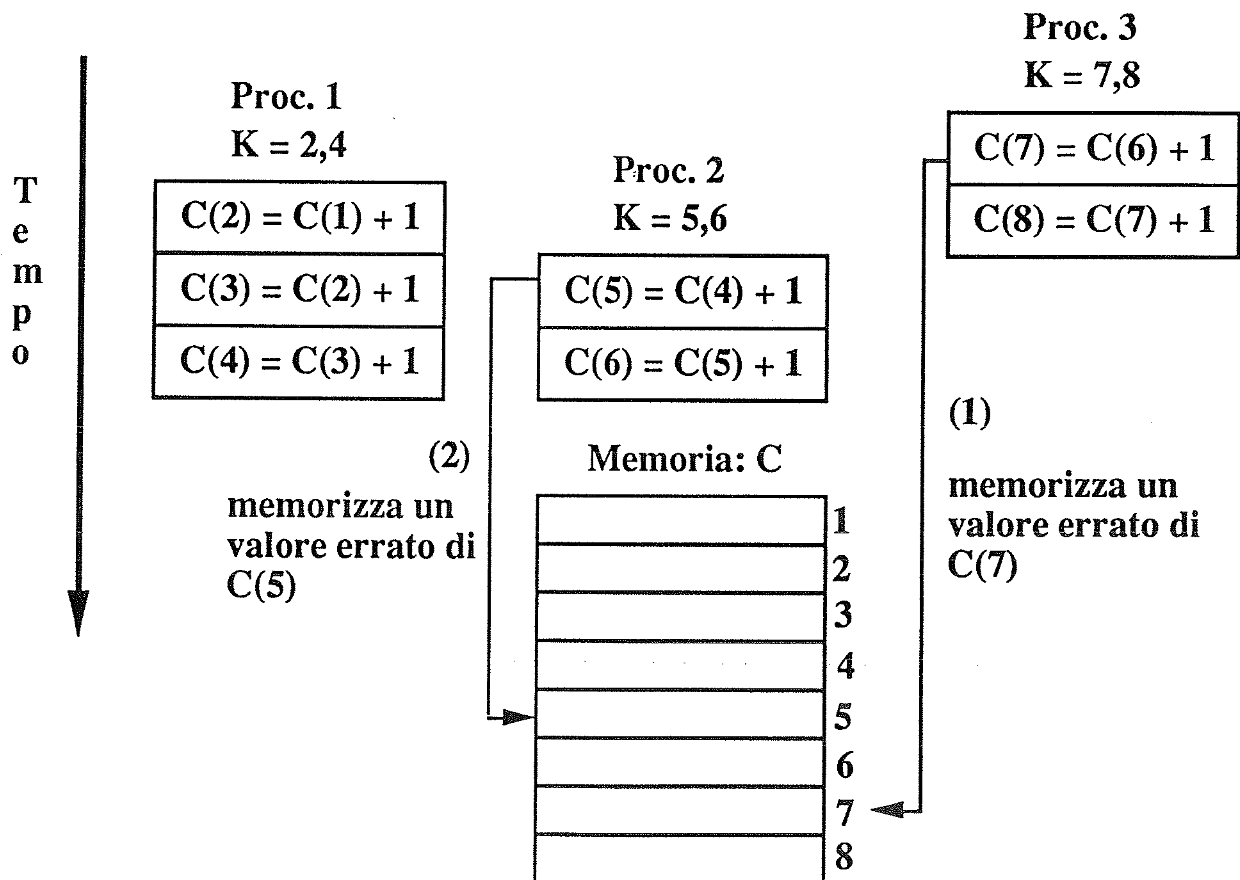
```
              DO 40 K = 1,N
S1            C(K) = B(K)*A(K)
S2            D(K) = C(K)+1
           40 CONTINUE
```

Ogni iterazione è indipendente dalle altre, la dipendenza non inibisce la parallelizzazione del DO-loop.

PARALLELISMO IMPLICITO - dipendenze

Dipendenza vera

```
              DO 40 K = 2,N
S1,2          C(K) = C(K-1)+1
           40 CONTINUE
```



C(K) alla k-esima iterazione può essere calcolato solo al termine dell'iterazione precedente.

Il DO-loop non é parallelizzabile

PARALLELISMO IMPLICITO - dipendenze

Antidipendenza

S1 e S2 accedono la stessa locazione di memoria, rispettivamente, in lettura ed in scrittura.

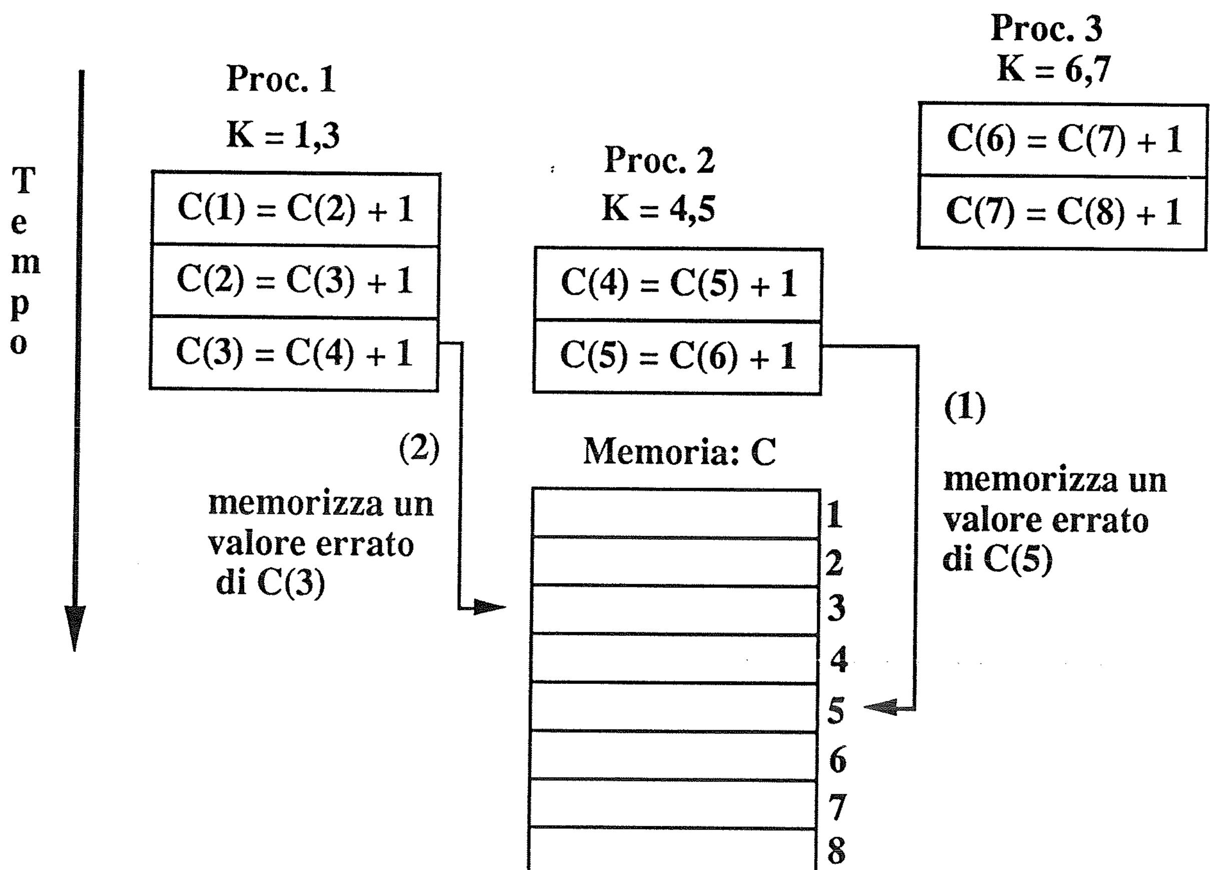
```
              DO 40 K = 1,N
S1            C(K) = B(K)*A(K)
S2            B(K) = A(K)+1
           40 CONTINUE
```

Ogni iterazione è indipendente dalle altre, la dipendenza non inibisce la parallelizzazione del DO-loop.

PARALLELISMO IMPLICITO - dipendenze

Antidipendenza

```
             DO 40 K = 1,N-1
S1,2         C(K) = C(K+1)+1
          40 CONTINUE
```



C(K+1) può essere calcolato solo al termine della k-esima iterazione.

Il DO-loop non é parallelizzabile

PARALLELISMO IMPLICITO - dipendenze

Dipendenza di output

S1 e S2 eseguono due accessi in scrittura alla stessa locazione di memoria.

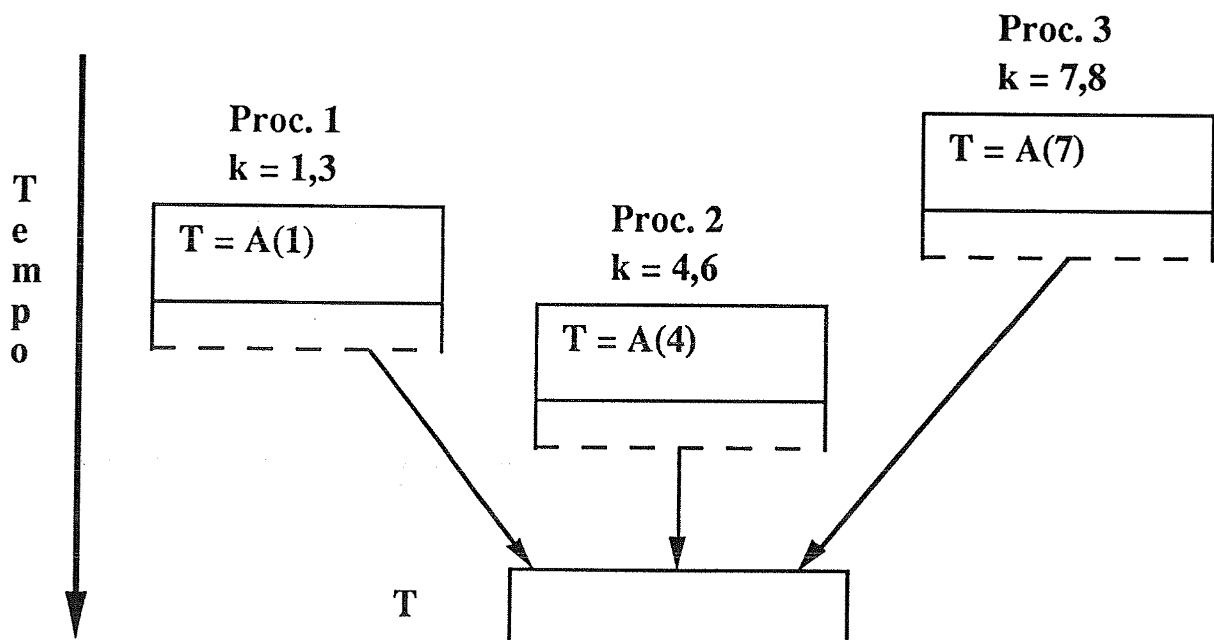
```
              DO 40 K = 1,N
S1            C(K) = A(K)+1
              . . . . . .
S2            C(K) = B(K)+1
           40 CONTINUE
```

Il DO-loop può essere eseguito in parallelo.

PARALLELISMO IMPLICITO - dipendenze

Dipendenza di output

```
                DO 40 K = 1,N
S1,2            T = A(K)
                A(K) = B(K)
                B(K) = T
             40 CONTINUE
```



Il DO-loop può essere parallelizzato solo se la variabile T viene sostituita con un vettore.

PARALLELISMO IMPLICITO - dipendenze

Variabili induttive

una variabile la cui sequenza di valori forma una progressione aritmetica è detta induttiva.

```
      J = 1
      DO 40 I = 2,N,2
      C(J) = C(I) * 2
      J = J + 2
40    CONTINUE
```

Il DO_loop non è parallelizzabile, a meno di una ristrutturazione del tipo:

```
      DO 40 I = 2,N,2
      C(I-1) = C(I)*2
40    CONTINUE
```

PARALLELISMO IMPLICITO

Un DO-loop viene parallelizzato quando:

- non esistono inibitori
- la stima del tempo di esecuzione è favorevole all'elaborazione parallela.

Il compilatore suddivide l'esecuzione delle iterazioni di un DO_loop:

- allocandole ai processori in blocchi contigui (IBM PF):

CPU1 esegue 1,2,................,N/P
CPU2 esegue N/P +1,..........,2 N/P
.
.

- assegnando un'iterazione allo stesso processore ogni P iterazioni (CRAY CFT77, CFT):

CPU1 esegue 1, P+1, 2P+1, ...
CPU2 esegue 2, P+2, 2P+2, ...
.
.

PARALLELISMO IMPLICITO

Un DO-loop può essere contemporaneamente parallelizzato e vettorizzato.

E' questo il caso in cui si ottengono le migliori prestazioni.

La parallelizzazione automatica viene richiesta con: comandi od opzioni di compilazione

```
PARALLEL(AUTO)      ==> PF IBM
cf77  -ZF  alfa.f   ==> CFT77  CRAY
```

PARALLELISMO IMPLICITO

Il programmatore può intervenire tramite l'uso delle direttive per indicare al compilatore:

- il numero di processori da usarsi per l'elaborazione di un DO_loop o di altre forme parallele

- le parti di codice da parallelizzare

- come fare asserzioni che gli permettono di riconoscere strutture parallele anche in presenza di potenziali inibitori.

IBM PARALLEL FORTRAN (PF)

Ottenuto dalla versione 2.1 del compilatore Fortran VS con:

- ***Estensioni ai servizi del compilatore***

  parallelizzazione automatica (DO-loop)

- ***Estensioni al linguaggio***
  - **iterazioni in parallelo**
  - **istruzioni in parallelo**
  - **subroutines in parallelo**

- ***Estensioni alla libreria***
  - **locks**
  - **events**
  - **trace**

PF - PARALLELISMO ESPLICITO

Iterazioni in parallelo

PARALLEL LOOP

Istruzioni in parallelo

PARALLEL CASES

Subroutines in parallelo

ORIGINATE

SCHEDULE

DISPATCH

WAIT

TERMINATE

PF - PARALLEL LOOP

Iterazioni in parallelo:

```
PARALLEL LOOP lab ind=in,term[,step]
[PRIVATE(var[,var] ....)]
[DOFIRST[LOCK]]
..........
prologo, eseguito un volta per task
..........
DOEVERY
..........
corpo del DO-loop parallelo
..........
[DOFINAL[LOCK]]
..........
epilogo, eseguito un volta per task
..........
CONTINUE
```

***var*:**

variabili artmetiche o logiche, una copia per ogni task implicito, valore iniziale e finale indefinito.

