Capitolo 1

Calcolo parallelo: l'idea di base

Una misura delle prestazioni del software è fornita dal tempo necessario alla sua esecuzione in uno specifico ambiente di calcolo (efficienza del software).

In generale, una rappresentazione semplificata[1] di tale tempo è:

$$\tau \cong k \cdot T(n) \cdot \mu$$

dove:

- $T(n)$ è la funzione complessità di tempo, definita dal numero di operazioni richieste dall'algoritmo;
- μ è il tempo di esecuzione di 1 operazione e dipende dal *periodo di clock*;
- k è un fattore di proporzionalità.

[1] Tale rappresentazione non tiene conto di alcuni fattori che dipendono dall'architettura utilizzata.

Le fasi di elaborazione in un calcolatore sono scandite dal segnale di clock emesso per sincronizzare le varie unità. L'intervallo temporale che intercorre tra due impulsi successivi è detto periodo (o ciclo) di clock.
Un'importante caratteristica di un calcolatore è la *peak performance*, P_{max}, che specifica il numero massimo di operazioni floating point che possono essere teoricamente eseguite nell'unità di tempo (generalmente il secondo).
La peak performance si può calcolare come il rapporto tra il massimo numero N_c di operazioni che possono essere eseguite in un ciclo di clock e il tempo di ciclo T_c:

$$P_{max} = \frac{N_c}{T_c}$$

Essa è misurata in flop/sec (floating-point operations per second) o in Mflop/sec (10^6 flop/sec), Gflop/sec (10^9 flop/sec) e in Tflop/sec (10^{12} flop/sec).
Ad esempio il tempo di ciclo della workstation IBM Power2 modello 591 è $T_c = 13\ nsec$. Possono essere eseguite ad ogni ciclo di clock 4 operazioni floating point (una moltiplicazione e un'addizione). Quindi:

$$P_{max} = \frac{N_c}{T_c} = \frac{4\ operazioni}{13 \cdot 10^{-9}\ secondi} = 308\ Mflop/sec$$

Il Pentium III esegue 1 operazione floating point e $T_c = 650\ nsec$, quindi

$$P_{max} = \frac{N_c}{T_c} = \frac{1\ operazione}{65 \cdot 10^{-10}\ secondi} = 650\ Mflop/sec$$

Il processore Athlon esegue 2 operazioni floating point e $T_c = 600\ nsec$, quindi:

$$P_{max} = \frac{N_c}{T_c} = \frac{2\ operazioni}{6 \cdot 10^{-11}\ secondi} = 1200\ Mflop/sec$$

Il processore Power 3 esegue 4 operazioni floating point e $T_c = 375\ nsec$, quindi:

$$P_{max} = \frac{N_c}{T_c} = \frac{4\ operazioni}{375 \cdot 10^{-9}\ secondi} = 1500\ Mflop/sec$$

La frequenza f del processore, in termini di Hertz, indica quanti cicli vengono eseguiti in un secondo ed è quindi calcolabile come il reciproco di T_c, ovvero

$$f = \frac{1}{T_c}$$

Se rapportiamo tale quantità al numeri di cicli p necessari per eseguire una operazione otteniamo il numero di operazioni eseguite al secondo, cioè:

$$\frac{f}{p} = [Mflop/sec]$$

Tenuto conto di tale relazione è possibile calcolare la *peak performance* nel modo seguente:

$$P_{max} = N_c \times f$$

È quindi chiaro che la *peak performance* dipende strettamente dal fattore N_c ovvero dal numero massimo di operazioni floating point che possono essere eseguite in un ciclo di clock. In altre parole la frequenza del processore non basta a determinare le sue prestazioni massime.
I Mflop/sec possono essere utilizzati per confrontare le prestazioni di macchine diverse. Un *benchmark* è un programma scritto appositamente a tale scopo. Esistono varie collezioni di benchmarks. LINPACK, un package per l'algebra lineare, è spesso utilizzata per confrontare differenti architetture valutando il tempo di esecuzione dell'algoritmo di eliminazione di Gauss su di una matrice di ordine 100.
Il Linpack Benchmark utilizza le routine LINPACK/sgefa e LINPACK/sgesl per la soluzione di sistemi di equazioni lineari con $n = 100$. La prestazione P è determinata dal rapporto tra il numero di operazioni eseguite ($W = 2n^3/3 + 2n^2$) e il tempo effettuato per il calcolo T:

$$P = \frac{W}{T}[flop/sec]$$

Confrontando tale valore con la peak performance della macchina si ha un'indicazione della capacità con cui due routines di LINPACK sfruttano le risorse della macchina (http://www.netlib.org/benchmark/).

L'efficienza del software dipende dalla complessità di tempo dell'algoritmo e quindi dal numero di operazioni eseguite dall'algoritmo e dalle caratteristiche dell'ambiente di calcolo, ovvero dal tempo necessario alla esecuzione delle operazioni richieste dall'algoritmo e all'accesso dei dati in memoria.

È possibile dimostrare che per alcune classi di problemi esistono algoritmi con complessità di tempo minima; tali algoritmi sono, per

questo, detti *"ottimali"*. Ad esempio, per il prodotto di due matrici di dimensione $n \times n$, l'algoritmo di Strassen è ottimale[2] [50], oppure per la valutazione di un polinomio l'algoritmo di Horner è ottimale [46]. Come è anche possibile progettare algoritmi che ottimizzino gli accessi alla memoria.

[2] Il numero di operazioni scalari richieste per il prodotto di due matrici di dimensione $n \times n$ è $M(n) = 2n^3 - n^2$. Nel 1969 Strassen ha descritto un algoritmo per la risoluzione del prodotto matrice-matrice che richiede un numero di operazioni di base $S(n) = 7 \cdot 7^{lg\ n} - 6 \cdot 4^{lg\ n}$ dove lg è il logaritmo in base 2.

Date due matrici A e B, l'algoritmo di Strassen suddivide le due matrici in 2×2 blocchi:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

e calcola:

$$\begin{array}{ll} P_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) & C_{11} = P_1 + P_4 - P_5 - P_7 \\ P_2 = (A_{21} + A_{22}) \cdot B_{11} & C_{12} = P_3 + P_5 \\ P_3 = A_{11} \cdot (B_{12} - B_{22}) & C_{21} = P_2 + P_4 \\ P_4 = A_{22} \cdot (B_{21} - B_{11}) & C_{22} = P_1 + P_3 - P_2 + P_6 \\ P_5 = (A_{11} + A_{12}) \cdot B_{22} & \\ P_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) & \\ P_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) & \end{array}$$

Se viene applicato un solo livello dell'algoritmo di Strassen ad una matrice $n \times n$ i cui elementi sono blocchi di dimensione $n/2 \times n/2$ e viene poi utilizzato l'algoritmo standard per le sette moltiplicazioni di blocchi di matrice che occorrono nel calcolo dei P_i, il numero totale di operazioni è:

$$S(n) = 7 \cdot \left[2 \cdot \left(\frac{n}{2}\right)^3 - \left(\frac{n}{2}\right)^2\right] + 18 \cdot \left(\frac{n}{2}\right)^2 = \frac{7}{4}n^3 + \frac{11}{4}n^2$$

Il rapporto tra $S(n)$ e $M(n)$ è il seguente:

$$\frac{S(n)}{M(n)} = \frac{7n^3 + 11n^2}{8n^3 - 4n^2}$$

che all'umentare di n tende a $\frac{7}{8}$. Questo implica che per matrici sufficientemente grandi l'applicazione dell'algoritmo di Strassen produce un miglioramento del $12,5\%$ rispetto al prodotto matrice-matrice standard.

