

Università degli Studi di Perugia

Corso di Laurea Triennale in Informatica

Corso:

SISTEMI DISTRIBUITI E PARALLELI

PARTE 2

Anno Accademico:

2022/2023

Docente:

SERGIO TASSO



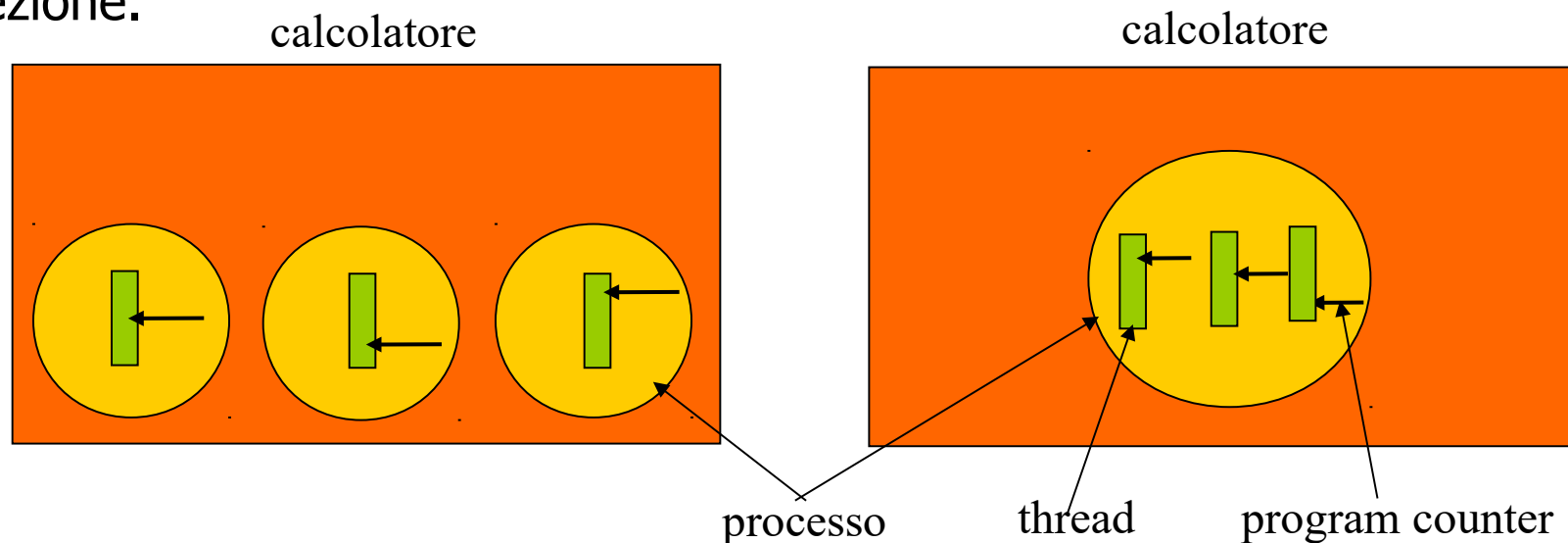
sergio.tasso@unipg.it

Processi e Thread

- ❑ I **thread** (fili) di controllo di un processo sono istanze del processo stesso che condividono lo stesso spazio di indirizzamento condividendo le stesse variabili globali. I thread vengono anche chiamati **lightweight processes** (processi a peso leggero).
- ❑ Ciascun processo ha un proprio contatore di programma, un proprio stack, un proprio insieme di registri e un proprio spazio di indirizzamento.
- ❑ I processi possono comunicare tramite messaggi, semafori di mutua esclusione e monitor.
- ❑ Ciascun thread gira in modo strettamente sequenziale e ha un proprio contatore di programma e uno stack per mantenere traccia di dove è arrivato.
- ❑ I thread condividono la stessa CPU, come fanno i processi, e su multiprocessori possono girare in parallelo. I thread possono creare thread figli e si possono bloccare in attesa che venga completata una chiamata.

Processi e Thread

- Poiché ciascun thread può accedere a qualunque indirizzo virtuale, un thread può leggere, scrivere o anche cancellare completamente lo stack di un altro thread.
- Ma al contrario dei processi, thread multipli creati dallo stesso processo appartengono ad un unico utente e si presume che non ci sia bisogno di protezione.



Tre processi con un thread ciascuno

Sistemi distribuiti e paralleli

Un processo con tre thread

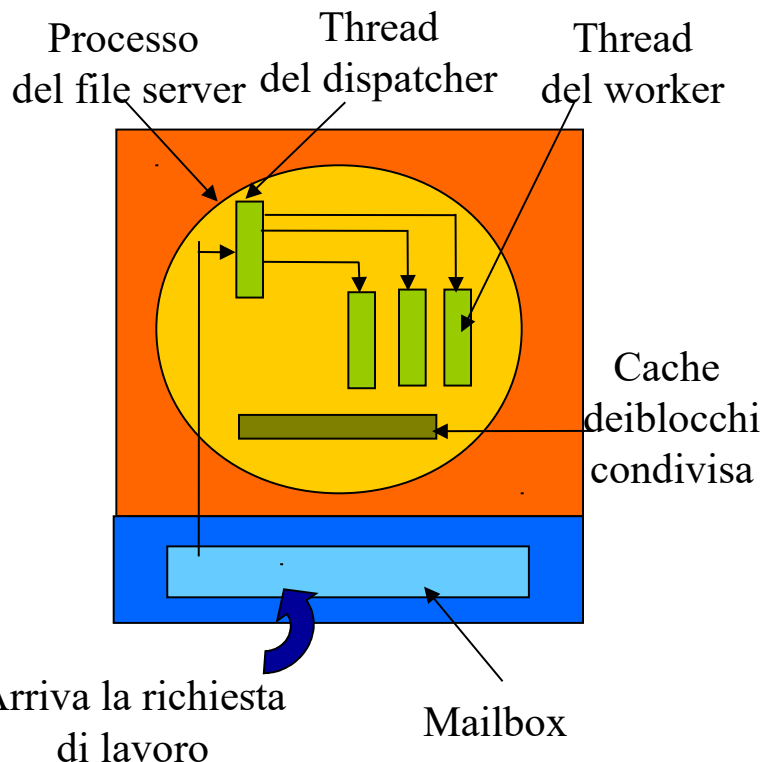
Processi e Thread

- Oltre che condividere lo spazio di indirizzamento tutti i thread condividono lo stesso insieme di file aperti, i processi figli, i timer, i segnali, ecc.

Elementi per ogni thread	Elementi per ogni processo
Program counter Stack Insieme dei registri Thread figli Stato	Spazio degli indirizzi Variabili globali File aperti Processi figli Timer Segnali Semafori Informazioni di accredito

Uso dei thread

- Supponiamo per es. di gestire un file server che occasionalmente deve bloccarsi in attesa del disco. L'organizzazione potrebbe essere così rappresentata:



Modello dispatcher/worker

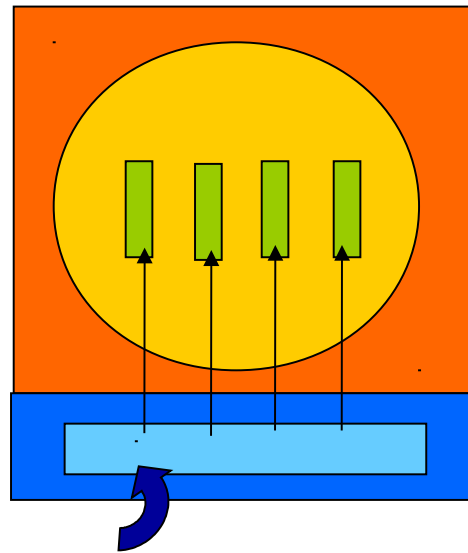
Un thread detto **dispatcher** (distributore) legge le richieste in arrivo da una mailbox di sistema, dopo aver esaminato le richieste sceglie un **worker** thread inattivo (bloccato) e gli passa una richiesta, scrivendo il puntatore al messaggio in una prefissata locazione associata a ciascun thread.

Il dispatcher risveglia poi il worker.