PF - PARALLEL LOOP

- può essere eseguito da più CPU
- ciascuna CPU può eseguire un sottoinsieme di iterazioni per volta
- l'esecuzione sequenziale continua solo dopo che tutte le iterazioni sono state eseguite
- può contenere altri costrutti paralleli (parallel loop, parallel cases e DO_loop, loop nidificati max 25 livelli)
- l'indice del loop parallelo è implicitamente definito privato
- prevede l'uso di funzioni intrinsiche ed operazioni di I/O
- DOFIRST e DOFINAL non possono contenere variabili indice del parallel loop.

PARALLEL LOOPS - Esempio

```
      SUM = 0.D0
      DO 10 I = 1,N
      DO 10 J = 1,N
      SUM = SUM + A(I,J)
   10 CONTINUE
```

```
      SUM=0.D0
      PARALLEL LOOP 10 I=1,N
       PRIVATE(PSUM)
       DOFIRST
        PSUM=0.D0
       DOEVERY
         DO 20 J = 1,N
         PSUM = PSUM + A(I,J)
   20    CONTINUE
       DOFINAL LOCK
        SUM=SUM+PSUM
   10 CONTINUE
```

DOFIRST e DOFINAL sono eseguite una sola volta da ciascuna CPU.

PARALLEL LOOPS - Esempi

```
      PARALLEL  LOOP 10 I=1,N
      PRIVATE(T)
      T = A(I)
      A(I) = B(I)
      B(I) = T
10    CONTINUE
```

```
      PARALLEL  LOOP 10 I=1,N
         ..........
         ..........
        IF (A(I).EQ.V) THEN
           LI = I
           STOP LOOP 10
        ENDIF
         ..........
         ..........
10    CONTINUE
```

STOP LOOP interrompe l'esecuzione del Parallel Loop.

PF - PARALLEL LOOP

Viene eseguito in modo seriale se contiene:

- variabili di tipo carattere
- salti nel DOFIRST
- salti fuori dal DOFIRST
- salti nel DOEVERY
- salti fuori dal DOEVERY
- salti nel DOFINAL
- salti fuori dal DOFINAL
- variabili indice non I*4
- chiamate a subroutine o function di utente

PF - PARALLEL CASES

Sequenze di istruzioni in parallelo:

```
PARALLEL CASES
 [PRIVATE(var[,var] ....)]
 CASE
 . . . . .
 . . . . .
 CASE 1
 . . . . .
 . . . . .
 CASE 2,WAIT FOR CASE 1
 . . . . .
 . . . . .
 CASE[m[,WAIT FOR CASE(n1[,n2]....)]]
 . . . . .
 . . . . .
 END CASES
```

var:

variabili artmetiche o logiche, una copia per ogni case, valore iniziale e finale indefinito.

PF - PARALLEL CASES

- può essere eseguito da più CPU
- ciascuna CPU può eseguire un case per volta
- l'esecuzione sequenziale riprende solo dopo che tutti i cases sono stati eseguiti
- può contenere altri costrutti paralleli (parallel cases, parallel loop e DO_loop)
- le variabili indice di un loop parallelo, contenuto in un case, sono private
- l'istruzione WAITING FOR CASE sincronizza l'esecuzione di più case, all'interno dello stesso Parallel Cases
- un case può usare funzioni intrinsiche ed operazioni di I/O
- l'indipendenza tra i case deve essere controllata dal programmatore.

PARALLEL CASES - Esempio

```
        PARALLEL CASES
         CASE 1
           DO 10 I = 1, N
            C(I) = SQRT(ABS(A(I)))
10         CONTINUE
         CASE 2
           DO 20 I = 1,N
            D(I) = SQRT(ABS(B(I)))
20         CONTINUE
         CASE 3, WAITING FOR CASE 1
           DO 30 I = 1,N
            E(I) = E(I) + C(I)
30         CONTINUE
         CASE 4, WAITING FOR CASE 2
           DO 40 I = 1,N
            F(I) = F(I) + D(I)
40         CONTINUE
        END CASES
```

PF - PARALLEL CASES

Viene eseguito in modo seriale se contiene:

- variabili di tipo carattere
- salti all'interno di un case
- salti fuori da un case
- variabili indice non I*4
- chiamate a subroutine o function di utente.

PF - PARALLELISMO ESPLICITO

Subroutines in parallelo:

ORIGINATE ANY TASK *taskid*
TERMINATE TASK *taskid*

ORIGINATE

- crea un task parallelo
- ritorna al programma l'identificatore del task

TERMINATE

cancella un task e rilascia la memoria ad esso allocata

ORIGINATE/TERMINATE-Esempio

```
        INTEGER*4  TASKID(4)
        ...............
        ...............
        DO 10 I = 1,4
          ORIGINATE  ANY  TASK  TASKID(I)
   10   CONTINUE
        ...............
        ...............
        ORIGINATE  ANY  TASK  ITASK
        ...............
        ...............
        ...............
        DO 10 I = 1,4
          TERMINATE  TASK  TASKID(I)
   10   CONTINUE
        ...............
        ...............
        TERMINATE  TASK  ITASK
        ...............
```

PF - PARALLELISMO ESPLICITO

Subroutines in parallelo:

```
SCHEDULE TASK taskid,
 [TAGGING(tag1[,tag2]  .....),]
 [SHARING(common1[,common2]  ....),]
 [COPYING(common1[,common2]  ....),]
 [COPYINGI(common1[,common2]  ....),]
 [COPYINGO(common1[,common2]  ...),]
 CALLING  nomesub[arg1[,arg2]  ...)]

SCHEDULE ANY TASK taskid, ....
```

SCHEDULE

assegna una subroutine ad un task per l'esecuzione parallela

SCHEDULE - Esempio

```
      .........
      INTEGER*4  TASKID(4)
      .........
      .........
      DO 10 I = 1,4
        ORIGINATE  ANY  TASK  TASKID(I)
10    CONTINUE
      .........
      .........
      DO 20 I = 1,4
        SCHEDULE  TASK  TASKID(I),CALLING  SUB3
20    CONTINUE
      .........
      .........
      DO 30 I = 1,4
        TERMINATE  TASK  TASKID(I)
30    CONTINUE
      STOP
      END
      SUBROUTINE  SUB3
      .........
      .........
      RETURN
      END
```

PF - PARALLELISMO ESPLICITO

Subroutines in parallelo:

```
DISPATCH TASK taskid,
  [TAGGING(tag1[,tag2] .....),]
  [SHARING(common1[,common2] ....),]
  [COPYING(common1[,common2] ....),]
  [COPYINGI(common1[,common2] ....),]
  [COPYINGO(common1[,common2] ...),]
  CALLING nomesub[arg1[,arg2] ...)]

DISPATCH ANY TASK taskid, ........
```

DISPATCH

assegna una subroutine ad un task per l'esecuzione parallela

Dispatch può assegnare del nuovo lavoro ad un task senza attendere il completamento del lavoro ad esso assegnato in precedenza.

Schedule può riassegnare lavoro ad un task, solo dopo che il lavoro ad esso assegnato è completato.

PF - PARALLELISMO ESPLICITO

Subroutines in parallelo:

```
WAIT FOR TASK taskid
      [,TAGGING(,tag1[,tag2]  .....)]

WAIT FOR ANY TASK taskid
      [,TAGGING(,tag1[,tag2]  .....)]

WAIT FOR ALL TASK
```

WAIT

- aspetta la fine di uno o di tutti i task paralleli
- definisce un punto di sincronizzazione

SINCRONIZZAZIONE - Esempi

```
          .........
          INTEGER*4  TASKID(4),TASK2(8)
          .........
          .........
          DO 10 I = 1,4
            ORIGINATE ANY TASK TASKID(I)
   10     CONTINUE
          .........
          SCHEDULE TASK TASKID(1),CALLING SUB1
          .........
          WAIT FOR TASK TASKID(1)
          SCHEDULE TASK TASKID(1),CALLING SUB2
          .........
          DO 20 I = 1,4
            SCHEDULE TASK TASKID(I),CALLING SUB3
   20     CONTINUE
          .........
          WAIT FOR ANY TASK J
          .........
          DISPATCH TASK J,CALLING SUB4
          WAIT FOR ALL TASKS
          .........
          DO 30 I = 1,8
            DISPATCH ANY TASK TASK2(I),
            CALLING SUB5
   30     CONTINUE
          WAIT FOR ALL TASKS
          .........
          DO 40 I = 1,4
            TERMINATE TASK TASKID(I)
   40     CONTINUE
          STOP
          END
```

PF - PARALLELISMO ESPLICITO

Clausola *TAGGING*:

Permette di associare un identificatore alla subroutine assegnata ad un task parallelo.

```
SCHEDULE TASK taskid,TAGGING(tag1,..),CALLING...
   ............
   ............
WAIT FOR TASK taskid,TAGGING(,tagx,...)
```

In tagx viene memorizzato il contenuto di tag1 al completamento dell'esecuzione della subroutine chiamata.

TAGGING - Esempio

```
      .........
      INTEGER*4 IT(4)
      .........
      .........
      DO 10 I = 1,4
      ORIGINATE ANY TASK IT(I)
10    CONTINUE
      DO 10 K = 1,NTIMES
      DO 20 J = 1,N
      SCHEDULE TASK IT(1),TAGGING(K,J),CALLING SUB
      SCHEDULE TASK IT(2),TAGGING(K,J),CALLING SUB
      SCHEDULE TASK IT(3),TAGGING(K,J),CALLING SUB
      SCHEDULE TASK IT(4),TAGGING(K,J),CALLING SUB
      WAIT FOR TASK IT(1),TAGGING(I1,I2)
      IF(I1.EQ.3.AND.I2.EQ.2) THEN
      .........
      .........
      ENDIF
      WAIT FOR ALL TASKS
      .........
      .........
20    CONTINUE
      .........
      .........
10    CONTINUE
```

Quando IT(1) ha completato l'esecuzione di SUB può essere eseguito un certo processo se K=3 e J=2.

PF - PARALLELISMO ESPLICITO

Data exchange :

SHARING (nome,.....)

i common sono messi in comune

COPYING (nome,.....)

i common del programma chiamante sono copiati nei corrispondenti common della subroutine parallela.

COPYINGI (nome,.....)

i common del programma chiamante sono copiati nei corrispondenti common della subroutine parallela.

COPYINGO (nome,.....)

i common della subroutine parallela sono copiati nei corrispondenti common del programma chiamante, alla esecuzione della corrispondente WAIT.

SHARING - Esempio

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      .........
      SCHEDULE TASK T1,SHARING(C1),CALLING SUB1(X)
      SCHEDULE TASK T2,SHARING(C1),CALLING SUB2(Y)
          .........
      WAIT FOR ALL TASKS
C     Z(1) contiene 2
C     Z(2) contiene 3
          .........
      SUBROUTINE SUB1(X)
      COMMON /C1/Z(1000)
C     Z(1) contiene 1 quando SUB1 è chiamata
      Z(1) = Z(1) + 1
      .........
      .........
      SUBROUTINE SUB2(X)
      COMMON /C1/Z(1000)
C     Z(2) contiene 2 quando SUB2 è chiamata
      Z(2) = Z(2) + 1
      .........
```

NO SHARING - Esempio

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      .........
      SCHEDULE TASK T1,CALLING SUB1(X)
      SCHEDULE TASK T2,CALLING SUB2(Y)
        .........
      WAIT FOR ALL TASKS
C     Z(1)  contiene 1
C     Z(2)  contiene 2
        .........
      SUBROUTINE SUB1(X)
      COMMON /C1/Z(1000)
C     Z(1)  contenuto non definito quando SUB1 è chiamata
      Z(1) = Z(1) + 1
      .........
      .........
      SUBROUTINE SUB2(X)
      COMMON /C1/Z(1000)
C     Z(2)  contenuto non definito quando SUB2 è chiamata
      Z(2) = Z(2) + 1
      .........
```

COPYING - Esempio

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      .........
      SCHEDULE TASK T1,COPYING(C1),CALLING SUB1(X)
      SCHEDULE TASK T2,COPYING(C1),CALLING SUB2(Y)
        .........
      WAIT FOR ALL TASKS
C     Z(1)  contiene 1 o 2
C     Z(2)  contiene 3 o 2
        .........
      SUBROUTINE SUB1(X)
      COMMON /C1/Z(1000)
C     Z(1)  contiene 1 quando SUB1 è chiamata
      Z(1) = Z(1) + 1
      .........
      .........
      SUBROUTINE SUB2(X)
      COMMON /C1/Z(1000)
C     Z(2)  contiene 2 quando SUB2 è chiamata
      Z(2) = Z(2) + 1
      .........
```

Il contenuto di Z dipende da quale subroutine termina per prima.

COPYINGI - Esempio

```
      .........
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
      Y(1) = 101
      Y(2) = 102
      .........
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,COPYINGI(C1),SHARING(C2),
      CALLING SUB1
      SCHEDULE TASK T2,SHARING(C1),CALLING SUB2
        .........
      WAIT FOR ALL TASKS
C     Y(1) contiene 2
C     Z(1) contiene 0
C     Z(2) contiene 3
        .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
C     Z(1) contiene 1
      Y(1) = 2 * Z(1)
      .........
      .........
      SUBROUTINE SUB2
      COMMON /C1/Z(1000)
C     Z(1) contiene 1
C     Z(2) contiene 2
      Z(1) = 0
      Z(2) = 3
      .........
```

COPYINGI - Esempio

```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,COPYINGI(C1),CALLING SUB1
        .........
      WAIT FOR ALL TASKS
C     Z(1) contiene 1
C     Z(2) contiene 2
        .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
C     Z(1) contiene 1
C     Z(2) contiene 2
      Z(1) = 0
      Z(2) = 3
      .........
```

Le modifiche fatte da SUB1 non sono ritornate al programma principale.

COPYINGO - Esempio