Ridurre μ è una sfida tecnologica (Nanotecnologia).
Per quanto riguarda i PC che tipicamente vengono utilizzati in casa, si è passati nel giro di pochi anni dai processori Intel con frequenza di clock di 100 MHz (486) agli attuali Pentium IV, appartenenti alla stessa casa produttrice, con una frequenza di 1.5 GHz. Tenendo conto che frequenza e periodo sono l'uno il reciproco dell'altro, risulta, nel caso del processore Intel, $\mu_c = 10^{-2}$, e, nel caso del processore Pentiun IV, $\mu_c = 10^{-9}$.

Considerazioni sulla riduzione di $T(n)$.
Per un fissato algoritmo con complessità T(n), sicuramente un principio generale per ridurre il tempo necessario all'esecuzione delle operazioni richieste dall'algoritmo e all'accesso ai dati in memoria è quello di tener conto, in fase di progettazione dell'algoritmo stesso, sia della località temporale sia di quella spaziale, organizzando opportunamente il flusso delle istruzioni e la fruizione dei dati. Infatti nella maggior parte del software vi sono istruzioni, spesso situate in aree ben localizzate del codice, che vengono eseguite ripetutamente, e si accede al resto delle istruzioni di rado. Se tali segmenti di software potessero risiedere il piú vicino possibile all'unità logica aritmetica (unità dei livelli di memoria: cache), il tempo totale di esecuzione verrebbe ridotto in modo significativo. In altre parole, sarebbe opportuno, in fase di memorizzazione delle istruzioni, tener conto del principio della località temporale, ovvero della probabilità che un'istruzione eseguita di recente venga eseguita nuovamente entro breve tempo. Accanto alla località temporale esiste un altro principio di località, ovvero la località spaziale che rappresenta la probabilità che istruzioni vicine[a] ad un'istruzione già eseguita di recente siano anch'esse eseguite quanto prima. L'aspetto temporale suggerisce di memorizzare un elemento (istruzioni o dati) nella memoria veloce quando questo viene richiesto per la prima volta, in modo tale che esso rimanga a disposizione nel caso di una nuova richiesta. L'aspetto spaziale suggerisce quindi di prelevare dalla memoria principale alla memoria veloce un insieme di elementi che risiedono in indirizzi adiacenti.

♣ **Esempio 4**
Si consideri un processore ad 1 GHz (1 nsec clock) connesso ad una DRAM con una latenza di 100 nsec. Si assuma, inoltre, che il processore abbia due unità per le operazioni FLOP [(b)] e che sia in grado di eseguire 4 istruzioni in ogni ciclo di 1 nsec.

[a] La vicinanza è espressa in termini di indirizzi delle locazioni di memoria contenenti le istruzioni.

[b] Un FLOP coincide con una operazione di somma e una di prodotto:

$$y := y + \alpha \cdot x$$

Il picco di prestazione del processore considerato è quindi di 4 $GFLOP/S$, mentre il tempo di latenza della memoria è di 100 cicli e la dimensione di blocco è una parola. Con tali caratteristiche, ogni volta che viene effettuata una richiesta alla memoria, si deve attendere 100 cicli prima di poter accedere al dato.
Si consideri il calcolo del prodotto di due matrici A e $B \in \Re^{n \times n}$:

$$c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} \cdot b_{ky} \;\; con \;\; i,j = 0, \ldots, n-1$$

dove $C \in \Re^{n \times n}$ è la matrice prodotto. Procedendo secondo lo schema di calcolo, effettuando il prodotto delle righe per le colonne, il prodotto $A \cdot B$ richiede n^2 prodotti scalari tra due vettori di dimensione n. L'esecuzione di un prodotto scalare, tra due vettori di dimensione n, richiede una moltiplicazione ed una somma su ogni coppia di elementi corrispondenti dei vettori, ovvero n FLOP. Per calcolare $A \cdot B$ bisogna, quindi, effettuare $n \cdot n^2$ FLOP. Se $n = 32$ il numero totale di FLOP (N_F) è

$$32^3 \approx 32K$$

Ogni FLOP richiede una fase di fetch, quindi il picco della velocità di calcolo è limitato dalla possibilità di eseguire un FLOP ogni 100 nsec. Per seguire $N_F = 32K$ FLOP occorre quindi un tempo T:

$$T = N_F \cdot 100 \; nsec = 32K \cdot 10^2 \; nsec = 32 \cdot 10^2 \; \mu sec$$

Si ha cosí un picco di 10 MFLOP/S, una frazione molto piccola della prestazione massima della macchina.

Supponiamo ora che il processore sia dotato di una cache di dimensione 32 KB con una latenza di 1 nsec o di 1 ciclo. Consideriamo nuovamente il prodotto tra le matrici A e B di dimensioni 32×32.

Si sono scelte con attenzione queste dimensioni in modo che la cache sia abbastanza grande da contenere le matrici A e B oltre alla matrice prodotto C. Inoltre, assumiamo una particolare strategia di allocazione della cache in modo che non vi sia sovrapposizione di dati.
Il caricamento delle due matrici nella cache corrisponde ad una fase di fetch di $2K$ words, che richiede circa 200 μsec . Le $32K$ operazioni richieste possono essere effettuate in $8k$ cicli che coincidono con 8 μsec, potendo eseguire 4 istruzioni per ciclo.

Il tempo totale per la computazione è dato dalla somma del tempo per l'acquisizione dei dati e del tempo per l'esecuzione delle operazioni stesse:

$$T = 200\ \mu sec + 8\ \mu sec = 208\ \mu sec$$

A tale valore di T corrisponde un picco computazionale di $32K/208\ \mu sec \approx 157$ MFLOP/S.
In questo caso l'introduzione di una seppur piccola cache permette di passare da un picco di 10 MFLOP/S a 157 MFLOP/S, cioè si possono migliorare considerevolmente le prestazioni del processore.

Vediamo ora cosa succede se portiamo la dimensione di blocco a 4 words (=1 long words = 1 lwords) nell'eseguire l'operazione di prodotto matrice per matrice[a].
Il caricamento delle due matrici nella cache corrisponde ad una fase di fetch di 512 lwords, che richiede circa 51 μsec.
Le $32K$ operazioni richieste possono essere effettuate in $8k$ cicli che coincidono con 8 μsec, potendo eseguire 4 istruzioni per ciclo. Il tempo totale per la computazione è dato dalla somma del tempo per l'acquisizione dei dati e del tempo per l'esecuzione delle operazioni stesse:

$$T = 51\ \mu sec + 8\ \mu sec = 59\ \mu sec$$

A tale valore di T corrisponde un picco computazionale di $32K/59\ \mu sec \approx 555$ MFLOP/S.
Si è cosí passati da 10 MPLOPS ai 555 MFLOP/S, ottenuti introducendo una memoria cache e l'utilizzo di un blocco di dimensione 4 words.