Quando il thread si sveglia, controlla se la richiesta può essere soddisfatta usando la memoria cache condivisa. Se no, spedisce un messaggio al disco per ottenere il blocco desiderato (se l'operazione è di READ) e si blocca in attesa del completamento dell'operazione.

Uso dei thread

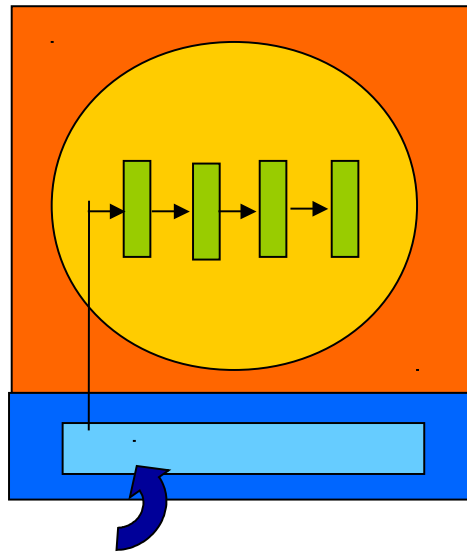
Un altro modello per gestire lo stesso server è quello a **team** (squadra). In cui tutti i thread sono uguali e ciascuno ottiene e processa i propri lavori. A volte per lavori che un thread non è in grado di gestire si mantiene una coda di lavori pendenti, in tal caso un thread deve controllare la coda dei lavori prima di accedere alla mailbox di sistema.



Modello a team

Uso dei thread

I thread possono anche essere organizzati con il modello **pipeline** in cui il primo thread genera alcuni dati e li passa al prossimo thread per essere processati e così via. Nel caso del file server non risulta adatto, ma è una buona scelta nel problema dei produttori-consumatori.



Modello pipeline

Uso dei thread

- Esistono tre modi per costruire un server:

Modello

Caratteristiche

Thread

Parallelismo, chiamate di sistema bloccanti

Processo con thread singolo

Nessun parallelismo, chiamate di sistema bloccanti

Macchina a stati finiti

Parallelismo, chiamate di sistema non bloccanti

Package di thread

- ❑ Un insieme di primitive relative ai thread disponibile per l'utente viene detto **thread package**.
- ❑ Relativamente alla gestione dei thread si possono avere due tipi di thread: thread statici e thread dinamici.
- ❑ I **thread statici** sono così chiamati perché il loro numero è previsto in fase di stesura del programma o al momento della sua compilazione; a ciascuno di essi viene assegnato uno stack fisso, questo è un approccio semplice ma non flessibile.
- ❑ I **thread dinamici** vengono creati e distrutti a tempo di esecuzione. La chiamata per la creazione specifica il programma principale del thread, la dimensione dello stack e altri parametri tra cui la priorità di schedulazione. La chiamata restituisce un identificatore da usare nelle chiamate successive relative al thread.

Package di thread

- ❑ Un thread può terminare in due modi diversi:
 - tramite una exit : una volta che il thread ha finito il suo lavoro
 - essere ucciso dall'esterno.
- ❑ Poiché i thread usano uno spazio di memoria comune ad altri thread, l'accesso alle strutture dati comuni viene programmato usando le regioni critiche gestite da semafori, monitor e altro.
- ❑ Una tecnica comunemente usata nei package di thread è quella dei mutex. Il **mutex** è un semaforo semplificato che ha due soli stati: chiuso (locked) o aperto (unlocked). Le operazioni sui mutex sono: la **LOCK** che tenta di chiudere il mutex, terminando con successo se il mutex era aperto, se invece era già chiuso il thread viene bloccato; la **UNLOCK** apre un mutex sbloccando uno e uno solo dei processi in attesa sul mutex.

Package di thread

- Una ulteriore operazione che di solito viene messa a disposizione è la **TRYLOCK** che tenta di chiudere un mutex: se il mutex è aperto restituisce un codice di successo, se è chiuso la TRYLOCK non blocca il thread, ma restituisce un codice di errore.
- Un'altra caratteristica nei package di thread è la **variabile di condizione**. A ciascuna variabile di condizione viene associato un mutex al momento della sua creazione. La differenza tra i mutex e le variabili di condizione è che i mutex vengono usati per bloccarsi a breve termine, per es. per controllare l'ingresso in una regione critica, mentre le variabili condizione vengono usate per attendere più a lungo, per es. per gestire le attese della disponibilità di una specifica risorsa.

Package di thread

- ❑ La situazione di utilizzo può essere così riassunta:
 - Un thread chiude un mutex per entrare nella regione critica, poi una volta dentro esamina le tabelle di sistema e si accorge che una risorsa di cui ha bisogno è occupata. Se usasse un secondo mutex per la risorsa, il primo mutex già chiuso impedirebbe al thread relativo di liberare la risorsa. Se al contrario liberasse il primo mutex, altri thread potrebbero entrare nella regione critica intasandola.

- ❑ La soluzione che prevede le variabili di condizione è la seguente:

Fai un lock del mutex;
 controlla struttura dati;
 while (risorsa occupata)
 wait (variabile condizione)
 marca la risorsa come occupata;
fai un unlock del mutex;

acquisizione della risorsa

Fai un lock del mutex;
 marca la risorsa come libera;
fai un unlock del mutex;
wakeup(variabile condizione)

rilascio della risorsa

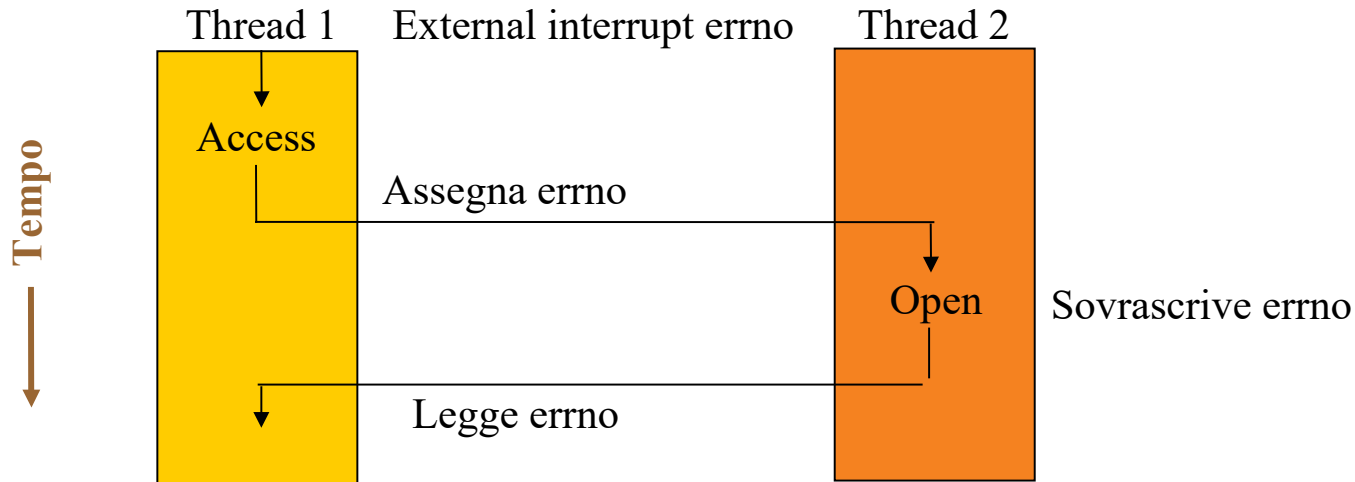
Aspettare (wait) sulla variabile di condizione è definito come effettuare l'attesa e aprire il mutex in maniera atomica.

La wakeup viene usata per risvegliare uno dei thread bloccati in attesa della risorsa.

Package di thread

- Le variabili globali ad un thread possono creare problemi, per es.:

In Unix quando un processo o thread fallisce una chiamata di sistema, il codice di errore viene messo nella variabile `errno`.



Se dopo un fallimento su una `Access` del thread 1 viene impostato `errno`, ma subito dopo la CPU viene tolta al thread 1 a favore del thread 2 e questo a sua volta fallisce una `Open` e sovrascrive il valore di `errno`, quando poi il thread 1 riavrà il controllo per agire sul fallimento, andrà ad operare con un valore non corretto.