```
      .........
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
      Y(1) = 101
      Y(2) = 102
      .........
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,SHARING(C1,C2),CALLING SUB1
      SCHEDULE TASK T2,COPYINGO(C1),CALLING SUB2
          .........
      WAIT FOR ALL TASKS
C     Y(1)  contiene 2
C     Z(1)  contiene 0
C     Z(2)  contiene 3
          .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
      COMMON /C2/Y(1000)
C     Z(1)  contiene 1
      Y(1) = 2 * Z(1)
      .........
      .........
      SUBROUTINE SUB2
      COMMON /C1/Z(1000)
C     Z     ha un volore indeterminato quando SUB2 è chiamata
      Z(1) = 0
      Z(2) = 3
      .........
```

COPYINGO - Esempio

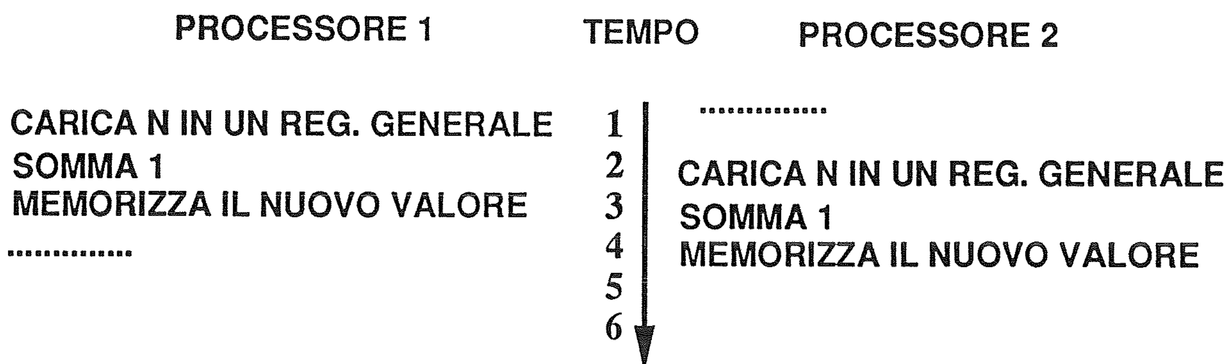
```
      .........
      COMMON /C1/Z(1000)
      Z(1) = 1
      Z(2) = 2
      .........
      SCHEDULE TASK T1,COPYINGO(C1),CALLING SUB1
         .........
      WAIT FOR ALL TASKS
C     Z(1)  contiene 0
C     Z(2)  ha un valore indeterminato
         .........
      SUBROUTINE SUB1
      COMMON /C1/Z(1000)
C     Z       ha un valore indeterminato quando SUB1 è chiamata
      Z(1) = 0
      .........
```

PF SINCRONIZZAZIONE - Lock

Locking

stabilisce la modalità di accesso ad aree usate simultaneamente da più task paralleli.

Operazione N = N + 1 eseguita da due task concorrenti:



PF SINCRONIZZAZIONE - Lock

PLORIG(*lockid*)

- crea un *parallel lock*
- memorizza in *lockid* l'identificatore del lock
- *lockid* variabile o elemento di un array (I*4).

PLTERM(*lockid*)

- cancella un *parallel lock*

PLLOCK(*lockid*,*var1*, ...)

- ottiene il *parallel lock*
- *lockid* contiene l'identificatore del lock richiesto
- *var1*, ... sono variabili, array o elementi di array per i quali si richiede l'accesso sequenziale.

PLFREE(*lockid*,*var1*, ...)

rilascia un *parallel lock*

LOCK - Esempio

```
.........
INTEGER*4 LOCK,COUNT
COMMON /C1/LOCK,COUNT
Z(1) = 1
Z(2) = 2
.........
.........
CALL PLORIG(LOCK)
COUNT = 0
ORIGINATE ANY TASK IT1
ORIGINATE ANY TASK IT2
.........
.........
SCHEDULE TASK IT1,SHARING(C1),CALLING SUB1
SCHEDULE TASK IT2,SHARING(C1),CALLING SUB2
    .........
WAIT FOR ALL TASKS
CALL PLTERM(LOCK)
.........
.........
SUBROUTINE SUB1
INTEGER*4 LOCK,COUNT
COMMON /C1/LOCK,COUNT
CALL PLLOCK(LOCK,COUNT)
COUNT = COUNT + 1
CALL PLFREE (LOCK,COUNT)
.........
.........
SUBROUTINE SUB2
INTEGER*4 LOCK,COUNT
COMMON /C1/LOCK,COUNT
CALL PLLOCK(LOCK)
COUNT = COUNT + 1
CALL PLFREE (LOCK)
.........
```

PF SINCRONIZZAZIONE - Eventi

Un task parallelo in esecuzione può segnalare il completamento di una certa sua parte e dopo continuare l'esecuzione.

Un task parallelo in stato di wait in attesa del completamento di un certo evento, può riprendere l'esecuzione quando riceve la segnalazione che l'evento si è verificato.

In generale queste due azioni sono descritte con:

- **POST:** informa che un evento si è verificato
- **WAIT:** pone il task in attesa per un evento.

PF SINCRONIZZAZIONE - Eventi

PEORIG(*eventid*)

- crea un *parallel event*
- *eventid* memorizza l'identificatore dell'evento creato
- imposta i contatori *postcount*, *waitcount* e *eventtype* sono posti a 1.

PETERM(*eventid*)

cancella un *parallel event*

PEPOST(*eventid*)

esegue un *post* di un *parallel event*

PEWAIT(*eventid*)

il *task* chiamante viene messo in attesa di un *parallel event*

PF SINCRONIZZAZIONE - Eventi

PEINIT(*eventid,postcount,waitcount, eventtype*)

inizializza un *parallel event*

- ***postcount***

  indica il numero di *post* richieste per soddisfare l'evento

- ***waitcount***

  indica il numero di *wait* richieste per soddisfare un evento

- ***eventtype***

  può essere 0 o 1

  0 un task può eseguire più volte una post o una wait

  1 un task può eseguire una sola volta una post o una wait

Es. CALL PEINIT(IVENT1,5,1,1)

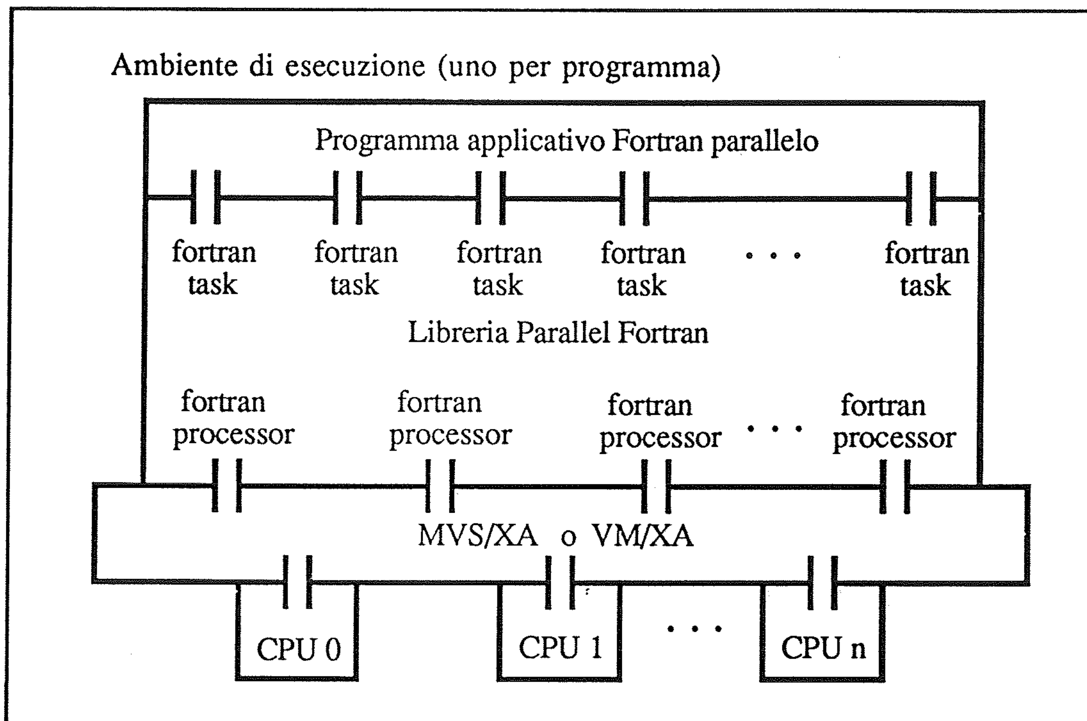
EVENT - Esempio

```
PROGRAM EVENTI
INTEGER*4 E1,E2
COMMON /C1/ E1,E2
COMMON /C2/ A(100)
  .........
NTASK = MIN(NPROCS,N1)
CALL PEORIG(E1)
CALL PEORIG(E2)
ORIGINATE ANY TASK IT1
ORIGINATE ANY TASK IT2
  .........
DISPATCH TASK IT1,SHARING(C1,C2),CALLING S1
DISPATCH TASK IT2,SHARING(C1,C2),CALLING S2
  .........
WAIT FOR ALL TASKS
CALL PETERM(E1)
CALL PETERM(E2)
  .........
END


SUBROUTINE S1                           SUBROUTINE S2
INTEGER*4 E1,E2                         INTEGER*4 E1,E2
COMMON /C1/ E1,E2                       COMMON /C1/ E1,E2
COMMON /C2/ A(100)                      COMMON /C2/ A(100)
  .........                               .........
   Calcola  A                    ------> CALL PEWAIT(E1)
  .........                     |         .........
CALL PEPOST(E1)        -------            .........
  .........                             Usa A
CALL PEWAIT(E2)        <------            .........
  .........                     |         .........
   Modifica A                    ----- CALL PEPOST(E2)
  .........                               .........
```

EVENT - Esempio

```
      PROGRAM MAIN
      PARAMETER(N=6)
      DIMENSION IT(N)
      COMMON /AB/ ..........
         .........
         .........
      DO 10 I = 1,N
   10 ORIGINATE ANY TASK IT(I)
         .........
C  CREA E INIZIALIZZA UN EVENTO CHE PERMETTE
C  DI SINCRONIZZARE TUTTI I TASK AD UN CERTO
C  PUNTO DELLA LORO ESECUZIONE
      CALL PEORIG(IEVNT)
      CALL PEINIT(IEVNT,N,N,1)
         .........
         .........
      DO 10 I = 1,N
   10 DISPATCH TASK IT(I),SHARING(AB),
      CALLING SUB(IEVNT,A,B)
      WAIT FOR ALL TASKS
         .........
      CALL PETERM(IEVNT)
      DO 10 I = 1,N
   10 TERMINATE TAS IT(I)
      END

      SUBROUTINE SUB(IEVNT,A,B)
      COMMON /AB/ ..........
         .........
         .........
C  LA POST SEGNALA CHE IL TASK HA RAGGIUNTO UN
C  CERTO PUNTO; LA WAIT LO PONE IN ATTESA FINO
C  A CHE TUTTI GLI ALTRI TASK ABBIANO RAGGUNTO
C  LO STESSO PUNTO;
      CALL PEPOST(IEVNT)
      CALL PEWAIT(IEVNT)
         .........
      RETURN
      END
```

PF - Ambiente di esecuzione



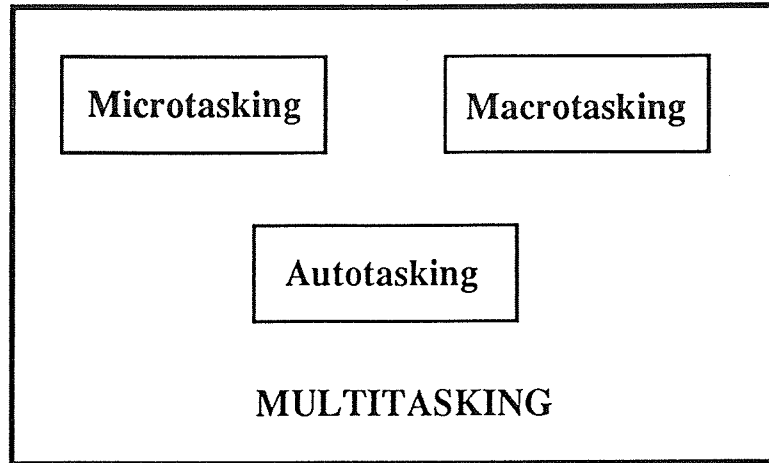
Programma

- crea i task Fortran
- i task sono accodati per l'esecuzione sui processori Fortran

Processori Fortran

- creati a run time
- implementati come task MVS o processori virtuali (VM)
- allocati ai processori reali dal sistema operativo

CRAY - Multitasking



Macrotasking

- realizza il parallelismo a livello di subroutine
- controllato dal programmatore
- le subroutine costituiscono dei task
- un task è allocato dal sistema operativo su una CPU reale.

CRAY - Multitasking

Microtasking

- realizza il parallelismo sia a livello di subroutine che di DO-loop
- richiesto tramite direttive di compilazione (le parti di codice parallelizzabile sono detti processi)
- generato utomaticamente dal compilatore
- introduce un minore overhead rispetto al macrotasking.

Autotasking

parallelizzazione automatica dei DO-loop.

CRAY - Multitasking

L'autotasking, il microtasking e il macrotasking possono coesistere in uno stesso programma, ma non nella stessa subroutine o function.

L'utilizzo di direttive di microtasking o funzioni di macrotasking inibiscono a livello di subroutine l'utilizzo dell'autotasking.

CRAY - Autotasking (parallelismo implicito)

E' realizzato in tre fasi distinte:

- analisi delle dipendenze
- traduzione
- generazione del codice

Analisi delle dipendenze:

- individuazione delle strutture parallele
- esecuzione della cost-analysis
- inserimento delle direttive che esprimono il parallelismo.

CRAY - Autotasking

Traduzione

sostituzione delle direttive con chiamate alle routine di sistema che realizzano e controllano l'elaborazione parallela.

Generazione del codice

traduzione del codice Fortran prodotto nella fase precedente in codice eseguibile.

CRAY - Autotasking

Intervento del programmatore:

- sul codice sorgente

- output della fase di analisi delle dipendenze

Sul codice sorgente attraverso due categorie di direttive:

- CFPP$

  specificano informazioni utili al compilatore per l'individuazione di DO-loop parallelizzabili.

- CMIC$

  segnalano dove esiste il parallelismo, anche in presenza di inibitori.

Sull'output della prima fase, inserendo direttive(CMIC$) per le parti di codice parallelizzabili ma non riconosciute dal compilatore.

CRAY - Microtaking

Direttive per il *Microtasking*

CMIC$ GETCPUS *n*

- specifica il numero massimo di processori
- va nel main prima di iniziare il microtasking

CMIC$ RELCPUS *n*

- rilascia i processori
- va nel main dopo routine con direttive di microtasking

CRAY - Microtaking

CMIC$ MICRO

- specifica che la subroutine che immediatamente precede deve essere eseguita in microtasking

- lo statement RETURN segnala la fine del lavoro parallelo

- una Function non può usare il microtasking, deve essere riscritta come subroutine.

CRAY - Microtaking

CMIC$ PROCESS

- segnala l'inizio di una sequenza di istruzioni per le quali si richiede l'elaborazione parallela

- indica che il codice che segue costituisce un singolo processo.

CMIC$ ALSO PROCESS

- segnala l'inizio di un processo diverso dal precedente

- indica la fine del processo precedente.

CMIC$ END PROCESS

- segnala la fine di un processo

- Process-End Process sono usate per assicurare l'esecuzione di porzioni di codice su un singolo processore.

CRAY - Microtaking

CMIC$ DO GLOBAL

- indica che il DO-loop che immediatamente precede deve essere eseguito in parallelo

- la label del DO-loop segna la fine del processo.

CMIC$ STOP ALL PROCESS

- interrompe l'esecuzione delle strutture *Process* e *DO Global*