△

[a] Si sta ora considerando un'architettura VLIW, introdotta nel capitolo 1

Come ridurre ulteriormente il tempo richiesto dalla risoluzione di un problema? L'idea nuova è di organizzare i calcoli in modo diverso.

L'architettura di un calcolatore sequenziale è basato sullo schema funzionale della macchina di Von Neumann descritto in Fig. 1.1.

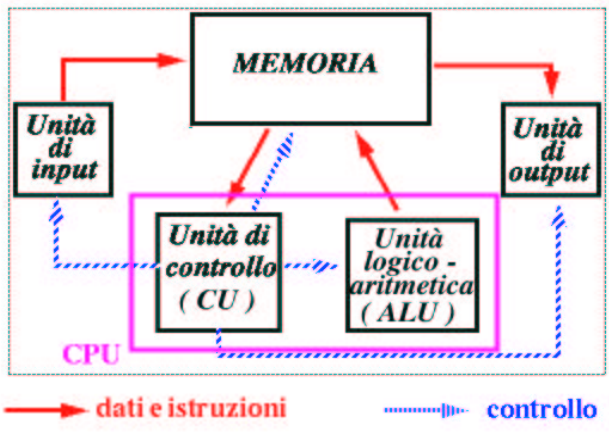

Figura 1.1: *Schema funzionale della macchina di Von Neumann*

In tale schema una singola unità di elaborazione è preposta all'esecuzione di un insieme di istruzioni. In particolare, è presente una singola unità preposta all'esecuzione delle operazioni floating point (**A**ritmetic **L**ogic **U**nit).

♣ Esempio 5

Si consideri la somma di due numeri floating point. Ricordiamo che, fissato il sistema aritmetico floating point, un numero è rappresentato dalla sua mantissa e dall'esponente. La somma di due numeri viene eseguita confrontando inizialmente gli esponenti dei due numeri, viene quindi eseguito uno shift della mantissa del numero con esponente piú piccolo e le due mantisse vengono sommate. Infine, il risultato viene normalizzato.

Siano $a = 0.956$ e $b = -1.23$, e fissiamo il sistema aritmetico $F = \{10, 3, -2, 20\}$. In tale sistema i due numeri sono cosí rappresentati:

$$fl(a) = 0.956 \times 10^0$$

$$fl(b) = -0.123 \times 10^1$$

Nei sistemi tradizionali l'operazione di addizione floating point è composta da 4 fasi

qui di seguito descritte.

I FASE	*confronto esponenti*	$esp(+0.956 \times 10^0) < esp(-0.123 \times 10^1)$
II FASE	*shift mantissa*	$+0.956 \times 10^0 = 0.095 \times 10^1$
III FASE	*somma mantisse*	$+0.095 - 0.123 = -0.028$
IV FASE	*normalizzazione*	$-0.028 \times 10^1 = -0.280 \times 10^0$

Mentre viene eseguita una delle quattro fasi i segmenti adibiti all'esecuzione delle altre operazioni rimangono inattivi. Ad esempio, mentre viene eseguito il confronto tra gli esponenti, i segmenti adibiti allo shift della mantissa, alla somma delle mantisse e alla normalizzazione non eseguono alcuna operazione.

△

In uno dei primi articoli introduttivi al Calcolo Parallelo "*Architetture per i supercalcolatori*" di G. Fox e P. Messina del 1987 [23] l'organizzazione dei calcolatori paralleli viene paragonata all'organizzazione di una squadra di operai che sovraintende alla costruzione di una casa. Nella macchina di Von Neumann è come se un operaio affrontasse l'intero lavoro da solo: egli espleta ognuno dei compiti un passo per volta, eseguendo le parti diverse di ogni lavoro secondo un certo ordine. Questo modo di costruire una casa è ovviamento lento: molti compiti possono essere assolti piú velocemente se vengono ripartiti tra operai diversi simultaneamente all'opera (come di fatto avviene).

In maniera analoga, l'unità funzionale adibita all'addizione f.p., l'ALU, può essere divisa in segmenti, ciascuno dei quali preposto all'esecuzione di una singola fase dell'operazione.

Questa idea è implementata già da diversi anni nei processori con ALU ***pipelined*** sia sul ciclo delle istruzioni sia su quello delle operazioni [51].

♣ Esempio 6

Si consideri l'esecuzione della somma di N coppie di numeri

$$(a_1, b_1), (a_2, b_2), \ldots, (a_N, b_N)$$

su di una unità tradizionale e su di una unità pipelined. La somma delle N coppie di numeri sui due tipi di unità è rappresentata in Fig. 1.2.

Sull'unità tradizionale ad ogni passo viene eseguita una fase del processo di somma su di una sola coppia di numeri. Sull'unità pipelined ad ogni passo vengono eseguite contemporaneamente tutte le fasi su coppie distinte di numeri.
Si indichi con t_u il tempo di esecuzione di un segmento dell'operazione. Su di una unità tradizionale solo un segmento alla volta è attivo mentre gli altri rimangono inattivi. Quindi il tempo di esecuzione è :

$$T_{trad} = 4N \cdot t_u$$

Invece, su di una unità pipelined a regime sono attivi tutti i segmenti contemporaneamente: vengono impiegati inizialmente $4t_u$ per andare a regime, successivamente, in $N-1$ passi, vengono effettuate tutte le somme, quindi:

$$T_{pl} = [4 + (N-1)] \cdot t_u = 4t_u + (N-1)t_u = 3t_u + Nt_u = (3+N)t_u$$

Essendo:

$$\frac{T_{pl}}{T_{trad}} = \frac{(3+N)\, t_u}{4N\, t_u} = \frac{3}{4N} + \frac{1}{4}$$

al crescere di N si ottiene che:

$$\frac{T_{pl}}{T_{trad}} = \lim_{N \to \infty} \frac{3}{4N} + \frac{1}{4} = \frac{1}{4}$$

ovvero, il fattore di riduzione del tempo T_{pl}, rispetto al tempo impiegato senza l'uso della pipeline T_{trad}, tende a $\frac{1}{4}$. Tale fattore è pari all'inverso del numero di stadi in cui viene suddivisa la pipeline; infatti nel precedente esempio si avevano 4 stadi. In generale, se s indica il numero di stadi in cui è suddivisa la pipeline, il fattore:

$$\frac{T_{pl}}{T_{trad}} = \frac{1}{s}$$

△

Un'unità pipelined è simile a una catena di montaggio in cui

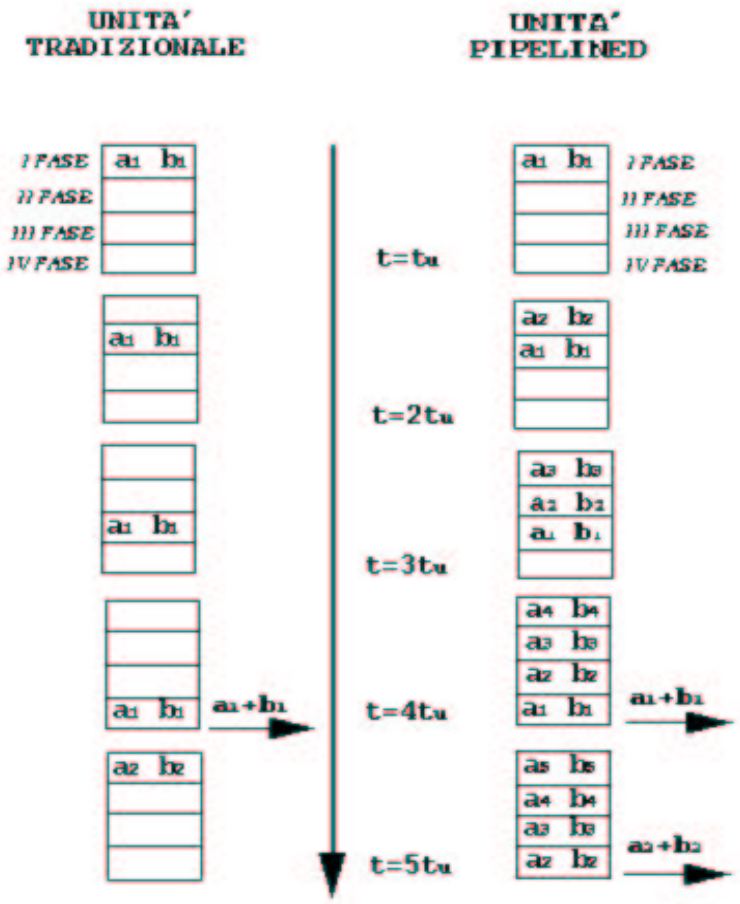

Figura 1.2: *Somma di 5 coppie di numeri su di una unità tradizionale e su di una unità pipelined*

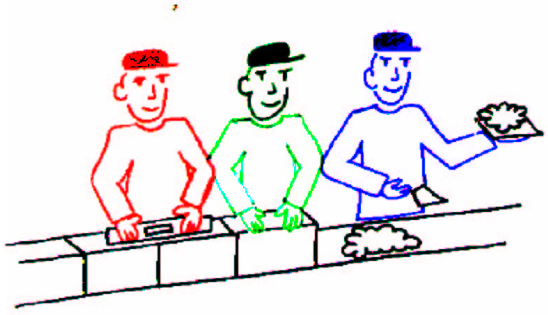

Figura 1.3: *Più operai eseguono contemporaneamente fasi successive dello stesso lavoro (parallelismo temporale)*

piú operai eseguono contemporaneamente fasi successive dello stesso lavoro, come mostrato in Fig. 1.3.

Questo tipo di parallelismo si definisce "***parallelismo temporale***" in quanto interviene sulla sequenza temporale in cui uno stesso insieme di dati viene elaborato.

♣ Esempio 7

L'esecuzione di una istruzione consiste di una sequenza di azioni detta *"ciclo delle istruzioni"* (istruction cycle). Tipicamente, un ciclo di istruzioni comprende le seguenti cinque fasi:

1. Instruction Fetch (IF): l'istruzione viene trasferita nei registri;
2. Decoding (D): l'istruzione viene decodificata e interpretata; vengono estratti gli indirizzi degli operandi;
3. Operand Fetch (OF): gli operandi vengono trasferiti nei registri;
4. Execution (EX): gli operandi vengono processati dall'unità aritmetica;
5. Save (SV): il risultato viene trasferito dai registri alla memoria.

Una schematizzazione del ciclo delle istruzioni è mostrata in Fig. 1.4.
Quando i dati o le istruzioni vengono prelevati dalla memoria durante le fasi di IF e OF tutte le unità funzionali del processore rimangono inutilizzate. Anche alla fine, quando il risultato viene trasferito in memoria, le unità funzionali rimangono inattive.

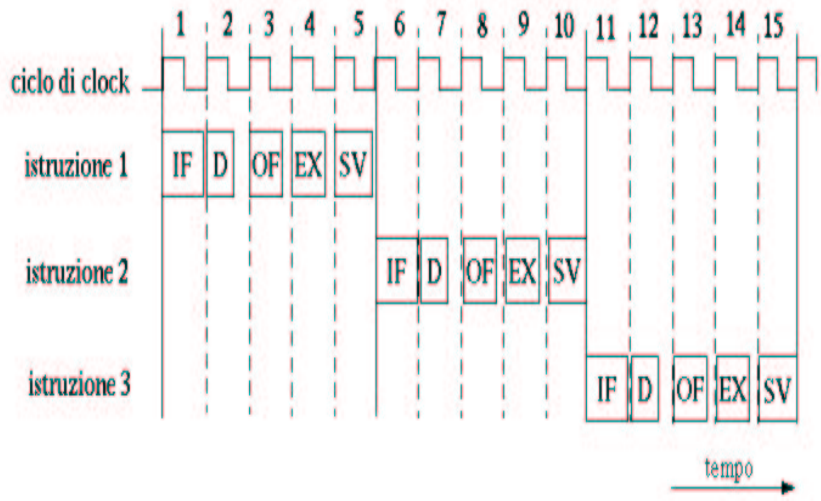

Figura 1.4: *Sequenza delle fasi per l'esecuzione di una istruzione.*

Per migliorare l'utilizzo delle risorse è possibile eseguire le varie fasi del ciclo in pipeline. Esse possono essere eseguite indipendentemente nello stesso tempo, allora l'esecuzione di istruzioni consecutive può essere sovrapposta. Tale procedimento è simile alla catena di montaggio nelle industrie. Nella Fig. 1.5 viene mostrato come vengono eseguite concorrentemente le diverse fasi.

L'esecuzione concatenata delle istruzioni non riduce il numero di cicli di clock necessario per l'esecuzione di una istruzione: il tempo di esecuzione di una istruzione rimane lo stesso. Tuttavia, il numero di cicli di clock necessario per completare un set di istruzioni successive viene ridotto di un fattore corrispondente al numero di stadi (lunghezza) della pipeline.

△

♣ Esempio 8

Esaminiamo la tipologia di un processore RISC (**R**educed **I**struction **S**et **C**compuer). L'esecuzione sequenziale delle istruzioni come mostrato in Fig. 1.4 richiede cinque cicli di clock per completare le istruzioni. Nel caso di una pipeline a cinque stadi, come

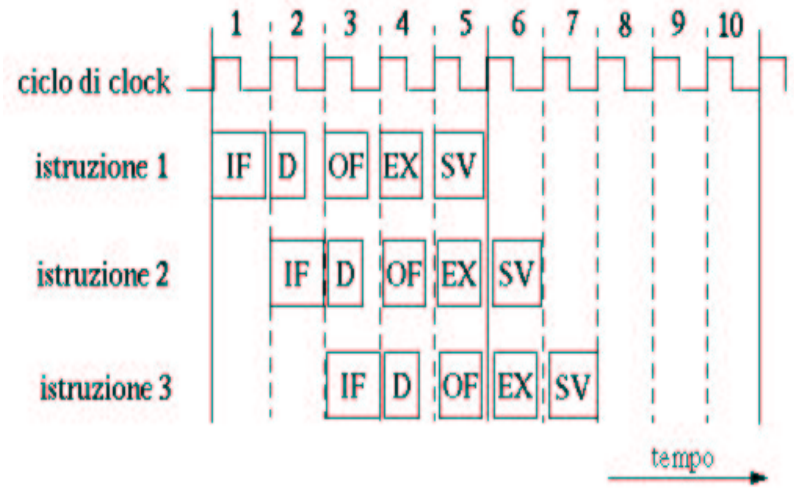

Figura 1.5: *Esecuzione contemporanea delle diverse fasi del ciclo sulle istruzioni su di un pipeline a cinque stadi.*

mostrato in Fig. 1.5, l'intervallo minimo per completare istruzioni consecutive è un ciclo di clock. La riduzione ad 1/5 del tempo necessario a completare la sequenza di istruzioni in sequenziale è uguale alla lunghezza della pipeline.