Package di thread

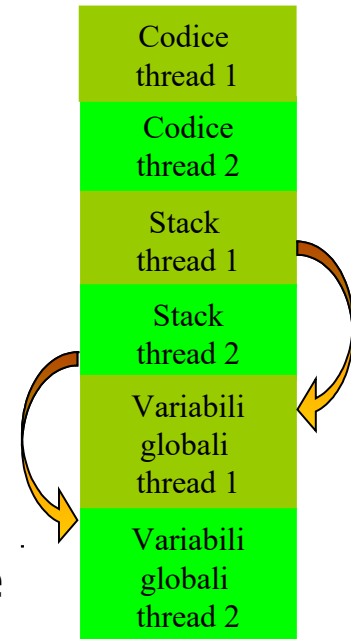
Le possibili soluzioni possono essere:

- Eliminare le variabili globali (ma Unix non funziona così)
- Assegnare a ciascun thread le proprie variabili globali effettuando una copia locale. L'accesso alle variabili globali private è un po' macchinoso essendoci già nei linguaggi variabili locali e globali ma non intermedie, è però possibile allocare una porzione di memoria per le variabili globali e passarla a ciascuna procedura come parametro extra. Soluzione poco elegante, ma funziona.
- Introdurre nuove funzioni di libreria per creare, assegnare e leggere queste variabili globali rispetto al thread:

create_global("bufprt"); che assegna un'area di memoria speciale riservata per la chiamata dei thread, solo il thread che ha eseguito la chiamata ha accesso alla variabile globale,

set_global("bufprt",&buf); che memorizza il valore del puntatore nella locazione di memoria creata dalla *create_global* (scrittura),

bufprt = read_global("bufprt"); che restituisce l'indirizzo memorizzato nella variabile globale (lettura)

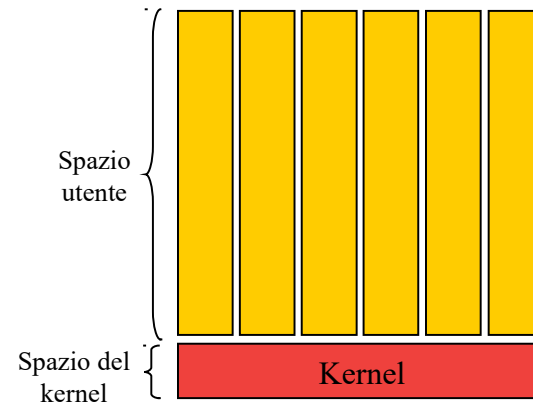
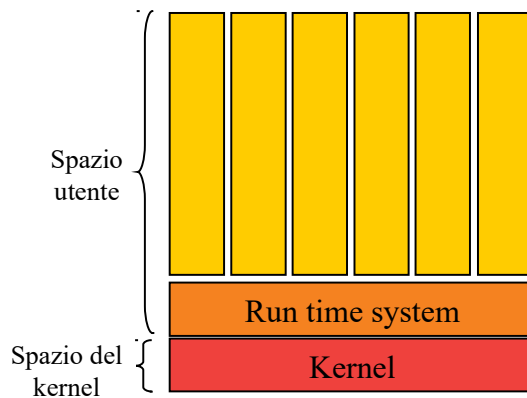


Implementazione di package di thread

- ❑ Ci sono due modi per implementare un package di thread:
 - internamente allo spazio di indirizzamento dell'utente
 - nel kernel.
- ❑ Nel primo metodo c'è il vantaggio che il package dei thread può essere implementato anche su quei sistemi operativi che non supportano i thread (Unix). I thread girano sopra un run time system, cioè sopra un insieme di procedure che si occupano della gestione dei thread. Quando un thread esegue una qualsiasi funzione che prevede la sua sospensione, chiama una procedura del supporto e questa controlla se deve essere sospeso, se sì, memorizza i propri registri in una tabella, cerca un thread non bloccato da mandare in esecuzione e carica i registri di macchina con i valori salvati relativi al nuovo thread. Appena lo stack pointer o il program counter cambiano, il nuovo thread ritorna in vita. Lo scambio dei registri tra thread è di un ordine più veloce rispetto ad eseguire una trap al kernel. E questo è un altro vantaggio del primo metodo.

Implementazione di package di thread

- Nel secondo caso, cioè se il kernel sa dei thread e li gestisce, non occorre il run time system, per ogni processo il kernel ha una tabella con un elemento per ciascun thread, che contiene i registri del thread, lo stato, la priorità e altro. Tutte le chiamate sono a livello di sistema, con un costo più alto.



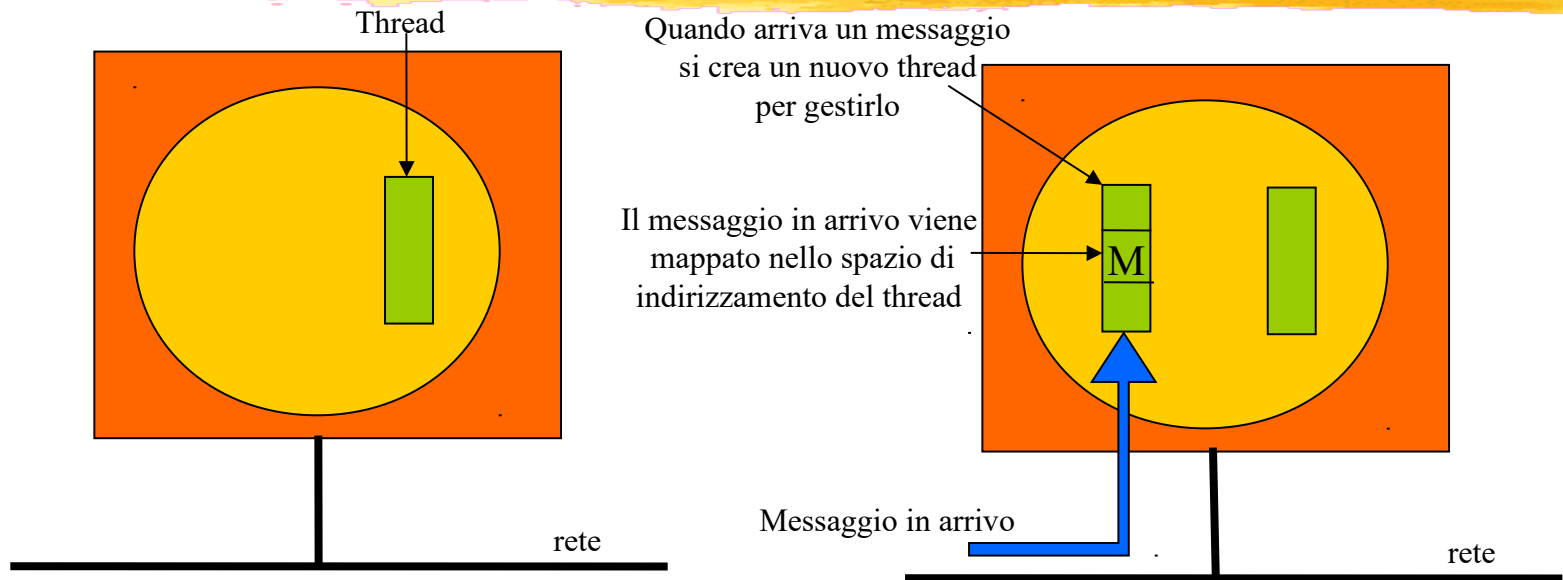
Implementazione di package di thread

- Un grosso problema del primo approccio è quello relativo all'implementazione delle chiamate di sistema bloccanti. Se per es. un thread tenta di leggere da un pipe vuoto o fa qualcosa d'altro che lo blocca, nel caso dell'implementazione sul kernel il thread esegue una trap al kernel che lo blocca e manda in esecuzione un altro thread, nel caso invece dell'implementazione nello spazio utente non può effettuare la chiamata perché bloccherebbe tutti i thread. A tale scopo bisogna poter prevedere questa anomalia con del codice introdotto intorno alla chiamata di sistema che esegue i vari controlli. Tale codice si chiama jacket (giacca).
- Un secondo svantaggio è che quando un thread di un processo va in esecuzione, nessun altro thread di quel processo andrà mai in esecuzione finché il primo thread non rilascerà la CPU. Nel caso a livello kernel è lo scheduler che genera le interruzioni.