```
CMIC$ DO GLOBAL
      DO 1 I = 1,N
          ......
          IF (condizione) THEN
CMIC$ STOP ALL PROCESS
         GO TO 2
         ENDIF
1     CONTINUE
          .......
2     CONTINUE
```

CRAY - Microtaking

CMIC$ GUARD *n*

- segnala l'inizio di una sezione critica
- solo *guards* con lo stesso numero non possono essere eseguite in parallelo

CMIC$ END GUARD *n*

- segnala la fine di una sezione critica

CRAY - Esempio di Microtaking

```
      CMIC$ PROCESS
      DO 100 .....
      .......
      .......
100   CONTINUE
      CMIC$ ALSO PROCESS
      DO 200 .....
      .......
      .......
200   CONTINUE
      CMIC$ ALSO PROCESS
      A = B + 1
      .......
      .......
      CDIR$ END PROCESS
      .......
      .......
      RETURN
      END
```

CRAY - Macrotasking

Consiste di un insieme di routine per la creazione e la sincronizzazione dei task.

Controllo dei *Task*

CALL TSKSTART(taskid,subname,args)

CALL TSKWAIT(taskid)

Controllo degli *Eventi*

CALL EVPOST(eventid)

CALL EVWAIT(eventid)

CALL EVCLEAR(eventid)

Controllo dei *Lock*

CALL LOCKON(lockid)

CALL LOCKOFF(eventid)

CRAY - Esempio di Macrotasking

Codice originale

```
PROGRAM MAIN
CALL ALFA
CALL BETA
STOP
END
```

Le subroutine Alfa e Beta eseguono operazioni indipendenti su dati differenti

Versione Macrotasking

CPU 0		CPU 1
PROGRAM MAIN		
CALL TSKSTART (IDT,ALFA)	===>	ESECUZIONE ALFA
CALL BETA		
CALL TSKWAIT (IDT)	===>	RETURN
STOP		
END		

CRAY - Macrotasking

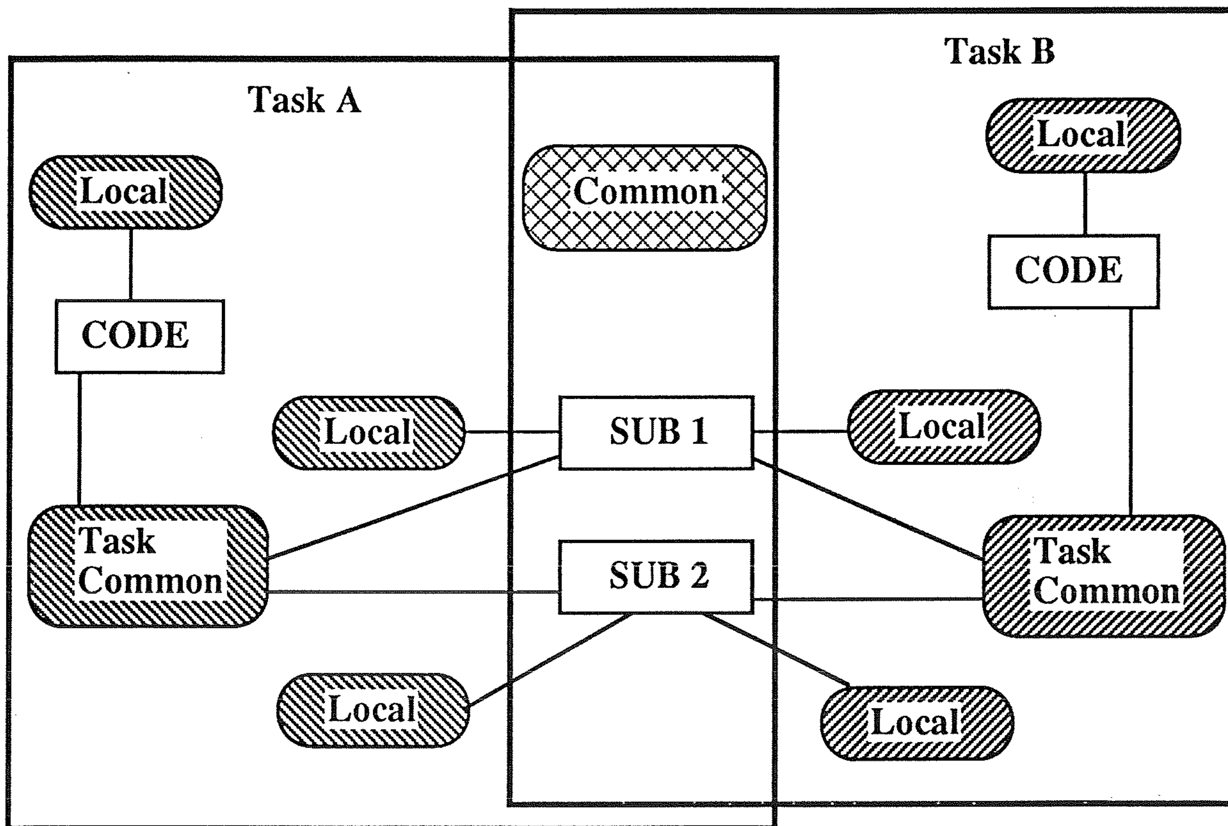
Data exchange:

é utilizzato lo statement TASK COMMON:

- costituisce una estensione per i compilatori CFT e CFT77
- dichiara dati comuni a tutte le routine di un task
- permette la suddivisione dei dati fra le routine di un task
- i dati sono privati per le routine in quel task
- permette a routine chiamate da task diversi di lavorare su dati differenti in ciascun task.

CRAY - PARALLELISMO ESPLICITO

Data exchange:



CRAY - Esempio di Macrotasking

Codice originale:

```
      PROGRAM MAIN
      COMMON /X/ A(100),B(100),C(100),N
      N = 100
      DO 20 J = 1,2
        DO 10 I = 1,N
        A(I) = I+J
        B(I) = I*J
10    CONTINUE
      CALL SUB(J)
20    CONTINUE
      PRINT *,(C(I),I=1,2)
      STOP
      END


      SUBROUTINE SUB(J)
      COMMON /X/ A(100),B(100),C(100),N
      DO 30 K = 1,N
        C(J) = C(J) + A(K)*B(K)
30    CONTINUE
      RETURN
      END
```

CRAY - Esempio di Macrotasking

Versione Macrotasking