△

Il tempo che trascorre tra l'inizio dell'operazione e l'apparire del primo risultato è detto *tempo di latenza* di una pipeline. Nell'esempio citato è $4t_u$. In generale esso dipende dal numero di segmenti da cui è composta la struttura e dal tempo necessario per eseguire ogni compito[3] . Una struttura a catena con un tempo di latenza elevato è efficiente solo quando la sequenza di dati su cui opera è

[3]È possibile costruire pipeline anche per altri tipi di operazioni di base come la moltiplicazione e piú in generale concatenare due operazioni diverse. L'architettura RISC 6000/590 ha una unità pipeline a 2 passi, uno per la moltiplicazione e l'altro per l'addizione. Ad esempio, dovendo calcolare $a(i) = b(i) * c(i) + d(i)$ *con* $i = 1, \ldots, 4$, al passo $i = 1$ il primo segmento calcola $b(1) * c(1) = e(1)$; ai passi $i = 2, 3, 4$ il primo segmento calcola $b(i) * c(i) = e(i)$, mentre il secondo segmento calcola $e(i-1) + d(i-1)$ (il secondo segmento inizierà, quindi, a lavorare al secondo passo); all'ultimo passo, $i = 5$, solo il secondo segmento calcola $e(4) + d(4)$.

sufficientemente lunga, solo in tal caso infatti, il termine $(N-1)t_u$ prevale rispetto a $4t_u$, che, quindi, può considerarsi trascurabile.

Un esempio di calcolatore nel quale il pipeline ha trovato un'applicazione efficiente è fornito dai calcolatori vettoriali. Tali calcolatori consentono l'esecuzione di operazioni su vettori mediante l'utilizzo di opportune componenti hardware come ad esempio quelle predisposte all'indirizzamento di dati strutturati a vettore (registri vettoriali).

Tutte le soluzioni menzionate - strutture pipelined e vettoriali, calcolatori VLIW [4] - sono utili per incrementare la velocità e l'efficienza di una singola unità di elaborazione.

Ritorniamo all'analogia con il lavoro degli operai. Nella costruzione di una casa il lavoro viene suddiviso in modo opportuno tra gli operai in base alle competenze e alla necessità di ridurre i tempi di costruzione. L'idea piú naturale allora è quella di **decomporre un problema di dimensione N in P sottoproblemi di dimensione N/P e risolverli contemporaneamente su piú calcolatori (Fig. 1.6)**, riorganizzando opportunamente le componenti del calcolatore.

Si possono ad esempio inserire nella CPU piú unità adibite alle operazioni aritmetiche (ALU) o connettere due o piú processori (CPU) aumentando il numero di istruzioni eseguite nell'unità di tempo o collegare in rete piú calcolatori (processore, memoria, I/O) che eseguono la stessa applicazione.

Questa semplice idea è alla base della metodologia del **Calcolo Parallelo**.

[4] VLIW: *V*ery *L*ong *I*nstruction *W*ord. Il principio di funzionamento delle architetture VLIW, la cui struttura è pipelined, si basa sulla specifica di un certo numero di istruzioni, che costituiscono piú *Word*, caricate ed eseguite contemporaneamente dal processore.

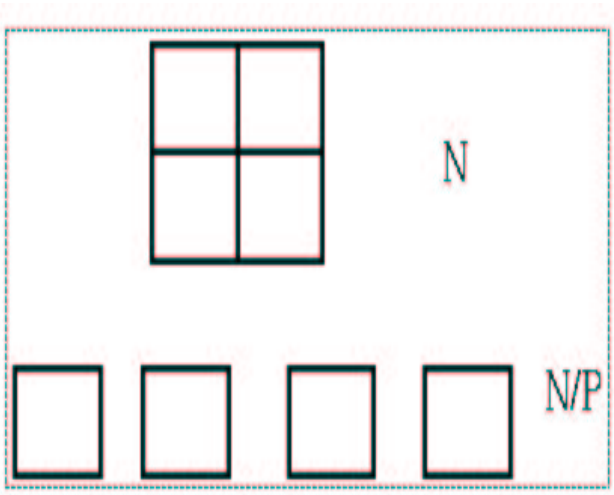

Figura 1.6: *Un problema di dimensione N può essere suddiviso in P sottoproblemi di dimensione N/P da risolvere contemporaneamente.*

Nel paragrafo successivo consideriamo un semplice problema, il calcolo della somma di N numeri, e analizziamo come si affronta il progetto di un algoritmo parallelo che tenga conto dell'architettura che caratterizza l'ambiente di calcolo nel quale l'algoritmo verrà implementato. Introduciamo, per semplicità, due tipi di calcolatori paralleli: quelli che **hanno diverse CPU che condividono un'unica memoria** (sistemi a memoria condivisa (*shared*)) (Fig. 1.7) **e quelli che sono costituiti da piú processori ognuno con una propria memoria** (sistemi a memoria distribuita (*distributed*)) (Fig. 1.8).

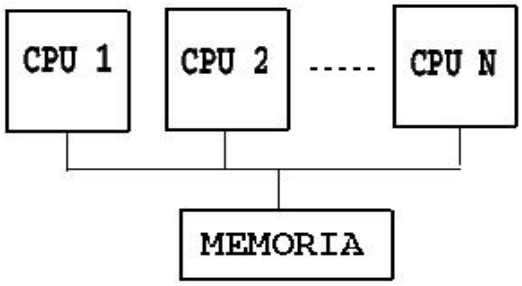

Figura 1.7: *Sistema a memoria condivisa.*

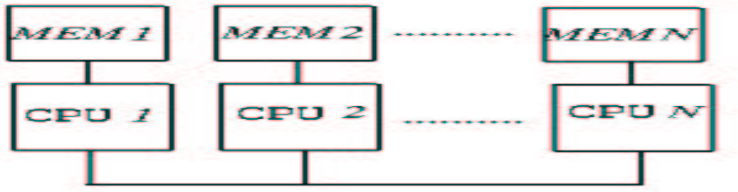

Figura 1.8: *Sistema a memoria distribuita.*

1.1 L'operazione di somma

Problema
Calcolo della somma S di $N = 16$ numeri:

$$a_0, a_1, \ldots, a_{15}.$$

Con un calcolatore monoprocessore la somma di N numeri è calcolata eseguendo $N-1$ addizioni una per volta secondo un ordine prestabilito, cioè, ad esempio, seguendo l'ordine naturale, si ha:

$$S = a_0 + a_1 + ... + a_{15}.$$

Vediamo come può essere modificato l'algoritmo se eseguito da un calcolatore costituito da piú CPU che condividono un'unica memoria (memoria condivisa); lo schema funzionale di un siffatto calcolatore è schematizzato in Fig. 1.7.