Thread e RPC

- ❑ Nei sistemi operativi distribuiti vengono usati sia la RPC che i thread, in particolare i thread sono stati ideati come alternativa economica ai normali processi. Anche la RPC è stata realizzata a peso leggero.
- ❑ Un metodo per velocizzare la RPC è di creare uno stack condiviso dal thread client e dal thread server. In tal caso molte RPC anche se remote verranno trattate come se fossero locali.
- ❑ Un'altra tecnica si basa sull'osservazione che quando un thread del server si blocca in attesa di una nuova richiesta, non ha nel suo stack dati o informazioni di contesto interessanti, per cui, una volta che un thread ha terminato una richiesta, svanisce semplicemente. Quando arriva un nuovo messaggio il kernel crea un nuovo thread per servire la richiesta, mappa il messaggio nello spazio di indirizzamento del server e prepara un nuovo thread per accedere al messaggio. Questo schema viene chiamato **receive implicita** e il thread che è stato creato spontaneamente per trattare la RPC in arrivo viene detto **thread di pop-up**.

Thread e RPC



Thread del server

Creazione di un thread all'arrivo di un nuovo messaggio

- ❑ Questo metodo ha dei vantaggi importanti rispetto alle RPC convenzionali: i thread non si devono bloccare in attesa di un nuovo lavoro e quindi non c'è nessun contesto da salvare, inoltre creare un nuovo thread è più economico che resuscitarne uno vecchio (non c'è contesto da recuperare), infine non occorre copiare il messaggio di arrivo.

Un esempio di package di thread: il DCE

- ❑ Il **Distributed Computing Environment** della Open Software Foundation è un package di thread con un totale di 51 primitive (procedure di libreria) che hanno a che fare con i thread e che possono essere chiamate dagli utenti. Molte di queste chiamate non sono necessarie, ma sono convenienti.
- ❑ DCE è conforme allo standard **POSIX** (**P**ortable **O**perating **S**ystem **I**nterface for **U**nix), che consente la portabilità dei programmi. I thread POSIX sono comunemente noti come *pthreads*. Lo standard è supportato da molti sistemi operativi basati su UNIX.
- ❑ Possiamo raggruppare le chiamate in **sette categorie** in base al loro uso.
- ❑ La prima riguarda la gestione dei thread: le chiamate permettono di creare e di far terminare thread. Un thread padre può attendere un figlio usando la *join*, ma se invece il padre non ha interesse a ciò che fa o farà il figlio può chiamare la *detach* in tal modo quando il thread figlio termina la sua memoria viene rilasciata immediatamente, senza aspettare che il padre esegua la *join*.

Un esempio di package di thread: il DCE

Chiamata	Descrizione
Create	Creazione di un nuovo thread
Exit	Chiamata da un thread quando ha finito
Join	Come la chiamata di sistema WAIT di UNIX
Detach	Rende non necessario che un thread padre aspetti quando il chiamante termina

Un esempio di package di thread: il DCE

- Una seconda categoria permette di creare, distruggere e gestire **template** (schemi) per thread, mutex e variabili condizione e tali template possono essere predisposti a opportuni valori iniziali. Quando si crea un oggetto, uno dei parametri della chiamata per la creazione è un puntatore ad un template (per esempio si può creare un template di thread dando come attributo che abbia lo stack da 8k).
- Il vantaggio dell'avere i template sta nel fatto che eliminano la necessità di specificare come parametri separati tutte le opzioni, man mano che il package si evolve le chiamate possono essere identiche, basta aggiungere nuovi attributi ai template.

Un esempio di package di thread: il DCE

Chiamata	Descrizione
Attr_create	Crea un template per predisporre i parametri dei thread
Attr_delete	Cancella un template per i thread
Attr_setprio	Fissa la priorità di schedulazione di default nel template
Attr_getprio	Legge la priorità di schedulazione di default nel template
Attr_setstacksize	Fissa la dimensione di default dello stack nel template
Attr_getstacksize	Legge la dimensione di default dello stack nel template
Attr_mutexattr_create	Crea il template per i parametri mutex
Attr_mutexattr_delete	Cancella il template per i mutex
Attr_mutexattr_setkind_np	Fissa il tipo di default del mutex nel template
Attr_mutexattr_getkind_np	Legge il tipo di default del mutex nel template
Attr_condattr_create	Crea il template per i parametri delle variabili condizione
Attr_condattr_delete	Cancella il template per le variabili condizione

Un esempio di package di thread: il DCE

- Un terzo gruppo riguarda i mutex che possono essere creati e distrutti dinamicamente.

Chiamata	Descrizione
Mutex_init	Crea un mutex
Mutex_destroy	Cancella un mutex
Mutex_lock	Cerca di fare un lock su un mutex; aspetta se è già in locked
Mutex_trylock	Cerca di fare un lock su un mutex; fallisci se era già in locked
Mutex_unlock	Fai un unlock su un mutex

Un esempio di package di thread: il DCE

- Un quarto gruppo riguarda le variabili condizione che possono essere creati e distrutti dinamicamente. I thread si possono sospendere sulle variabili condizione in attesa della disponibilità delle risorse richieste e sono disponibili due operazioni di sveglia: una dis segnalazione, che sveglia esattamente un thread in attesa, ed una di broadcast che li sveglia tutti quanti.

Chiamata	Descrizione
Cond_init	Crea una variabile condizione
Cond_destroy	Cancella una variabile condizione
Cond_wait	Attende su una condizione di wait fino a quando non arriva un segnale o un broadcast
Cond_signal	Sveglia almeno un thread in attesa su una variabile condizione
Cond_broadcast	Sveglia tutti i thread in attesa su una variabile condizione

Un esempio di package di thread: il DCE

- Un quinto gruppo riporta tre chiamate per manipolare le variabili globali ad un thread

Chiamata	Descrizione
keycreate	Crea una variabile globale per questo thread
setspecific	Assegna il valore di un puntatore alla variabile globale per thread
getspecific	Leggi il valore di un puntatore alla variabile globale per thread

Un esempio di package di thread: il DCE

- Un sesto gruppo riguarda l'uccisione dei thread o la protezione alla uccisione da parte di altri thread

Chiamata	Descrizione
Cancel	Cerca di uccidere un altro processo
Setcancel	Abilita o disabilita la possibilità per un altro thread di uccidere questo thread

Un esempio di package di thread: il DCE

- Il settimo ed ultimo gruppo permette ai thread di un processo di essere schedulati in accordo ad una politica FIFO, round robin, con o senza prelazione o secondo altri algoritmi, si possono assegnare le priorità e l'algoritmo di schedulazione.

Chiamata	Descrizione
Setscheduler	Fissa l'algoritmo di scheduling
Getscheduler	Legge l'algoritmo di scheduling corrente
Setprio	Fissa la priorità di schedulazione
Getprio	legge la priorità di schedulazione

Modelli di sistema

- ❑ Il **modello Workstation** consiste di workstation (personal computer di fascia alta) distribuite all'interno di un edificio o di un campus interconnesse per mezzo di una LAN. Alcune workstation hanno dischi locali (**workstation diskly** o diskfull) altre non ne hanno (**workstation diskless**).
- ❑ Le workstation diskless fanno riferimento ad un file server per le richieste di operazioni di lettura e scrittura dei file. Tali stazioni hanno i seguenti vantaggi:
 - costi minori
 - manutenzione software facilitata
 - meno rumorosità (i dischi hanno ventole)
 - simmetria (ogni stazione è paritetica alle altre)
 - flessibilità
- ❑ Le workstation con dischi locali non sono uguali fra loro né allineate e il loro utilizzo può variare a seconda del loro grado di dipendenza dal file server.
- ❑ Le varie situazioni possono essere così schematizzate:

Modelli di sistema

Dipendenza dal file server ↑

Uso del disco	Vantaggi	Svantaggi
Diskless	Basso costo, facile manutenzione hw e sw, simmetria e flessibilità	Uso pesante della rete; i file server diventano colli di bottiglia
Paginazione e file temporanei	Carico della rete ridotto rispetto al caso diskless	Costo più alto per via dell'alto numero di dischi necessari
Paginazione e file temporanei e binari	Ulteriore riduzione del carico della rete	Costo più alto: complessità addizionale per l'esecuzione degli aggiornamenti dei file binari
Paginazione e file temporanei, binari e caching dei file	Carico della rete ancora più basso; riduzione anche del carico del file server	Costo più alto; problemi di consistenza della cache
File system locale completo	Praticamente nessun carico sulla rete; eliminata la necessità del file server	Perdita di trasparenza

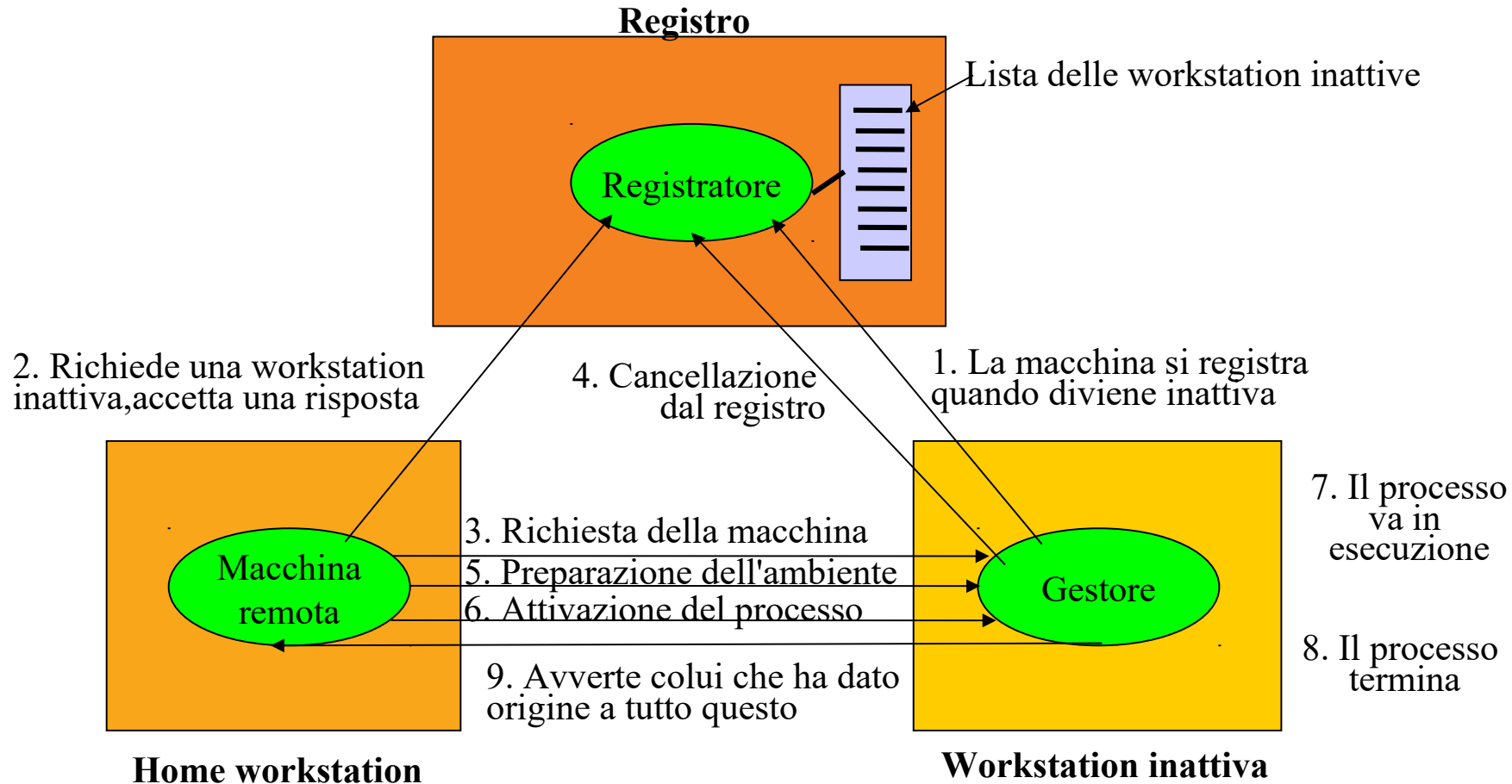
Utilizzo delle workstation inattive

- Un tentativo di utilizzo delle workstation inattive all'interno di una LAN viene dato dal comando UNIX

rsh macchina comando

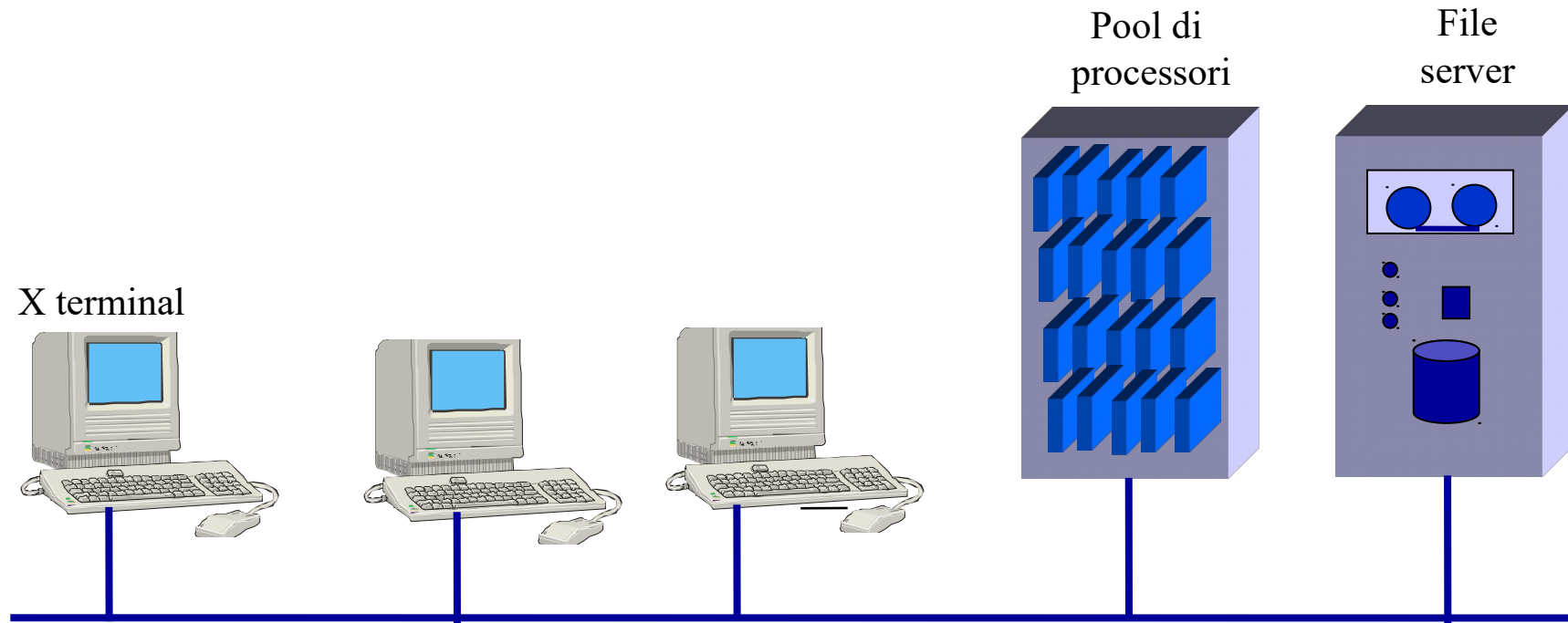
- Ma come si fa ad individuare le workstation inattive? E come ricreare l'ambiente uguale a quello locale?
- Ci sono diversi metodi per gestire tali workstation, uno basato sull'uso di un registro di workstation inattive può essere così schematizzato:

Utilizzo delle workstation inattive



Modelli di sistema

- Il **modello a pool di processori** (insieme coordinato di processori) consiste di un rack (mobiletto) pieno di CPU che possono essere dinamicamente allocate agli utenti su richiesta.