```
      INTERGER IDT
      COMMON /X/ C(100),N
      TASK COMMON /XX/ A(100),B(100)
      EXTERNAL SPROD
      N = 100
      L = 2
      CALL TSKSTART (IDT,SPROD,L)
      M = 1
      CALL SPROD (M)
      CALL TSKWAIT (IDT)
      PRINT *,(C(I),I = 1,2)
      STOP
      END
      SUBROUTINE SPROD(J)
      COMMON /X/ C(100),N
      TASK COMMON /XX/ A(100),B(100)
      DO 10 I = 1,N
        A(I) = I+J
        B(I) = I*J
 10   CONTINUE
      CALL SUB(J)
      RETURN
      END
      SUBROUTINE SUB(J)
      COMMON /X/ C(100),N
      TASK COMMON /XX/ A(100),B(100)
      DO 10 K = 1,N
        C(J) = C(J) + A(K)*B(K)
 30   CONTINUE
      RETURN
      END
```

Parte III

Principi di Programmazione per Elaboratori Paralleli a Memoria Distribuita

"Supercomputing Tools for Science and Engineering"

Pisa, 4-7 Dicembre 1989

A cura di Raffaele Perego
Reparto "Calcolo Parallelo"
CNUCE - Istituto del Consiglio Nazionale delle Ricerche

PROGRAM PARALLELIZATION

Automatica :

- Compilatori che estraggono il parallelismo implicito
  FORTRAN, C, LISP, PROLOG, ecc.

- Interpreti paralleli per linguaggi logici/funzionali (macchine a riduzione, macchine data-flow)

Esplicita :

- Linguaggi tradizionali estesi con costrutti paralleli e/o direttive di parallelizzazione

- Linguaggi concorrenti
  (CSP, OCCAM, ADA, ecc.)

Distributed-Memory vs. Shared-Memory

Architetture parallele a memoria condivisa:
(Cray X-MP, IBM 3090, Alliant FX/8, Sequent Balance, ecc.)

- Alto costo di progettazione e realizzazione
- Scarso parallelismo (<< 100)
- Pessima scalabilità
- Ambiente di programmazione sufficientemente ricco
- Basso costo di migrazione del codice

Linee di ricerca e sviluppo: Diminuzione Ciclo Clock, Aumento Banda di Memoria, Compilatori Parallelizzanti, ecc.

Architetture parallele a memoria distribuita:
(NCUBE, Intel IPSC/2, Meiko, CM-2, ecc.)

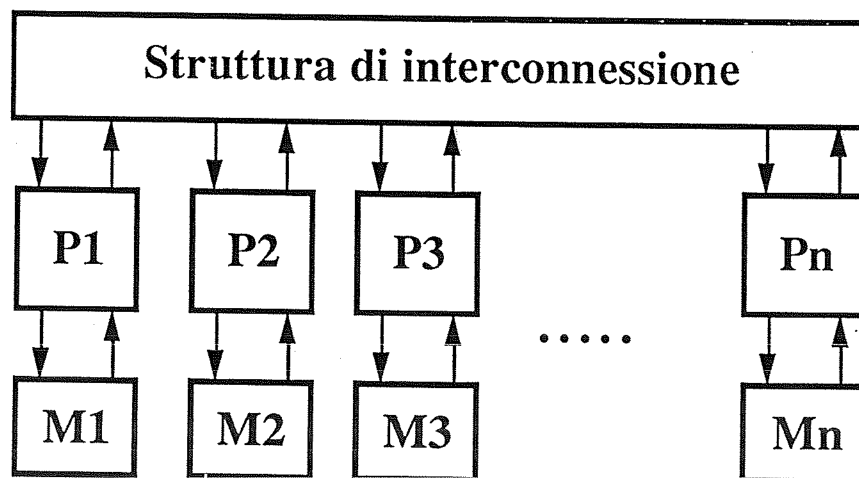
- Basso costo di progettazione e realizzazione
- Elevato parallelismo (>> 100)
- Ottima scalabilità
- Povertà dell'ambiente di programmazione
- Alto costo di migrazione del codice

Linee di ricerca e sviluppo: D.O.S., Tools software, Linguaggi, Reti di comunicazione, Routing systems, ecc.

Architetture ibride **(NUMA)**

DISTRIBUTED-MEMORY SYSTEMS

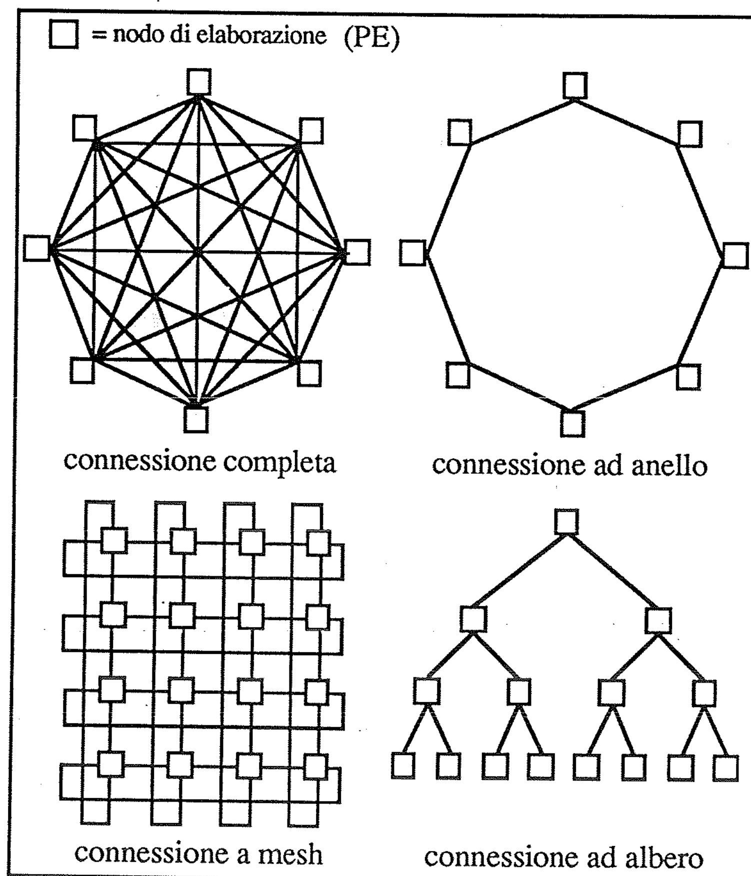
Modello di architettura parallela a memoria distribuita:



Classificazione secondo Flynn:

MIMD
(Multiple Istructions Multiple Data)

INTERCONNECTION NETWORKS



THE LOCAL ENVIRONMENT MODEL

Un programma concorrente é composto da un numero variabile di processi sequenziali comunicanti e cooperanti

PROCESSO
=
(codice sequenziale,
stato di avanzamento,
ambiente privato)

Il modello ad ambiente locale non prevede né spazi di memoria comune, né componenti passive (risorse comuni, procedure condivise in mutua esclusione, ecc.).

I processi concorrenti interagiscono esclusivamente con scambio esplicito di messaggi per:

- Scambiarsi dati, quando un valore calcolato da un processo viene comunicato ad un altro processo
- Sincronizzarsi, quando é necessario imporre un ordinamento (parziale o totale) nell'esecuzione (di parti) di processi

THE LOCAL ENVIRONMENT MODEL

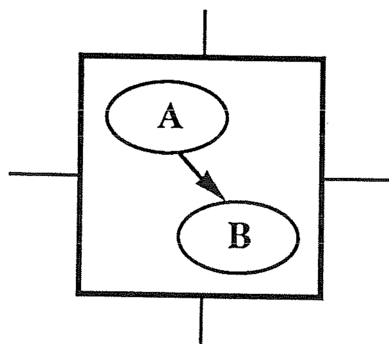
Processo A

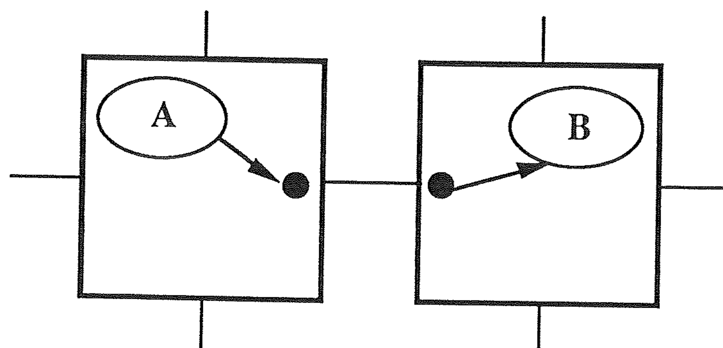
.
.
send(B, msg);
.
.

Processo B

.
.
receive(A, vtg);
.
.

Processore 1





PARALLEL PROGRAMMING

Desiderata:

Partire da un Programma (sequenziale o concorrente) ed ottenere automaticamente, in maniera invisibile per l'utente, la sua esecuzione parallela ottima su una architettura a parallelismo massiccio

Implicazioni su :

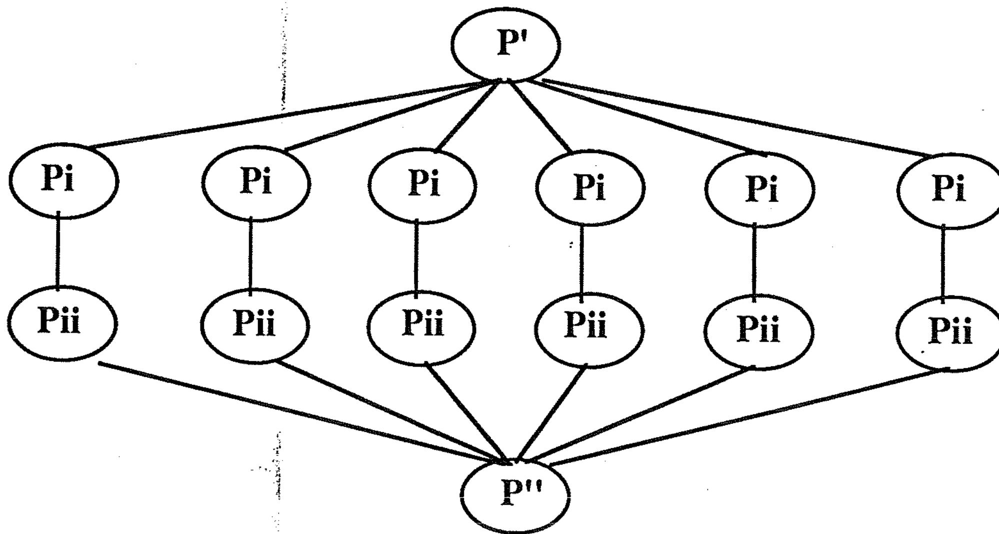
ARCHITETTURA:

- interconnessione efficiente
- diametro ridotto
- parallelismo nei links
- tolleranza ai guasti
- riconfigurabilità
- ecc.

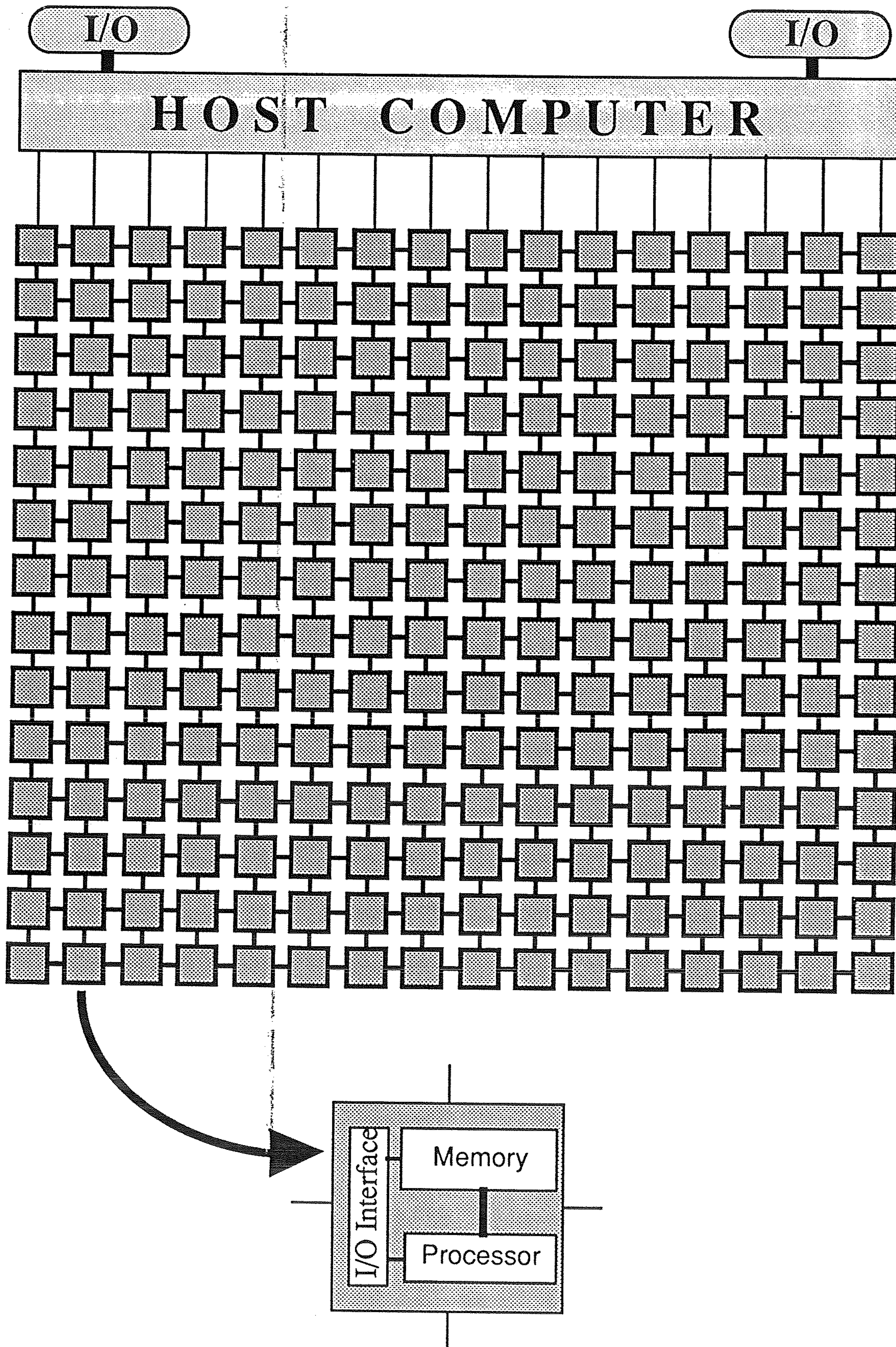
AMBIENTE DI PROGRAMMAZIONE:

- S.O. parallelo multiuser e multitasking
- supporto run-time efficiente
- tools per mapping ottimale, bilanciamento dinamico del carico, ecc.

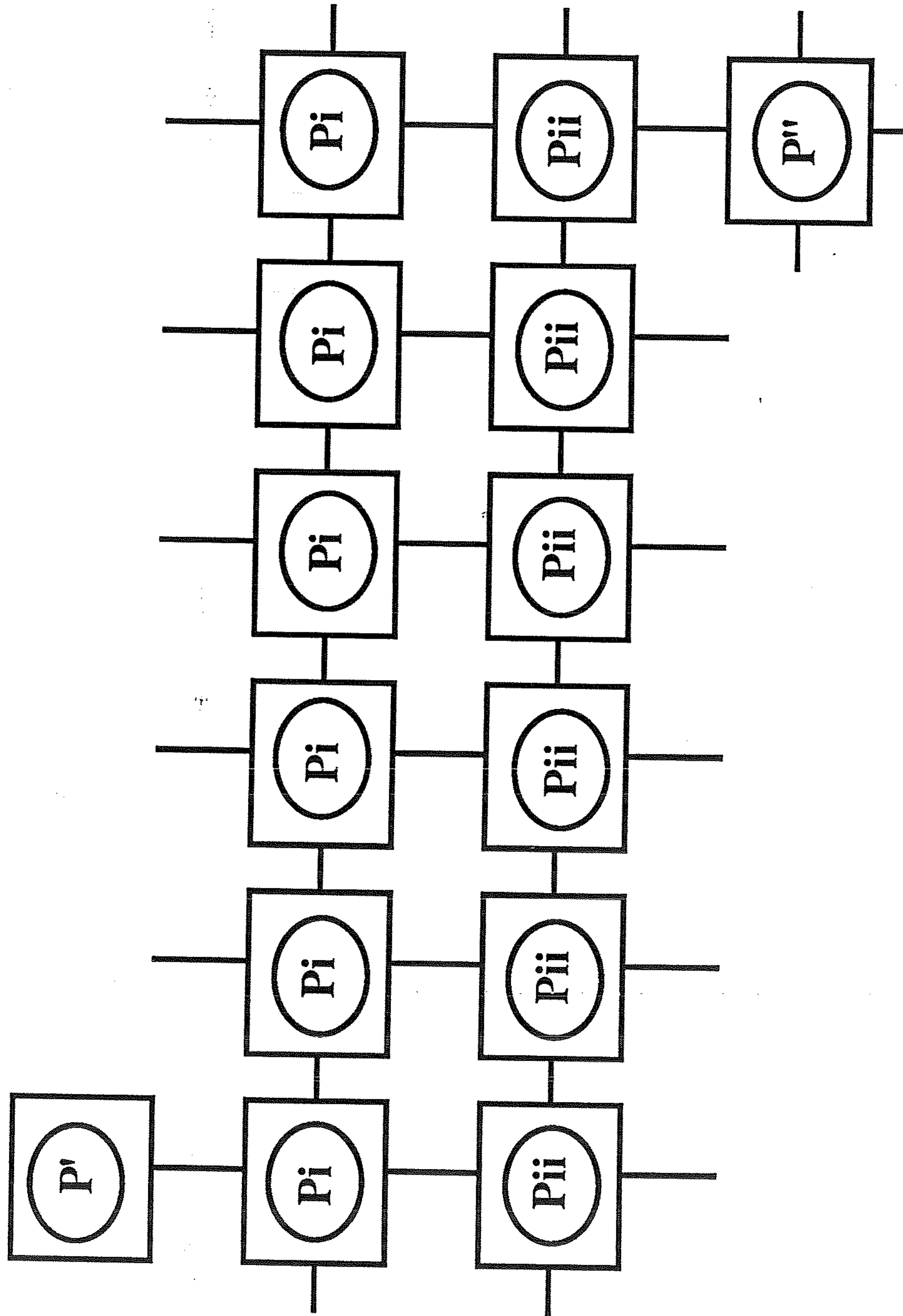
Process Interaction Graph



TARGET ARCHITECTURE



MAPPING



PARALLELISM LEVEL AND GRAIN

Coarse Grain

(Computer Networks)

Message Passing Systems

MIMD shared-memory

SIMD machines

VLIW machines

Fine Grain

Massive Parallelism

SIMD machines

Message Passing Systems

(Computer Networks)

MIMD shared-memory

VLIW machines

Low Parallelism

PARALLEL ARCHITECTURES

1) NCUBE family

2) TRANSPUTER

NCUBE family

NCUBE Corporation, Beaverton, OR, USA

Elaboratori MIMD ad ambiente locale

Da 4 a 1024 nodi interconnessi in una topologia ad ipercubo

NCUBE/four
workstation costituita da un IBM PC-AT compatibile (con funzioni di Host) potenziato con 4, 8, o 16 nodi NCUBE

NCUBE/seven
minisupercomputer stand-alone con 16-128 nodi

NCUBE/ten
supercomputer stand-alone con 64-1024 nodi
Performance di picco : 400 MFLOPS, 2500 MIPS

HYPERCUBE TOPOLOGY

$N=2^n$ nodi di elaborazione

Ogni nodo é posto al vertice di un cubo binario in uno spazio ad *n* dimensioni, ed ha collegamenti diretti, lungo gli spigoli del cubo, con *n* altri nodi suoi vicini.

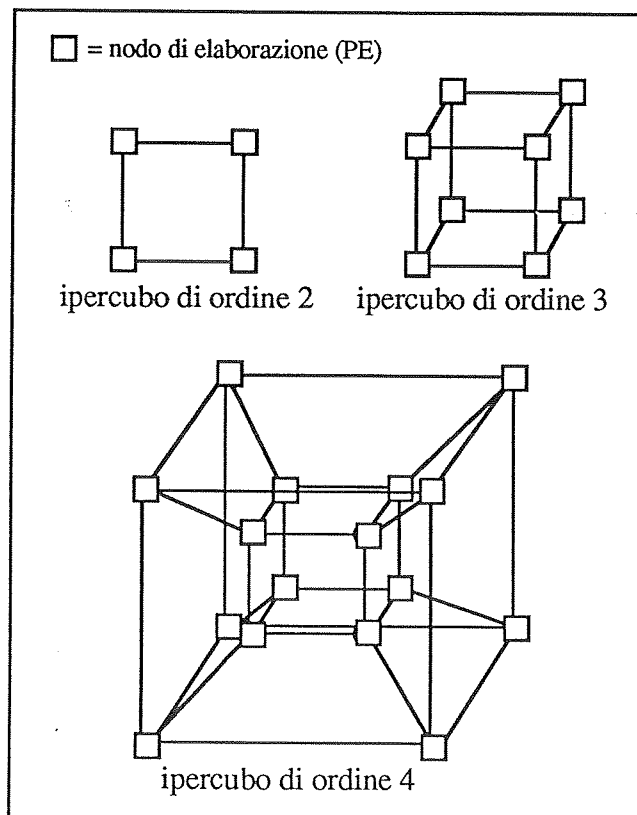
- connettività abbastanza elevata (**N*n/2** connessioni)

- regolarità (scalabilità ricorsiva)

- diametro ridotto ($n=\log_2 N$)

- relativa facilità di programmazione

- semplicità dell'algoritmo di routing



THE NCUBE MICROPROCESSOR

- Potenza paragonabile a quella di un VAX 11/780

- Set di istruzioni VAX-*like*

- Monotask

- ALU per interi a 32 bit

- Unità *floating point* a 64 bit (standard IEEE 754)

- 16 registri *general purpose*

- 13 *special purpose*

- Unità per il prelievo e la decodifica delle istruzioni con funzionamento a *prefetch*

- Interfaccia verso la memoria esterna.

- 22 canali DMA per l'*input* (11) e l'*output* (11) dei messaggi

THE NCUBE PROCESSOR BOARD

- 64 nodi di elaborazione su singola scheda connessi in un ipercubo di ordine 6

- Performance di picco: 100 MIPS, 30 MFLOPS

- Buon esempio di impiego di tecnologia VLSI

Ogni nodo è costituito da 7 *chip* (HMOS) :

- 6 chip di memoria *error-correcting* per una capacità totale di 512 Kbytes,

- un *microprocessor* NCUBE che combina le funzioni di processore e *I/O channel*

THE NCUBE HOST BOARD

Supporta :

- il sistema operativo ed il software di base
- i gestori delle periferiche
- 8 terminali
- il collegamento su Ethernet (opzionale)

La *host board* contiene :

- un microprocessore INTEL 80286
- un coprocessore 80287
- 2 Mbytes di memoria a doppia porta (una porta per il processore ed una per il *disk controller*)
- 16 nodi NCUBE.

I 16 nodi interni alla *host board*, configurati come due ipercubi di ordine 3, provvedono alle comunicazioni tra la *host board* ed il resto del sistema. A questo scopo, 8 degli 11 canali di ciascun nodo sono collegati ai *processor board* e forniscono un *I/O bus* ad alta velocità per il caricamento ed il controllo dei nodi di elaborazione.
Le memorie locali dei processori NCUBE interni alla *host board* possono essere accedute direttamente oltre che dal processore locale, dal *disk controller* e dal processore INTEL 80286.

Nel sistema NCUBE/four il compito della *host board* é svolto dal processore INTEL 80286 del PC.

THE NCUBE OPERATING SYSTEM

AXIS :

- Sistema operativo UNIX-like dell'HOST
- *multiuser* e *multitasking*
- Gestisce un *file system* distribuito
- Gestisce il partizionamento dell'ipercubo in domini statici

VERTEX :

- Nucleo di S.O. di Nodo
- Gestisce a software le comunicazioni internodo:
  - Primitive ***nwrite*** *(send)* e ***nread*** *(receive)*
  - Funzione *whoami*
  - Routing dei messaggi (store and forward)

L'ipercubo é trattato come un attached processor con funzioni di acceleratore computazionale

NCUBE PROGRAMMING LANGUAGES

- C

- FORTRAN 77

I linguaggi sono integrati con funzioni di libreria che forniscono l'interfaccia del linguaggio verso **AXIS** e **VERTEX**.

I valori delle variabili possono essere condivisi solamente attraverso esplicite chiamate alle funzioni di VERTEX *nwrite* e *nread*.

NCUBE family
NCUBE Corporation, Beaverton, OR, USA

Elaboratori MIMD ad ambiente locale

Da 4 a 1024 nodi interconnessi in una topologia ad ipercubo

NCUBE/four
workstation costituita da un IBM PC-AT compatibile (con funzioni di Host) potenziato con 4, 8, o 16 nodi NCUBE

NCUBE/seven
minisupercomputer stand-alone con 16-128 nodi

NCUBE/ten
supercomputer stand-alone con 64-1024 nodi
Performance di picco : 400 MFLOPS, 2500 MIPS

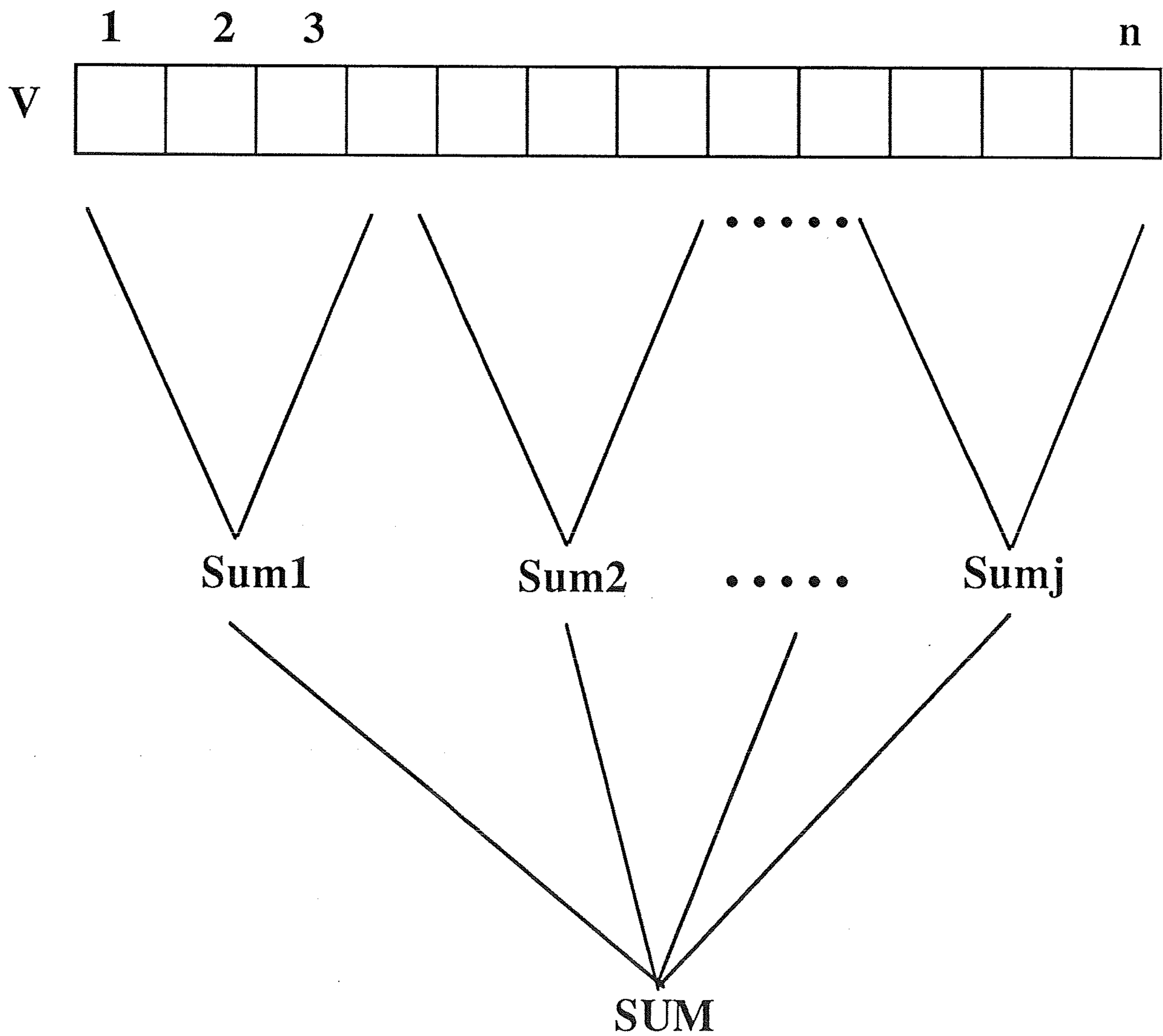
PARALLEL ARCHITECTURES

1) NCUBE family

2) TRANSPUTER

NCUBE PROGRAMMING EXAMPLE

CALCOLO DI SUM(V(I)**2)



NCUBE PROGRAMMING EXAMPLE

CALCOLO DI SUM(V(I)**2)

PROGRAMMA DELL'HOST