Supponiamo di disporre di quattro processori che indicheremo nel seguito con P_i, $i = 0, 1, 2, 3$. L'idea piú naturale è quella di suddividere la somma in somme parziali ed assegnare il calcolo di ciascuna somma parziale ad un processore.

Ogni processore può quindi effettuare la somma parziale ad esso assegnata. Inoltre, tale operazione può essere eseguita da ciascun processore *indipendentemente* dagli altri e *concorrentemente* agli altri. Ad esempio, come schematizzato nella Fig. 1.9, il processore P_0 calcola:

$$s_0 := a_0 + a_1 + a_2 + a_3 \ ;$$

il processore P_1 calcola:

$$s_1 := a_4 + a_5 + a_6 + a_7 \ ;$$

$$
\begin{array}{ll}
P_0 \rightarrow & s_0 := a_0 + a_1 + a_2 + a_3 \\
P_1 \rightarrow & s_1 := a_4 + a_5 + a_6 + a_7 \\
P_2 \rightarrow & s_2 := a_8 + a_9 + a_{10} + a_{11} \\
P_3 \rightarrow & s_3 := a_{12} + a_{13} + a_{14} + a_{15}
\end{array}
$$

Figura 1.9: *Schema per il calcolo delle somme parziali da parte di ciascun processore.*

il processore P_2 calcola:

$$s_2 := a_8 + a_9 + a_{10} + a_{11} \ ;$$

ed infine il processore P_3 calcola:

$$s_3 := a_{12} + a_{13} + a_{14} + a_{15} \ .$$

In memoria saranno quindi presenti le somme parziali $s_0, s_1, s_2, \ ed \ s_3$. Si pone ora il problema di calcolare la somma totale. Supponiamo che in memoria sia stata definita la variabile $SUMTOT$ inizializzata a zero, che dovrà contenere la somma totale. Le strategie per calcolare la somma totale sono molteplici: un solo processore si puó occupare di fare la somma o piú processori possono coordinarsi per calcolare la somma totale.
Nel primo caso, ad esempio, il solo processore P_0 accede in memoria alle somme parziali s_i, per $i = 0, \ldots, 3$, e calcola:

$$SUMTOT := SUMTOT + s_i \ \ per \ \ i = 0, \ldots, 3$$

Nel secondo caso, invece, nasce il problema della **sincronizzazione**. Affinché il valore di $SUMTOT$ sia correttamente aggiornato è necessario che ciascun processore abbia accesso esclusivo e in maniera coordinata a tale variabile durante il suo aggiornamento, cioè è necessario *sincronizzare* gli accessi in memoria.

$$\begin{array}{ll} P_1 \rightarrow & SUMTOT := SUMTOT + s_1 \\ P_3 \rightarrow & SUMTOT := SUMTOT + s_3 \end{array}$$

Figura 1.10: *Operazioni eseguite contemporaneamente da P_1 e P_3*

Cosa può succedere se non c'è sincronizzazione negli accessi in memoria?
Può ad esempio capitare che entrambi i processori P_1 e P_3 leggano il valore di $SUMTOT$ in memoria e aggiornino tale variabile (Figura 1.10), il processore P_1 calcolando:

$$SUMTOT := SUMTOT + s_1$$

e il processore P_3 calcolando:

$$SUMTOT := SUMTOT + s_3$$

Evidentemente $SUMTOT$ contiene l'aggiornamento effettuato dal processore che per ultimo accede in scrittura alla locazione di memoria di $SUMTOT$, ad esempio $SUMTOT$ non conterrà il contributo dovuto a s_1 se P_3 accede in scrittura a $SUMTOT$ dopo P_1 poichè l'aggiornamento di $SUMTOT$ effettuato da P_3 utilizza come valore precedente quello d'inizializzazione e cioè zero.

Una tecnica per sincronizzare l'accesso in memoria fa uso dei *Semafori*, contatori utilizzati per controllare l'accesso alle risorse condivise. Sono usati come meccanismi di blocco delle variabili per evitare che altri processori accedano alla memoria condivisa contemporaneamente al processore che la sta già utilizzando. In questo caso, un modo per evitare aggiornamenti sbagliati della variabile SUMTOT è stabilire che, per $i = 0, 1, 2, 3$, il processore P_i, e solo lui, acceda alla memoria e calcoli:

$$SUMTOT := SUMTOT + s_i$$

Al quarto aggiornamento la variabile $SUMTOT$ conterrà la somma totale.
In definitiva, l'algoritmo parallelo prevede che tutti i processori calcolino la somma parziale e addizionino tale valore alla variabile $SUMTOT$ che contiene infine la somma totale.

L'algoritmo riportato nella *Procedura* 1.1 apparentemente sembra, per la maggior parte delle operazioni, simile ad uno sequenziale. Fanno eccezione le istruzioni *forall*, *lock* e *unlock*.

La direttiva *forall* indica che le operazioni che seguono devono essere eseguite da tutti i processori, la direttiva *lock* sincronizza gli accessi in memoria: fa sì che solo un processore per volta abbia accesso alla variabile SUMTOT. La direttiva *unlock* fa terminare la sincronizzazione degli accessi in memoria.

```
SOMMA DI N=kp NUMERI SU UNA MACCHINA
MIMD - Shared Memory

begin
    forall P_i , 0 ≤ i ≤ p − 1 do
        sumtot := 0
        sum := 0
        h := i ∗ (n/p)
        for j=h to h+(n/p)-1 do
            sum := sum + a_j
        endfor
        lock (sumtot)
            sumtot := sumtot + sum
        unlock (sumtot)
    endforall
end
```

Procedura 1.1 - Algoritmo per il calcolo della somma di N numeri.

$$
\begin{array}{ll}
P_0 \rightarrow & s_0 := a_0 + a_1 + a_2 + a_3 \\
P_1 \rightarrow & s_1 := a_4 + a_5 + a_6 + a_7 \\
P_2 \rightarrow & s_2 := a_8 + a_9 + a_{10} + a_{11} \\
P_3 \rightarrow & s_3 := a_{12} + a_{13} + a_{14} + a_{15}
\end{array}
$$

Figura 1.11: *Schema per il calcolo delle somme parziali in un sistema costituito da piú CPU.*

Esaminiamo ora l'esecuzione dello stesso problema su un calcolatore costituito da piú CPU ognuna dotata di una propria memoria (Fig. 1.8) (memorie distribuite).