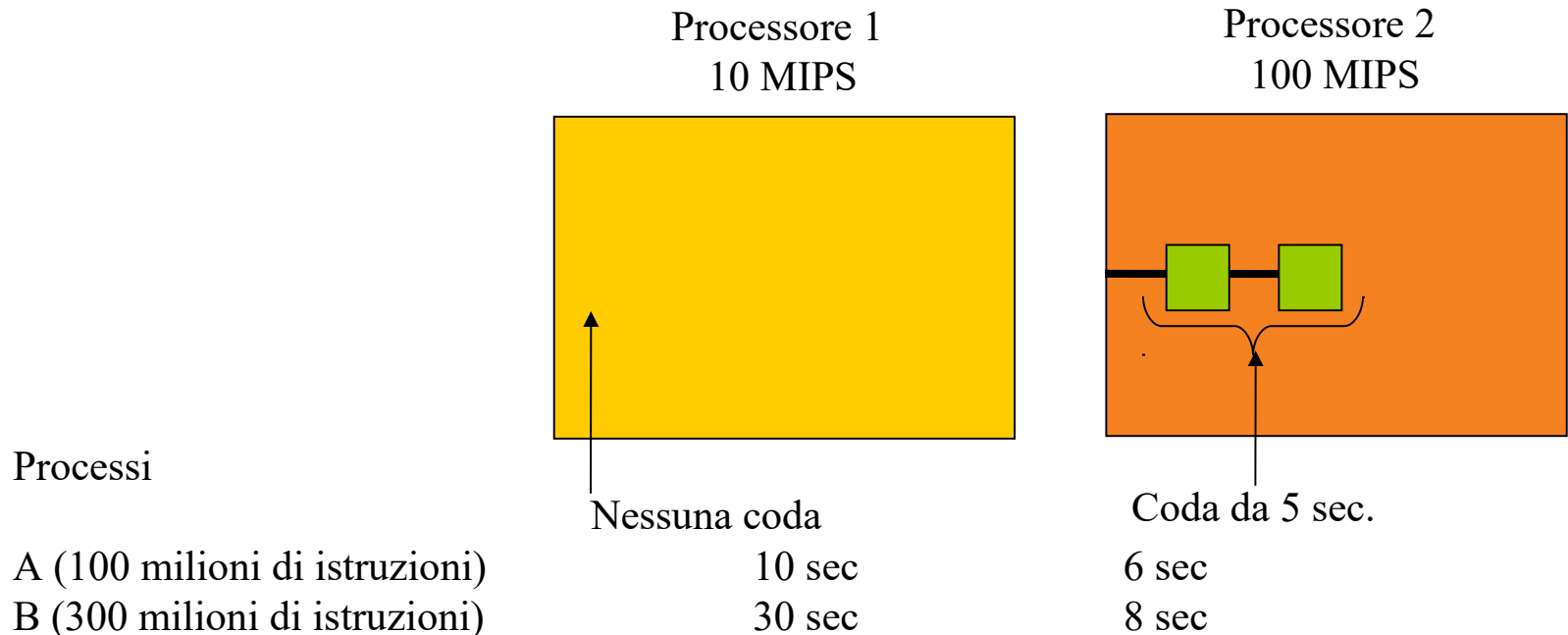


Allocazione dei processori

- ❑ Le strategie di allocazione dei processori si possono dividere in due classi:
 - **Non-migratoria** (o statica): quando si crea un processo si decide dove allocarlo e, una volta allocato ad una macchina, vi resta fino alla terminazione.
 - **Migratoria** (o dinamica): quando il processo si può spostare anche se è già andato in esecuzione.
- ❑ Le strategie migratorie pur essendo più efficienti nel bilanciamento del carico sono più complesse e pesanti nella progettazione di un sistema.
- ❑ L'obiettivo primario resta **l'ottimizzazione della CPU**, cioè massimizzare l'utilizzo della CPU a favore dei lavori utente. La CPU non deve mai restare inattiva.
- ❑ Un altro obiettivo è minimizzare il **tempo di risposta** o anche la minimizzazione del **rapporto di risposta** definito come il tempo necessario a far eseguire un processo su una qualche macchina diviso per il tempo necessario a far girare lo stesso processo su una macchina campione completamente scarica.

Allocazione dei processori

Esempio: Consideriamo i due processi e i due processori seguenti:



Possibili assegnamenti:

Processore 1	Processore 2	Tempo medio di risposta	Rapporto di risp.1	Rapporto di risp. 2	Rapporto medio di risposta
A	B	$(10+8)/2=9$ sec	$10/10=1$	$8/3=2,6$	$(1+2,6)/2=1,8$
B	A	$(30+6)/2=18$ sec	$30/30=1$	$6/1=6$	$(1+6)/2=3,5$
AB		$(10+30+5)/2=22,5$ sec	$40/10=4$ (A) $40/30=1,3$ (B)	$5/2,5=2$	$(4+1,3+2,5)/3=2,6$
	AB	$(6+8)/2=7$ sec		$14/1=14$ (A) $14/3=4,66$ (B)	$(14+4,66)/2=9,33$

Sistemi distribuiti e paralleli

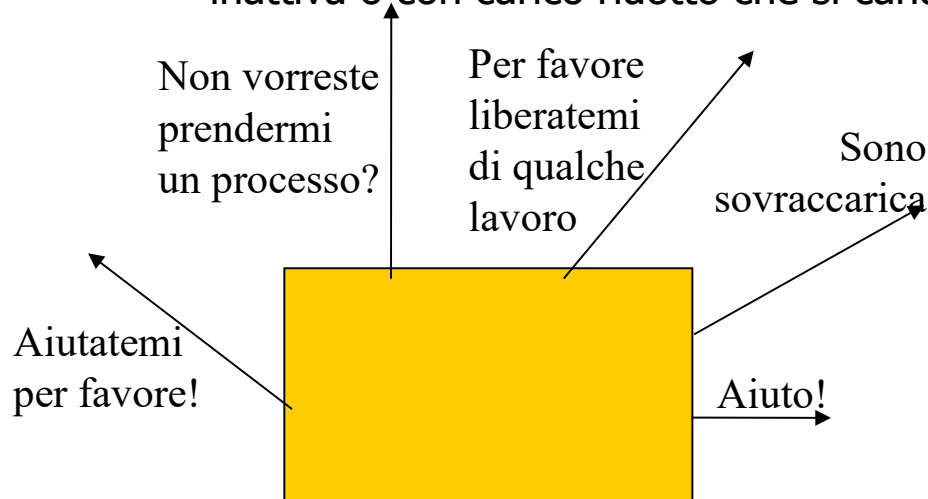
Algoritmi di allocazione dei processori

- Gli algoritmi per l'allocazione dei processori possono essere di vari tipi:
 - **deterministici** o **non deterministici**
 - ✧ deterministici per i processi il cui comportamento è noto a priori: in tal caso si può trovare l'assegnamento perfetto; non deterministici altrimenti: in tal caso l'assegnamento si attua con tecniche ad hoc dette **euristiche**.
 - **distribuiti** o **centralizzati**
 - ✧ Per i centralizzati occorre raccogliere tutte le informazioni del sistema in un posto, ciò permette di prendere le decisioni migliori, ma sovraccarica la macchina. I distribuiti sono da preferire.
 - **ottimi** o **sub-ottimi**
 - ✧ Le ottime sono più costose in termini di gestione, quindi sono da preferire le sub-ottime
 - **locali** o **globali**
 - ✧ Quando un processo è creato su una macchina sovraccarica dovrebbe essere inviato tramite una **politica di trasferimento** ad un'altra macchina. L'algoritmo di decisione (sia per riconoscere una macchina sovraccarica sia per decidere su quale altra macchina mandare il processo) può essere fatto in locale (cioè sulla macchina stessa e quindi più semplice ma meno preciso) o in globale (cioè esternamente più preciso ma a costi più elevati)