```
CHAN = NOPEN(CUBE_SIZE)

DO 10 I= 0, ((2**CUBE_SIZE) - 1)

RCODE = NLOAD(CHAN, "File name", I, BUFFER, BUFF_SIZE)

10 CONTINUE

* input del vettore V di N elementi *

N_EL = N / (2**CUBE_SIZE)

DO 20 I= 0, ((2**CUBE_SIZE) - 1)

RCODE = NWRITE(CHAN, V(N_EL * I), N_EL * 4, TYPE)

20 CONTINUE


RCODE = NREAD(CHAN, SUM, 4, 0, TYPE)


RCODE = NCLOSE(CHAN)

* output del risultato *
```

NCUBE PROGRAMMING EXAMPLE
CALCOLO DI SUM(V(I)**2)

PROGRAMMA DEL NODO

```
CALL WHOAMI(PN,PROC,HOST,M)
```

PN : numero del processore chiamante nel sotto-ipercubo allocato
PROC : nome unico del processo allocato sul nodo PN
HOST : nome del processo coordinatore per PN allocato sull'host
M : dimensione dell'ipercubo allocato per il job

```
SR = NREAD(V,N*4,HOST,TYPEH,FLAG1)
S = 0
DO 1 I =1, N
1     S = S + V(I)**2
```

Collezione dei risultati: il loop 2 viene eseguito una volta per ogni dimensione dell'ipercubo allocato

```
DO 2 I= M, 1, -1
IF (PN .LT. 2**I) THEN
```

se questo nodo fa parte del sotto-ipercubo di ordine I ancora attivo, vengono eseguiti i passi successivi, altrimenti termina l'esecuzione. NPN è il vicino di PN connesso lungo l'asse I.

```
    NPN = PN .NEQV. (2**(I-1))
    IF (NPN .LT. PN) THEN
```

se il numero di NPN è minore di PN invio dell'accumulazione parziale, altrimenti ricezione e aggiornamento del valore di S.

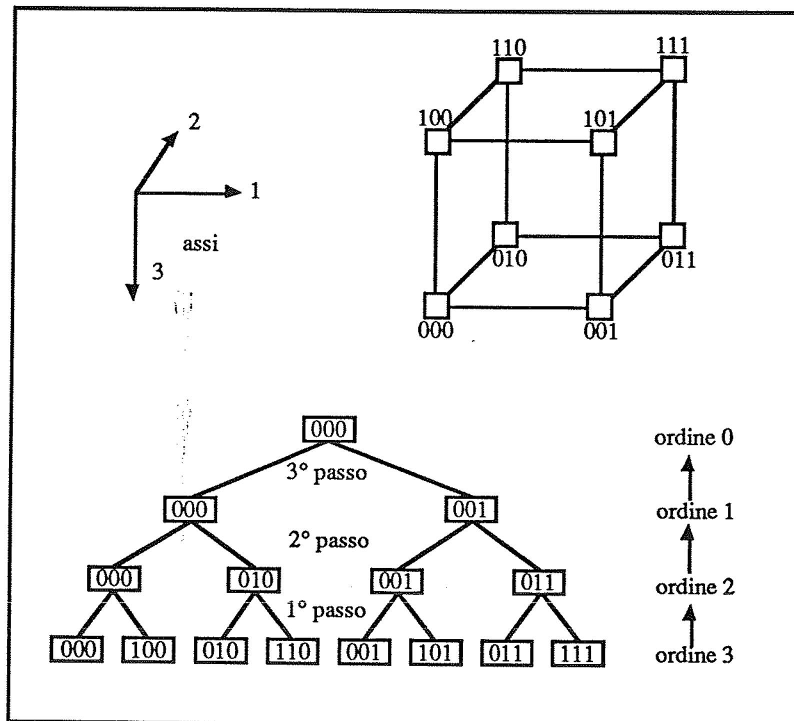
```
        SW = NWRITE(S,4,NPN,TYPEN,FLAG2)
    ELSE
        SR = NREAD(A,4,NPN,TYPEN,FLAG3)
        S = S + A
    ENDIF
ENDIF
2  CONTINUE
```

il nodo 0 invia all'host il risultato finale dell'algoritmo.