Supponiamo che ogni processore P_i, con $i = 0, \ldots, 3$, abbia, nella propria memoria, i valori con cui calcolare la corrispondente somma parziale s_i. Quindi (Figura 1.11), come prima, il processore P_0 calcola:

$$s_0 := a_0 + a_1 + a_2 + a_3 \ ;$$

il processore P_1 calcola:

$$s_1 := a_4 + a_5 + a_6 + a_7 \ ;$$

il processore P_2 calcola:

$$s_2 := a_8 + a_9 + a_{10} + a_{11} \ ;$$

ed infine il processore P_3 calcola:

$$s_3 := a_{12} + a_{13} + a_{14} + a_{15} \ .$$

In questo caso ogni processore possiede nella propria memoria i quattro numeri da sommare e la corrispondente somma parziale s_i.

Come calcolare la somma totale? Le strategie possibili sono diverse.

I strategia:
ogni processore calcola la sua somma parziale e invia tale valore ad un processore prestabilito, ad esempio il processore P_0, che conterrà la somma finale.

Al I passo[5] il processore P_1 *"invia"* al processore P_0 la sua somma parziale s_1 e il processore P_0 calcola:

$$s_0 := s_0 + s_1$$

Al II passo il processore P_2 *"invia"* al processore P_0 la sua somma parziale s_2 e il processore P_0 calcola:

$$s_0 := s_0 + s_2$$

Al III passo il processore P_3 *"invia"* al processore P_0 la sua somma parziale s_3 e il processore P_0 calcola:

$$s_0 := s_0 + s_3$$

In generale al k-esimo passo, $k = 1, 2, 3$, il processore il cui identificativo coincide con il passo corrente, ovvero il processore P_i, con $i \equiv k$, *"invia"* al processore P_0 la propria somma parziale s_i e il processore P_0 calcola:

$$s_{0,i} := s_{0,i-1} + s_i$$

con $s_{0,0} = s_0$. Alla fine, dopo 3 passi, P_0 possiede la somma totale .

Uno schema delle comunicazioni tra i processori è mostrato in Fig. 1.12 , mentre un primo schema dell'algoritmo è riportato nella *Procedura* 1.2 .

[5]Di seguito denoteremo con il termine *"passo"* il *"passo temporale"*, il quale è scandito da unità di tempo successive.

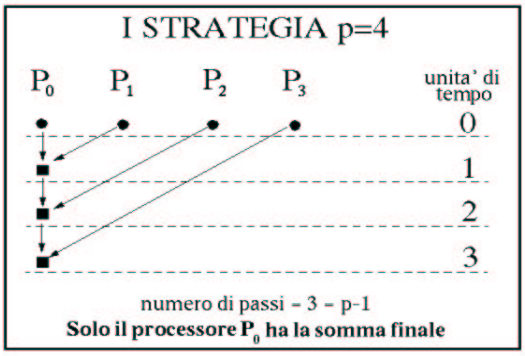

Figura 1.12: *Schema delle comunicazioni nella prima strategia.*

ALGORITMO 1

SOMMA DI N=kp NUMERI - I STRATEGIA

```
begin
    forall P_i 0 ≤ i ≤ p − 1 do
        h := i ∗ (n/p)
        s_i := 0
        for j=h to h+(n/p)-1 do
            s_i := s_i + a_j
        endfor
        if P_0 then
            s_{0,0} := s_0
            for k = 1 to p − 1 do
                recv(s_k, P_k)
                s_{0,k} := s_{0,k−1} + s_k
            endfor
        else if P_i then
            send(s_i, P_0)
        endif
    endforall
end
```

Procedura 1.2 - Algoritmo per la somma di N numeri - I Strategia.

La differenza tra questo algoritmo ed uno sequenziale è nelle due istruzioni di *invia* (send) e *ricevi* (recv). L'istruzione $send(s_i,\ P_i)$ implica l'invio al processore P_i della variabile s_i, mentre l'istruzione $recv(s_i,\ P_i)$ implica la ricezione della variabile s_i dal processore P_i.

II strategia:
ogni processore calcola la sua somma parziale s_i, poi, coppie distinte di processori comunicano tra loro le somme calcolate. In ogni coppia un processore invia all'altro la sua somma parziale. Il risultato finale è in un unico processore prestabilito.

Al I passo P_1 invia s_1 a P_0 e P_0 calcola:

$$s_{0,1} := s_0 + s_1$$

P_3 invia s_3 a P_2 e P_2 calcola:

$$s_{2,3} := s_2 + s_3$$

Al II passo P_2 invia $s_{2,3}$ a P_0 e P_0 calcola:

$$s_{0123} := s_{0,1} + s_{2,3}$$

Alla fine del II passo P_0 possiede la somma totale.

Se è pur vero che la II strategia sia piú efficace della I perché i passi temporali sono dimezzati e in ciascun passo coppie distinte di processori operano concorrentemente, un'attenta analisi dell'algoritmo 2 (*Procedura* 1.3) mostra una delle caratteristiche tipiche degli algoritmi strutturati ad albero, ovvero lo **sbilanciamento del carico di lavoro** dei processori.

ALGORITMO 2

SOMMA DI N=kp NUMERI - II STRATEGIA

```
begin
    forall P_i 0 ≤ i ≤ p − 1 do
        h := i ∗ (n/p)
        s_i := 0
        for j = 1 to log_2 p do
            s_i := s_i + a_j
        endfor
        for k = 1 to log_2 p do
            q := 2^(k−1)
            forall P_i i = q to p − q step 2^k do
                send(s_i, P_(i−q))
            endforall
            forall P_i i = 0 to p − q step 2^k do
                recv(s_(i+q), P_(i+q))
                s_i := s_i + s_(i+q)
            endforall
        endfor
    endforall
end
```

Procedura 1.3 - Algoritmo per la somma di N numeri - II Strategia

Nella Fig. 1.13 possiamo osservare, infatti, che al secondo passo i processori P_1 e P_3 rimangono inattivi e che alla fine dell'algoritmo solo un processore, P_0, è attivo.

III strategia:
ogni processore calcola la sua somma parziale, poi, ad ogni passo successivo, coppie distinte di processori comunicano contemporaneamente. In ogni coppia i processori si scambiano le proprie somme parziali. Il risultato finale è in tutti i processori.

Al I passo P_0 e P_1 si scambiano i valori delle proprie somme

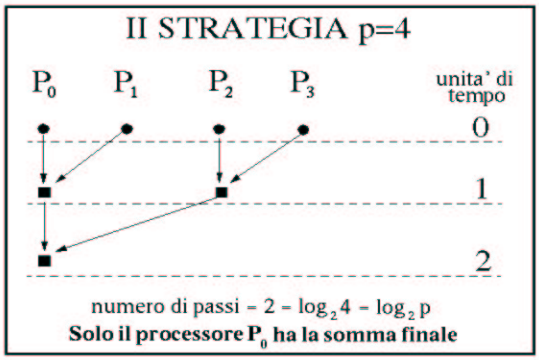

Figura 1.13: *Schema delle comunicazioni nella seconda strategia.*

parziali ed entrambi calcolano:

$$s_{0,1} := s_0 + s_1$$

Analogamente fanno P_2 e P_3 con le rispettive somme parziali.