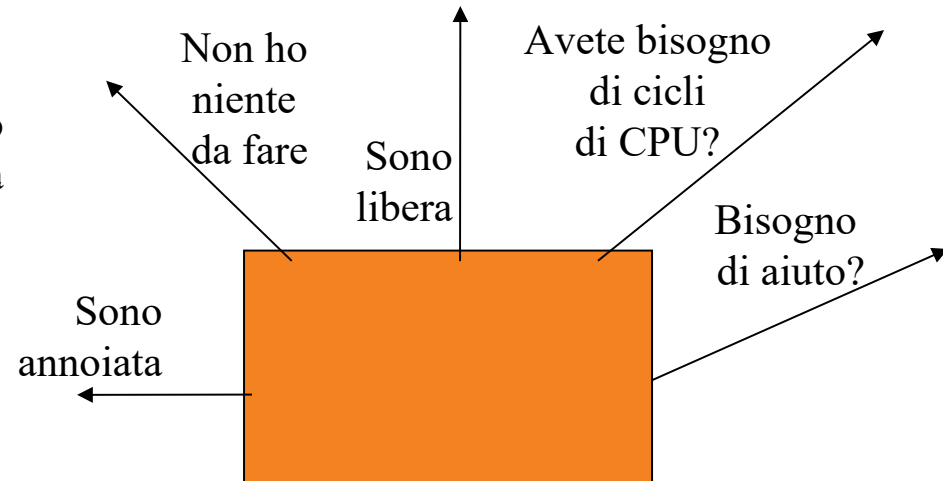
Algoritmi di allocazione dei processori

○ il cui inizio viene dato dal mittente o dal ricevente

- ✧ mentre la politica del trasferimento decide di disfarsi di un processo, la **politica del posizionamento** decide a chi spedirlo. Tale politica non può essere locale, poiché ha bisogno di informazioni globali sul carico per poter prendere una buona decisione. Queste informazioni possono essere sparse in due modi: o è il mittente che inizia lo scambio o è il ricevente che prende l'iniziativa. In figura sotto nel primo caso è il mittente che cerca di localizzare cicli di CPU, mentre nel secondo caso è una macchina inattiva o con carico ridotto che si candida come ricevente per fare qualcosa.



Una mittente cerca una macchina inattiva



Un ricevente cerca lavoro

Algoritmi di allocazione dei processori

- A livello pratico come si misura il carico?
 - ✧ **Numero dei processi allocati:** poco significativo perché comprensivo anche dei processi di sistema, dei servizi, ed altro presenti anche su macchine inattive.
 - ✧ **Percentuale di utilizzo della CPU in una frazione di tempo:** più significativa, ma a volte però anche questa stima è falsata da parti di codice critiche che disabilitano le interruzioni anche da timer, dato che il timer non scatterà finché il kernel non sarà entrato in un ciclo vuoto si tende a sottostimare l'effettivo utilizzo della CPU.
- Quanto è l'overhead dell'allocazione?
 - ✧ Ritardi per la raccolta delle informazioni, la comunicazione e lo spostamento.

Algoritmi di allocazione dei processori

○ Quanto è la complessità degli algoritmi?

- ✧ Eager ad altri hanno studiato tre algoritmi: in tutti e tre i casi ogni macchina nel sistema misura il proprio carico e decide da sola se sia sovraccarica o meno. Quando viene creato un nuovo processo la macchina controlla se è sovraccarica se lo è prende l'iniziativa e cerca una macchina meno carica.
- ✧ Algoritmo 1: prende una macchina a caso e vi spedisce il processo, se anche la seconda macchina è sovraccarica sceglierà a sua volta una terza macchina e così via finché non si raggiunge qualcuno che accetta il processo o si supera un prefissato contatore di passi oltre il quale non è permesso un ulteriore cambio.
- ✧ Algoritmo 2: sceglie una macchina a caso e le manda un messaggio di sondaggio, se la macchina non è sovraccarica le invia il processo, altrimenti questo resta sulla macchina che l'ha creato.
- ✧ Algoritmo 3: esegue un sondaggio fra k macchine per determinare i loro carichi e invia il processo a quella meno carica.
- ✧ Siccome il guadagno tra il 3 e il 2 è esiguo a scapito della complessità dell'algoritmo viene spesso scelto il 2.

Algoritmi di allocazione dei processori



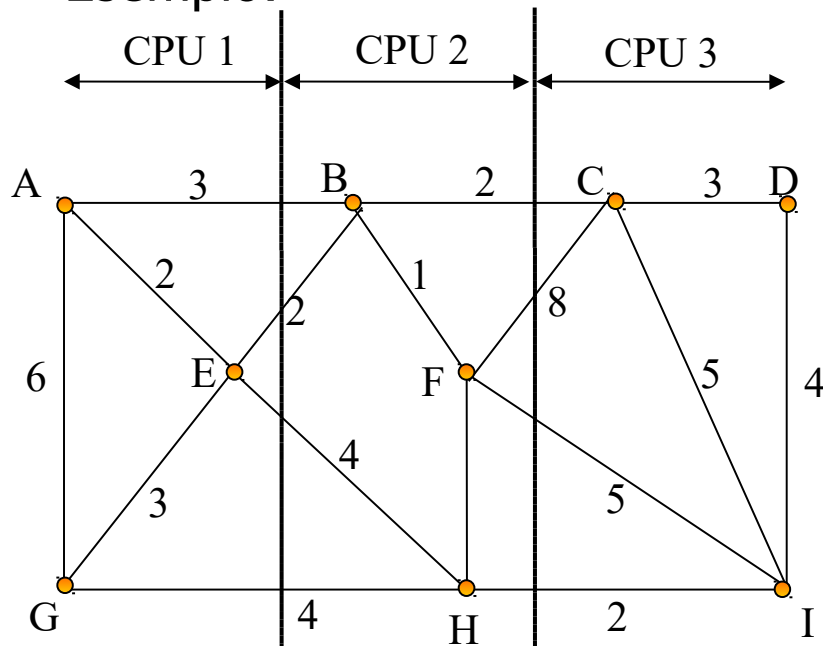
- Quanto sono stabili le valutazioni?
 - ✧ E' possibile avere situazioni in cui un processo viene rimbalzato tra due macchine perché l'una pensa di essere più sovraccarica dell'altra, e in particolare la valutazione viene fatta sempre quando un nuovo processo è nella macchina per cui il sistema non risulta mai in equilibrio.

Algoritmo di allocazione deterministico basato sulla teoria dei grafi

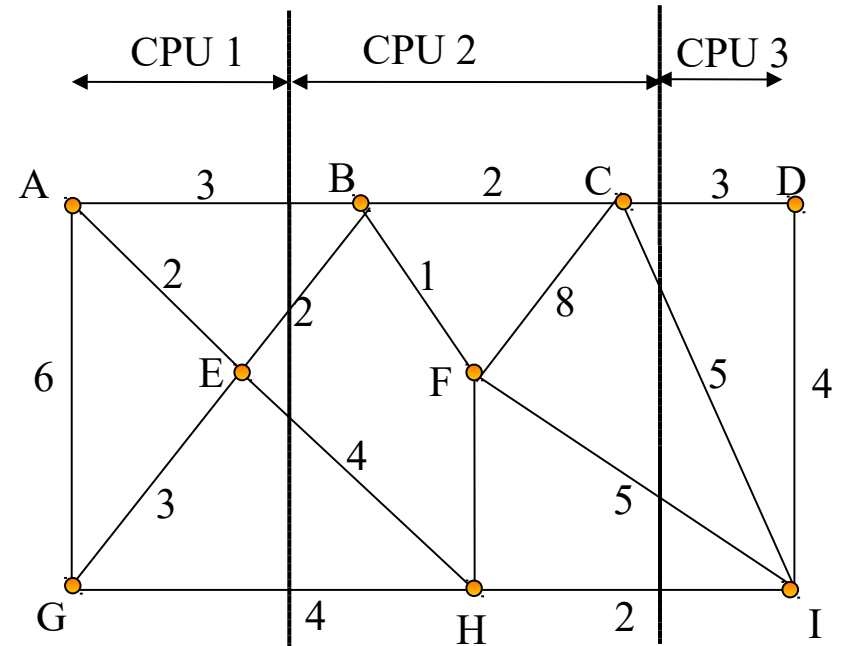
- In sistemi in cui sono noti i requisiti di CPU e memoria per ogni processo ed è nota la matrice che dà il traffico medio fra ogni coppia di processi, se il numero dei processori (k) è minore del numero dei processi, il sistema può essere rappresentato come un grafo pesato, con nodi che rappresentano i processi e gli archi che rappresentano il flusso dei messaggi fra i due processi. Matematicamente il problema si riduce a trovare un modo di partizionare (tagliare) il grafo in k sottografi disgiunti, soggetti a certi vincoli (per es. che i requisiti totali di memoria e CPU per ogni sottografo siano al di sotto di un certo limite). Per ciascuna soluzione che soddisfa i vincoli, gli archi che stanno completamente all'interno di un sottografo sono comunicazioni interne alla macchina e vanno ignorati, mentre quelli tra un sottografo e l'altro rappresentano il traffico sulla rete. L'obiettivo è trovare il partizionamento che minimizza il traffico sulla rete pur soddisfacendo i vincoli.