```
IF (PN .EQ. 0) THEN
    SW = NWRITE(S,4,HOST,TYPEH,FLAG4)
ENDIF
```

NCUBE PROGRAMMING EXAMPLE

CALCOLO DI SUM(V(I)**2)



TRANSPUTER FAMILY
INMOS Ltd., Bristol, UK

Commercializzati nel 1984, i Transputer sono tutt'oggi gli unici microprocessori progettati per supportare un linguaggio concorrente.

(OCCAM basato sul modello CSP (Hoare))

TRANSPUTER T414-20

- Architettura RISC, 32 bit, 10 MIPS, 20 MHz

- 2 Kbytes RAM on chip (50 ns)

- 4 links seriali bidirezionali asincroni
  (1 Mbyte/sec. ciascuno)

- una interfaccia di 32 bit alla memoria esterna
  (26.6 Mbytes/sec.)

- hardware scheduler (circa 1 μs per il context switch)

TRANSPUTER T800-20

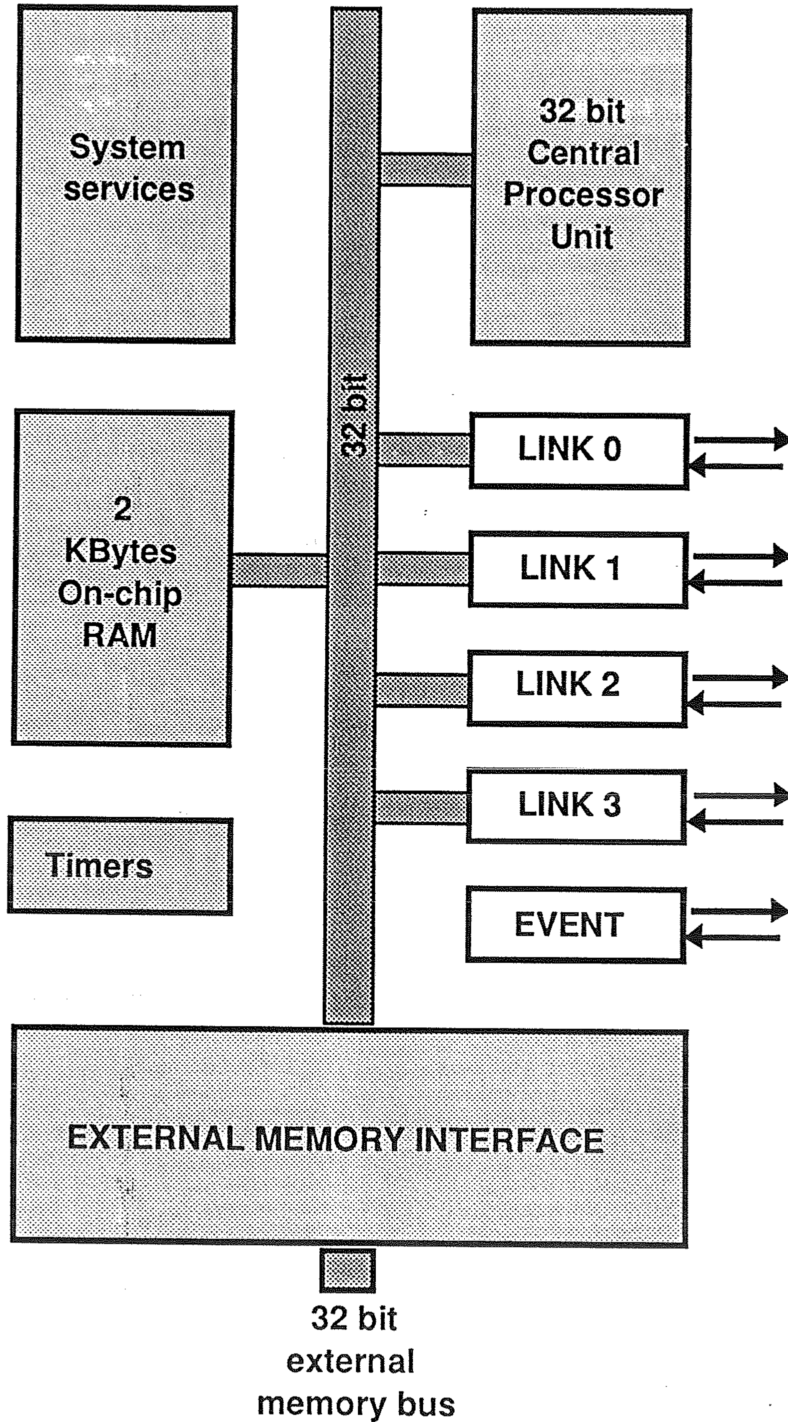
- 32 bit, 10 MIPS, 1.5 MFLOPS (singola precisione)

- 4 Kbytes RAM on chip (50 ns)

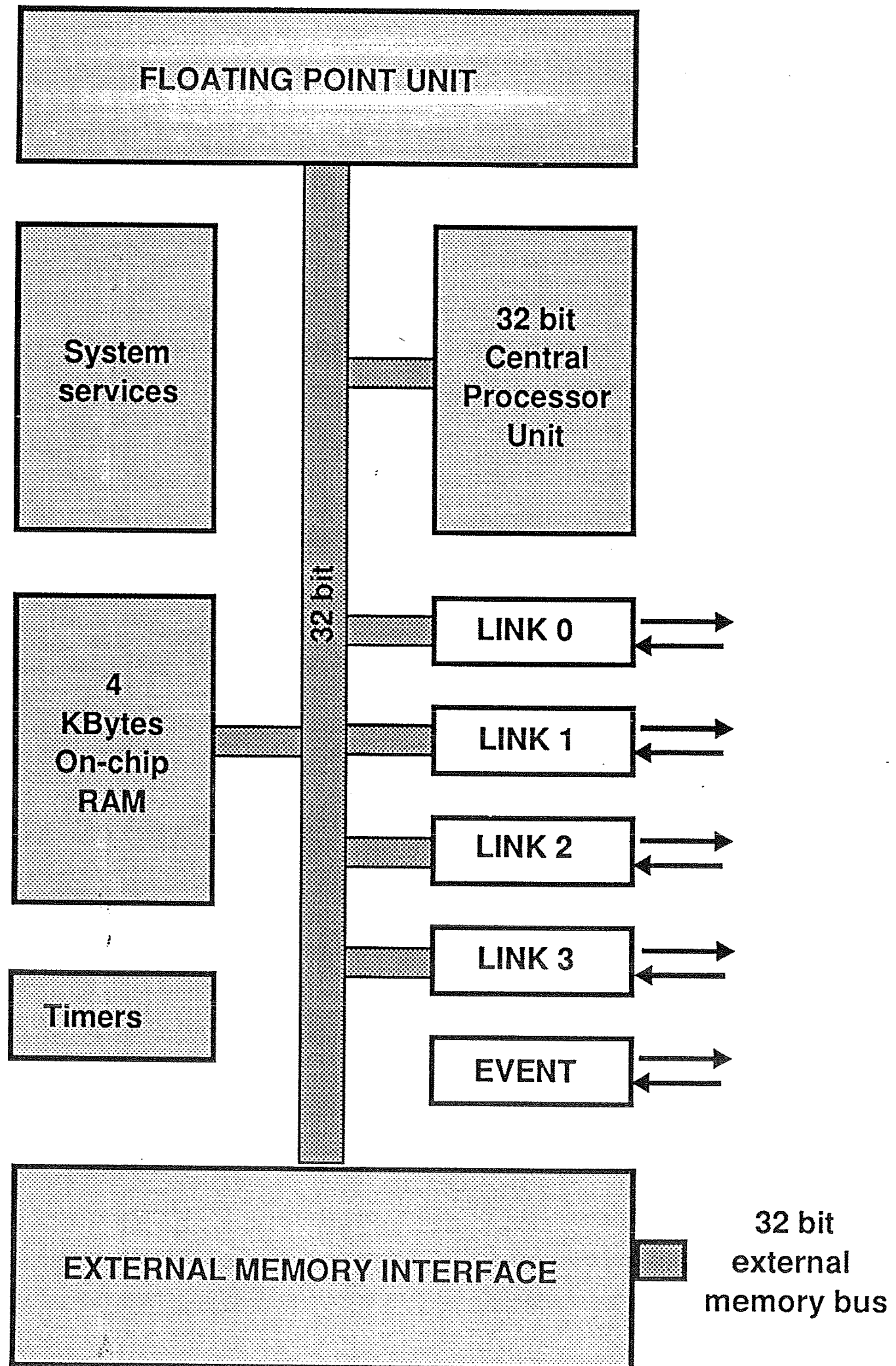
- unità Floating Point 64 bit standard IEEE 754

- miglioramenti sul protocollo asincrono dei link
  (1.8 Mbytes/sec. su ogni canale fisico)

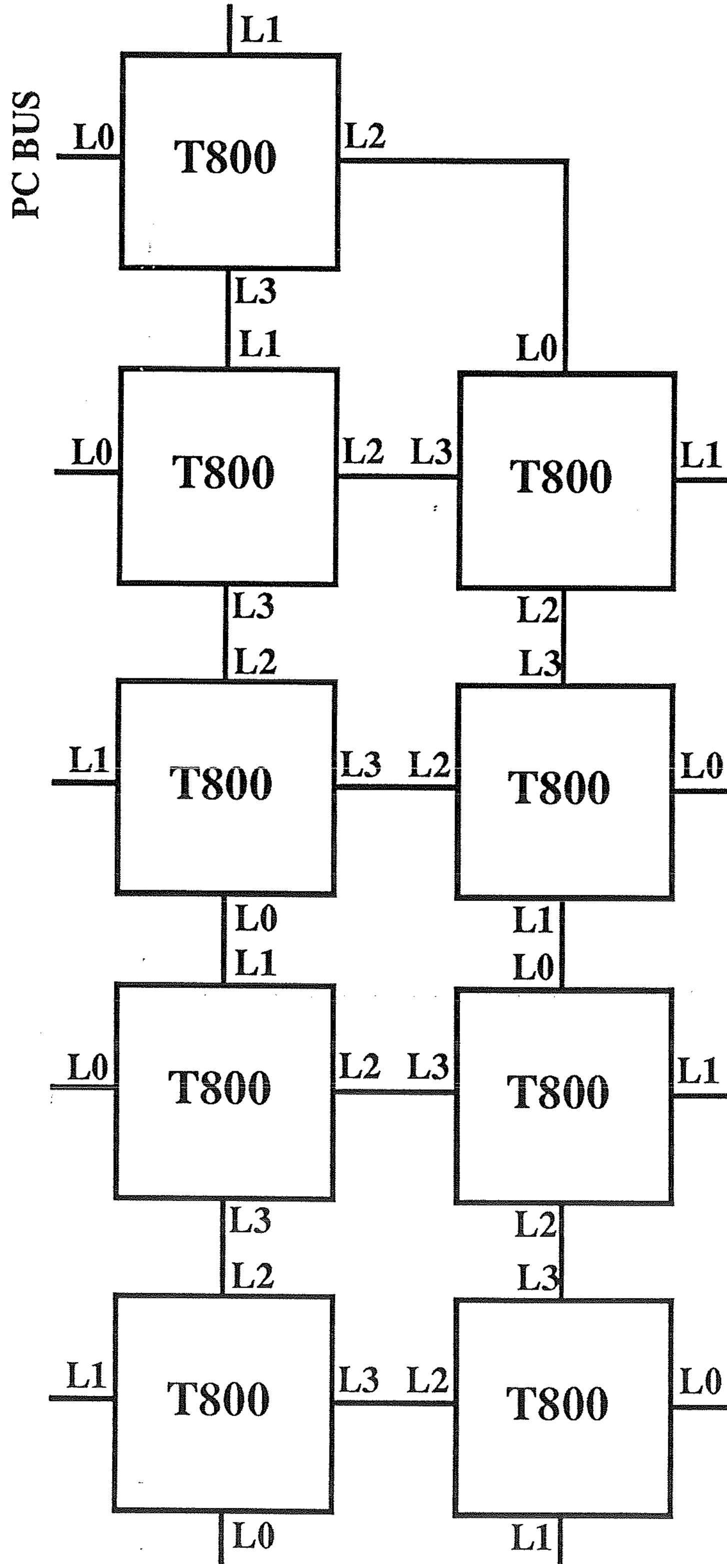
INMOS T414



INMOS T800



INMOS ITEM



OCCAM LANGUAGE

- Linguaggio nativo dei Transputer
- I processi concorrenti di Occam sono implementati direttamente sui Transputer
- La comunicazione tra processi allocati su processori diversi utilizza i links fisici
- La comunicazione tra processi allocati su uno stesso processore utilizza una struttura dati allocata in memoria
- Lo scheduling e i comandi di I/O sono istruzioni Occam microcodificate
- La comunicazione tra processi Occam é simmetrica sincrona a rendez-vous stretto
- I processi patners di una comunicazione riferiscono il nome di una stessa <u>Porta</u>
- L'implementazione di Occam su Transputer e molto efficiente ma é statica

OCCAM LANGUAGE

I programmi Occam sono costruiti a partire dai tre processi primitivi:

- **assegnamento** (v := exp)
- **output** (c ! expr)
- **input** (c ? vtg)

I processi primitivi sono combinati assieme tramite :

Costrutto sequenziale

```
SEQ
    from_P ? v
    v := v + 1
    to_next ! v
```

Costrutto parallelo

```
PAR
    from_P ? v1
    from_Q ? v2
    SEQ
        k := j / 2
        n := k + m
```

Costrutti condizionali (IF, CASE)

Costrutti ripetitivi (WHILE)

NONDETERMINISM CONTROL
(Costrutto alternativo)

Un linguaggio concorrente deve avere dei meccanismi per selezionare nondeterministicamente una comunicazione tra un insieme di possibilità. Tra due scelte nondeterministiche di un processo, la computazione é completamente deterministica.

```
ALT
  (y < 0) & chan1 ? v1
    ... first process
  (y = 0) & chan2 ? v2
    ... second process
  (y > 0) & chan3 ? v3
    ... third process
```

NONDETERMINISM CONTROL

```
SEQ
  Clock ? timenow
  ALT
    chan1 ? v1
      ... first process
    chan2 ? v2
      ... second process
    Clock ? AFTER timenow PLUS timeout
      ... third process
```

```
SEQ
  going := TRUE
  WHILE going
    PRI ALT
      stop ? ANY
          going := FALSE
      chan1 ? v1
          ... first process
      chan2 ? v2
          ... second process
      ......
```

PROCESS STRUCTURING (OCCAM)

Un processo P può attivare un insieme di processi {P1,P2,..,Pn} mediante un comando parallelo

```
PROC P (..)
  ... declarations
  PAR
    ... process P1
    ... process P2
    ......
    ... process Pn
```

o con un replicatore

```
PROC Pi (CHAN OF INT from_previous, to_next)
....
....
:
```

```
[n + 1] CHAN OF INT pipe :
PAR i = 0 FOR n
  Pi (pipe[i], pipe[i + 1])
```

BUFFERING
(Occam)

```
PROC Prod (CHAN OF INT to_buffer, ....)
  INT item :
  BOOL going :
  .....
  SEQ
    going := TRUE
    WHILE going
      SEQ
          ... produci item
          to_buffer ! item
  ....
:
```

```
PROC Cons (CHAN OF INT from_buffer,
           CHAN OF ANY req, ....)
  INT item :
  BOOL going :
  .....
  SEQ
    going := TRUE
    WHILE going
      SEQ
        req ! any
        from_buffer ? item
        ... consuma item
  ....
:
```

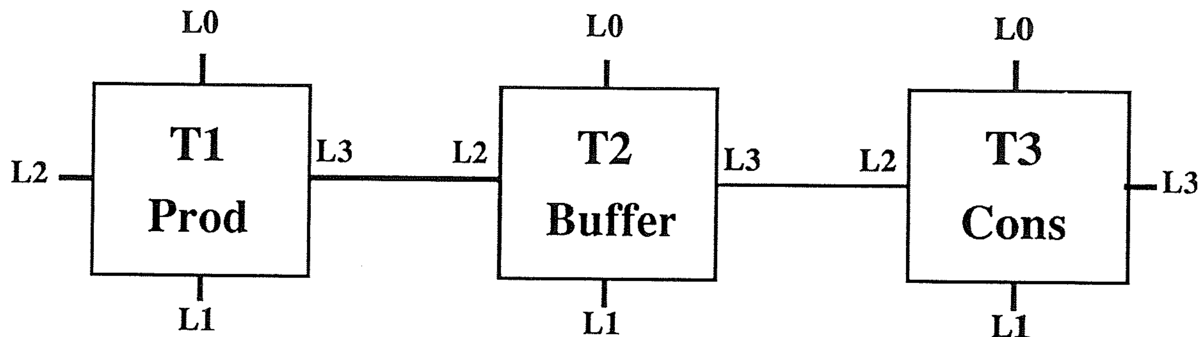
BUFFERING
versione 1

```
VAL n IS ...