Al II passo P_0 e P_2 si scambiano i valori delle somme parziali $s_{0,1}$ e $s_{2,3}$ ed entrambi calcolano la somma totale:

$$s_{0123} := s_{0,1} + s_{2,3}$$

Analogamente fanno P_1 e P_3. Alla fine, tutti i processori hanno la somma totale.
In Fig. 1.14 viene riportato lo schema delle comunicazioni tra i processori mentre, nella *Procedura* 1.4, viene descritto l'algoritmo per la somma di N numeri utilizzando la terza strategia.

La III strategia è, praticamente, identica alla II, l'unica differenza risiede nel fatto che in questa strategia le coppie di processori si scambiano i valori calcolati dalle somma parziali in maniera tale che, alla fine, tutti i processori abbiano il risultato finale.

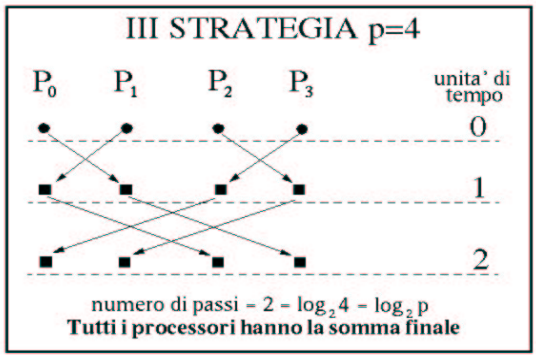

Figura 1.14: *Schema delle comunicazioni nella terza strategia.*

ALGORITMO 3

SOMMA DI N=kp NUMERI - III STRATEGIA

```
begin
    forall P_i 0 ≤ i ≤ p − 1 do
        h := i ∗ (n/p)
        s_i := 0
        for j = 1 to log2p do
            s_i := s_i + a_j
        endfor
        for k = 1 to log2p do
            resto := i − (i/2^k) × 2^k
            if (resto < 2^(k−1)) then
                send(s_i, P_(i+2^(k−1)))
                recv (s_(i+2^(k−1)), P_(i+2^(k−1)))
                s_i := s_i + s_(i+2^(k−1))
            else
                send(s_i, P_(i−2^(k−1)))
                recv(s_(i−2^(k−1)), P_(i−2^(k−1)))
                s_i := s_i + s_(i−2^(k−1))
            endif
        endforall
    endfor
end
```

Procedura 1.4 - Algoritmo per la somma di N numeri - III Strategia

In tutte e tre le strategie ogni processore calcola la propria somma parziale e invia tale valore agli altri processori in modo da ottenere la somma totale. Osserviamo che, alla fine della prima e della seconda strategia, solo il processore P_0 ha il risultato finale, mentre, alla fine della terza strategia lo hanno tutti i processori.

Inoltre, per ottenere il risultato finale nella prima strategia si effettuano $p-1$ passi, avendo indicato con p il numero di processori, in ciascuno dei passi si esegue 1 operazione, mentre la seconda e la terza impiegano $log_2 p$ passi. Nella seconda strategia, inoltre, ad ogni passo, vengono eseguite $p-k-1$ operazioni, avendo indicato con k il generico passo, mentre nella terza strategia in ciascun passo vengono eseguite p operazioni.
Nella Tab. 1.1 sono riassunti il numero di passi, le operazioni ad ogni generico passo k ed il numero totale di operazioni per ciascuna strategia.

	# passi	# operazioni concorrenti eseguite al passo k	# operazioni totali
I Strategia	$p-1$	1	$p-1$
II Strategia	$log_2 p$	$p-k-1$	$(p-2)+\ldots+(p-log_2 p-1)$
III Strategia	$log_2 p$	p	$p \cdot log_2 p$

Tabella 1.1: Per ciascuna delle tre strategie descritte sono indicati il numero di passi, il numero di operazioni al passo k e totali.

Volendo scegliere una delle tre strategie, basandoci sul numero di passi, siamo indotti a preferire la seconda o la terza strategia, essendo $log_2 p < p-1$, mentre, considerando le operazioni totali, la scelta cade sulla prima strategia.

In effetti, la valutazione basata soltanto sul numero di operazioni concorrenti, eseguite complessivamente da un algoritmo in ambiente parallelo, non è adeguata perché, come vedremo nel capitolo 4, ciò che influenza le prestazioni di un algoritmo in ambiente parallelo

è il numero di *"passi temporali"*, non il numero di operazioni eseguite. Qundi, è giusto optare per la seconda o la terza strategia. In altre parole, poiché in ciascun passo temporale, piú operazioni sono eseguite concorrentemente, ovvero nello stesso tempo, una corretta valutazione dell'algoritmo della somma in ciascuna delle tre strategie, ad ogni passo temporale deve tener conto del tempo di esecuzione di una sola operazione, come illustrato nella Tab. 1.2.

	# passi	# operazioni concorrenti eseguite al passo k	# operazioni concorrenti
I Strategia	$p-1$	1	$p-1$
II Strategia	$log_2 p$	$p-k-1$	$log_2 p$
III Strategia	$log_2 p$	p	$log_2 p$

Tabella 1.2: Per ciascuna delle tre strategie descritte sono indicati il numero di passi, il numero di operazioni al passo k e totali eseguite in paralello.

È ora piú evidente che la scelta tra le strategie cada sulla seconda o la terza.

Una valutazione completa, in ambiente di calcolo parallelo, non può però prescindere dal numero di comunicazioni, essendo gli algoritmi costituiti da operazioni e comunicazioni. Nella prima strategia viene effettuata una comunicazione ad ogni passo, si hanno cosí $p-1$ comunicazioni totali. Nella seconda e terza strategia, invece, ad ogni passo k si hanno piú comunicazioni, in particolare $p/2^k$ nella seconda strategia e p nella terza. Esse però avvengono nello stesso passo temporale, quindi sia la seconda sia la terza strategia richiedono un numero totale di comunicazioni uguale a $log_2 p$.

Lo schema riassuntivo del numero di passi e delle comunicazioni effettuate nelle tre strategie analizzate è illustrato nella Tab. 1.3.

	# passi	# comunicazioni al passo k	# comunicazioni
I Strategia	$p-1$	1	$p-1$
II Strategia	$log_2 p$	$p/2^k$	$log_2 p$
III Strategia	$log_2 p$	p	$log_2 p$

Tabella 1.3: Per ciascuna delle tre strategie descritte sono indicati il numero di passi, il numero di comunicazioni al passo k e totali eseguite concorrentemente.

La II e la III strategia sono, pertanto, quelle che richiedono il minor numero di comunicazioni e di operazioni.

Nell'algoritmo della somma si è evidenziato un primo aspetto fondamentale nella progettazione di un algoritmo per un calcolatore parallelo: la gestione **della concorrenza dell'esecuzione delle operazioni**.

Utilizzando un calcolatore multiprocessore a **memoria condivisa**, attraverso un *"meccanismo di sincronizzazione degli accessi in memoria"* si controlla la concorrenza delle operazioni. Nel caso, invece, di un calcolatore multiprocessore a **memoria distribuita**, la concorrenza delle operazioni viene gestita attraverso esplicite operazioni di *"scambio di messaggi"*, che consentono il controllo e la sincronizzazione delle operazioni.