PROC Buffer (CHAN OF INT from_Prod, to_Cons,
             CHAN OF ANY req,
              ....)
[n]INT buffer :
INT count, inpointer, outpointer :
SEQ
  count := 0
  inpointer := 0
  outpointer := 0

  WHILE TRUE
    ALT
      count > 0 & req ? ANY
        SEQ
          to_Cons ! buff[outpointer]
          outpointer := (outpointer + 1) REM n
          count := count - 1

      count < n & from_Prod ? buff[inpointer]
        SEQ
          inpointer := (inpointer + 1) REM n
          count := count + 1
:
```

BUFFERING
versione 2

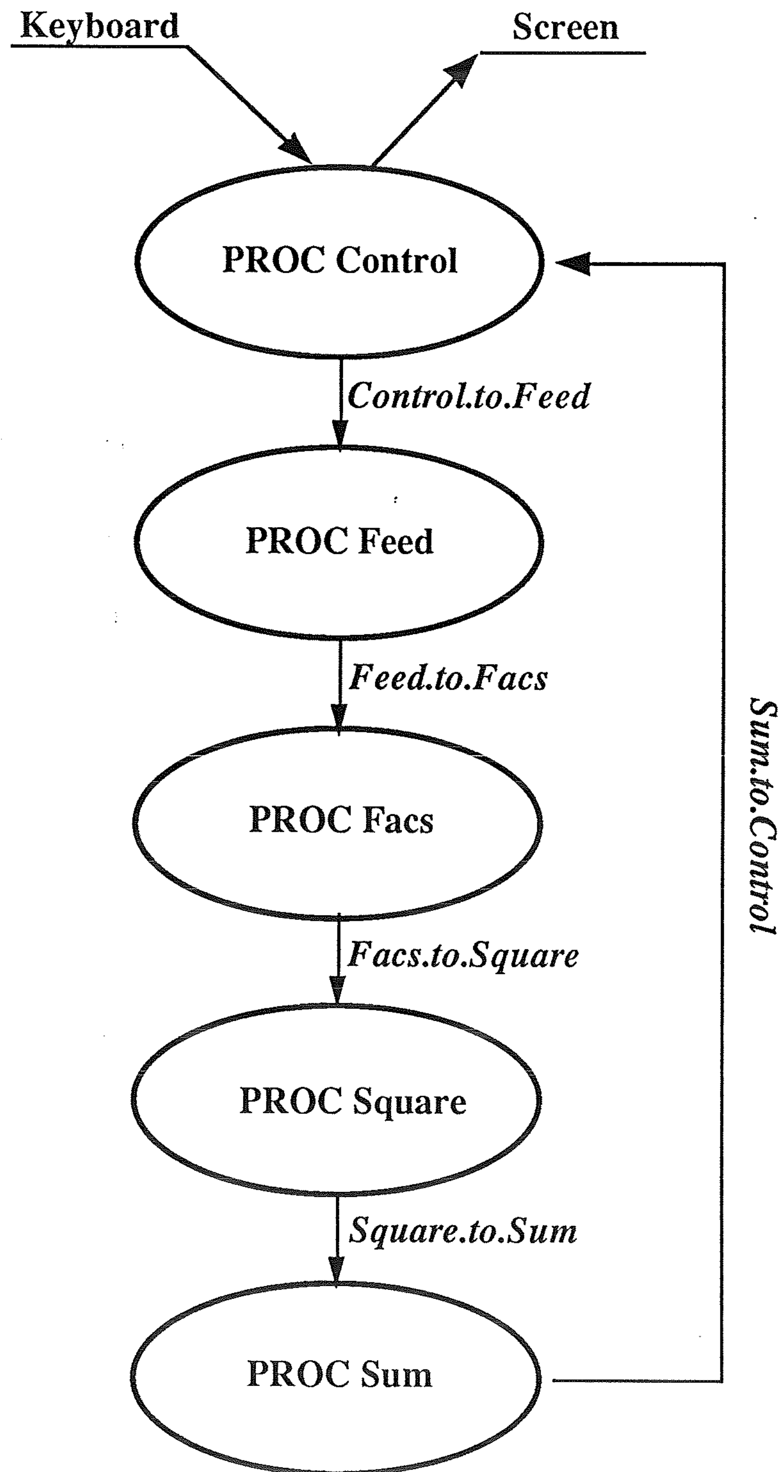
```
VAL n IS ...

PROC Buffer (CHAN OF INT from_Prod, to_Cons)
  [n - 1] CHAN OF INT pipe :
  INT item :
  PAR
    SEQ
      WHILE TRUE
        from_Prod ? item
        pipe[0] ! item

    SEQ
      WHILE TRUE
        pipe[n - 2] ? item
        to_Cons ! item

    PAR i = 0 FOR n - 2
      WHILE TRUE
        INT b :
        SEQ
          pipe[i] ? b
          pipe [i + 1] ! b
:
```

BUFFERING
configurazione




```
CHAN OF INT Prod_to_Buffer, Buffer_to_Cons :
CHAN OF ANY req :
....

PLACED PAR

  PROCESSOR 1 T8
    PLACE Prod_to_Buffer AT link3out :
    Prod(Prod_to_Buffer, ...)

  PROCESSOR 2 T8
    PLACE Prod_to_Buffer AT link2in :
    PLACE Buffer_to_Cons AT link3out :
    PLACE req AT link3in :
    Buffer(Prod_to_Buffer, Buffer_to_Cons, req )

  PROCESSOR 3 T8
    PLACE Buffer_to_Cons AT link2in :
    PLACE req AT link2out :
    Cons(Buffer_to_Cons, req )
```

Calcolo della somma dei quadrati dei fattoriali dei primi n interi



Calcolo del valore di Π su N Transputer

$$\Pi = \int_0^1 \frac{4}{1 + X^2} dx$$

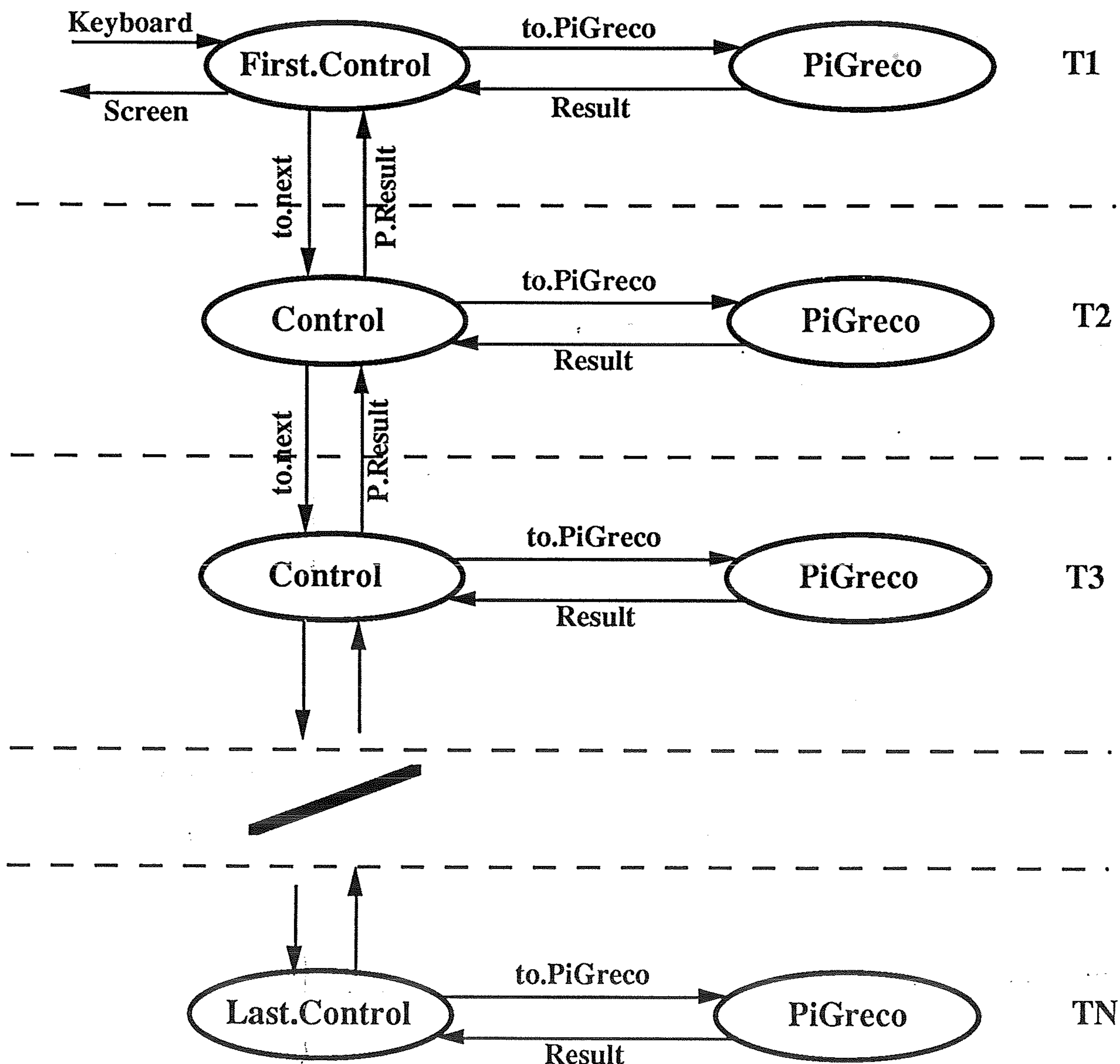
Applicando la regola del rettangolo abbiamo:

$$\lim_{n \to \infty} \sum_1^n \frac{4}{\left(\frac{i - 0.5}{n}\right)^2 + 1} = \Pi$$

6n operazioni Floating Point in doppia precisione (64 bit)

Possiamo suddividere il calcolo di Π in N sommatorie parziali assegnandone ciascuna ad un diverso processore collegato a pipeline.

Calcolo del valore di Π su N Transputer



Calcolo del valore di Π: alcuni risultati

ALLIANT FX/8, 8 Processori, 200M intervalli
17.5 MFLOPS

Butterfly BBN, 29 Processori, 10M intervalli
3.3 MFLOPS

CRAY XM/P, 4 Processori, 10M intervalli
116 MFLOPS

Sequent Balance, 29 Processori, 10M intervalli
2.2 MFLOPS

Dati presi dal testo "Programming Parallel Processors", edito da R. J. BABB II

INMOS ITEM, 10 Processori, 10M intervalli
7.13 MFLOPS
Speedup = 9,98

BIBLIOGRAFIA TEMATICA

SURVEYS ON PARALLEL ARCHITECTURES :

M. Vanneschi : Architetture general purpose, *Atti del convegno AICA-AEI su Elaborazione Parallela*, Milano, novembre 1988, p. 49-80.

D. P. Agrawal : *Advanced computer architecture, tutorial*, IEEE Computer Society Press/North-Holland, 1986.

O. A. McBryan : New architectures: performance highlights and new algorithms, *Parallel Computing*, n.7, 1988, p.477-499.

Hockney R.W. e Jesshope C.R., *Parallel Computers 2*, Adam Hilger, 1988

Hwang K. e Briggs F.A., *Computer architecture and parallel processing*, McGraw-Hill, 1985

R.J. Babb II (editor): *Programming Parallel Processors*, Addison-Wesley, 1988

CONCURRENT PROGRAMMING LANGUAGES:

C. A. R Hoare : Communicating sequential processes, *Communication of the ACM*, 21, n.8, Agosto 1978, p.666-677.

F. Baiardi, M. Vanneschi : *Linguaggi per la programmazione concorrente*, 272.1, Franco Angeli editore, 1985.

D. Pountain, D. May : *A tutorial introduction to Occam Programming*, INMOS, BSP Professional Books, 1987, U.K.

NCUBE :

J. P. Hayes, T.N. Mudge, Q.F.Stout : Architecture of a hypercube supercomputer, *Proceedings of the 1986 international conference on Parallel Processing*, agosto 1986, p.653-660.

D. Jurasek, W. Richardson, D.Wilde : A multiprocessor design in custom VLSI, *VLSI System Design*, giugno 1986, p.26-31.

NCUBE Corporation : *Users handbook*, versione P2.1, ottobre 1987.

J. L. Gustafson, G. R. Montry, R.E. Benner : Development of parallel methods for a 1024 processor hypercube, *SIAM journal on scientific and statistical computing*, Vol 9, n. 4, luglio 1988, p.609-638.

R. Perego, D. Laforenza : Gli ipercubi: una nuova generazione di elaboratori, mensile *Informatica Oggi*, n.50, Maggio 1989.

TRANSPUTER :

D. May, R. Shepherd, C. Keane, "Communicating Process Architecture: Transputer and Occam", in *Future Parallel Computers*, P. Treleaven & M. Vanneschi (eds.), *Lecture Notes in Computer Science*, n. 272, Springer-Verlag,1987.

INMOS: *Transputer Development System*, Prentice Hall International, 1988, U.K.