Algoritmo di allocazione deterministico basato sulla teoria dei grafi

Esempio:



Traffico di rete di 30 unità



Traffico di rete di 28 unità

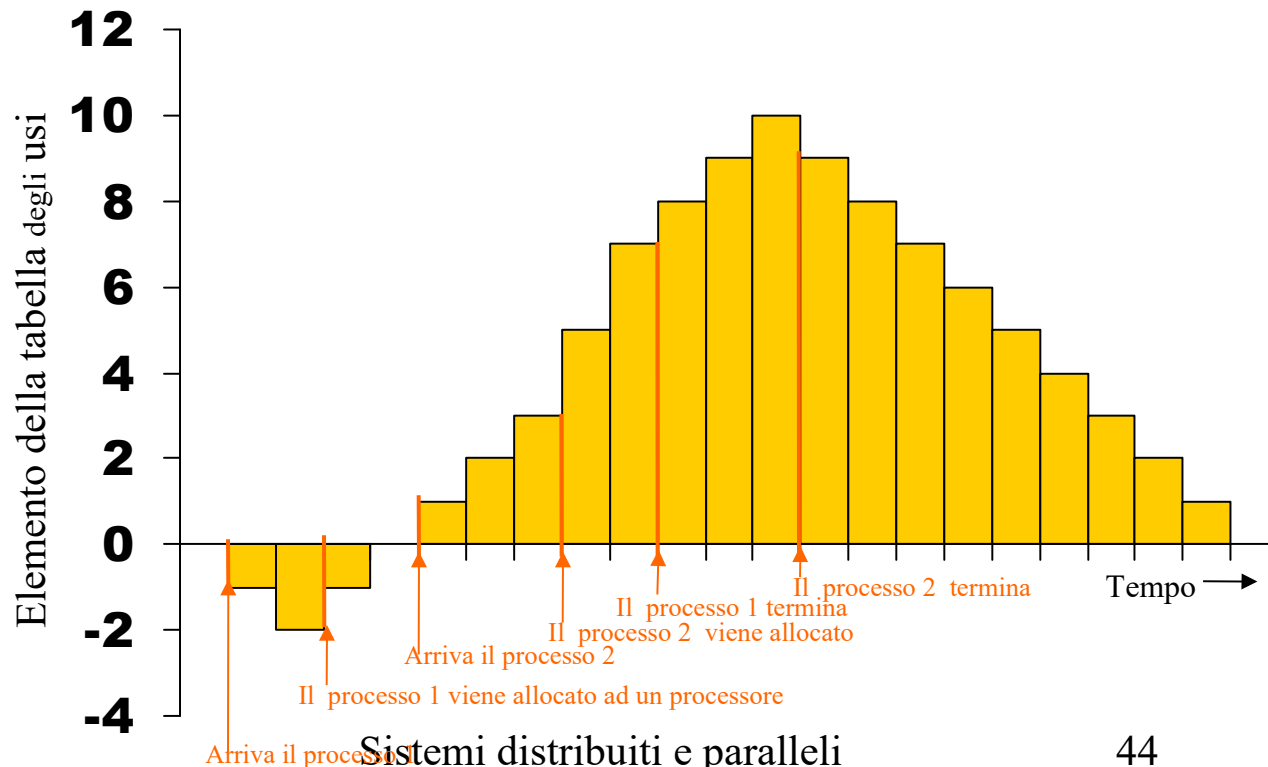
Supponendo che entrambi i partizionamenti soddisfino tutti i vincoli il secondo caso è la scelta migliore.

Algoritmo di allocazione centralizzato Up-down

- L'algoritmo **up-down** o su-e-giù (Mutka e Livny) non richiede informazioni preventive complete ed è centralizzato nel senso che un coordinatore mantiene una **tabella degli usi** con un elemento per ogni stazione di lavoro personale (per ogni utente) che inizialmente è posto a zero. Questo algoritmo è centralizzato non tanto per massimizzare l'uso della CPU, ma per mantenere una equa potenza di calcolo tra più utenti. Quando viene creato un processo che deve essere mandato in esecuzione da un'altra parte la macchina chiede al coordinatore di allocargli un processore, se non ce n'è disponibili la richiesta resta inevasa, ma viene annotata le seguente modo: quando un utente fa girare processi su macchine d'altri accumula punti di penalità (tot al secondo) e tali punti vengono sommati alla tabella degli usi. Quando invece ha richieste pendenti i punti vengono sottratti. Quando non si hanno né processi in esecuzione né richieste inevase il punteggio va a zero. In tal modo il punteggio va su e giù.

Algoritmo di allocazione centralizzato Up-down

- Gli elementi della tabella degli usi possono essere nulli, positivi o negativi: un valore positivo indica che l'utente usa risorse di rete, un valore negativo indica che l'utente necessita risorse di rete, un valore uguale a zero indica una situazione neutra.



Algoritmo di allocazione centralizzato Up-down



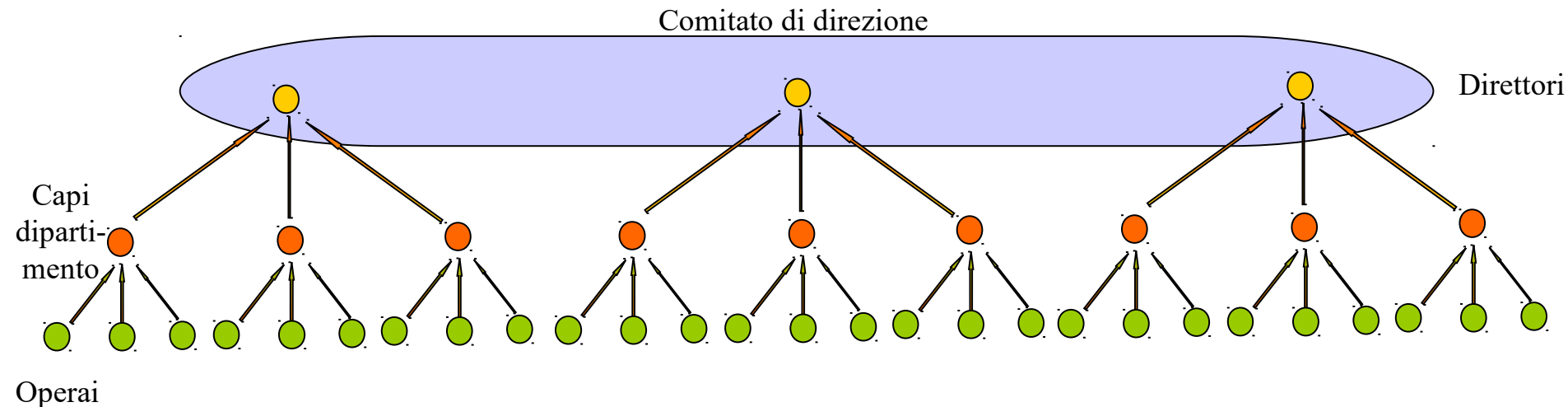
- Quando un processore si libera, vincono, fra le richieste pendenti, quelle dell'utente che ha il punteggio più basso. Un utente che non sta occupando processori e che ha una richiesta in attesa batterà sempre un utente che sta usando un alto numero di processori. L'algoritmo raggiunge quindi il suo obiettivo di allocare capacità in maniera equa.
- Studi di simulazione hanno dimostrato che l'algoritmo lavora come ci si aspetta sotto diverse condizioni di carico.
- Gli algoritmi centralizzati non sono però scalabili su sistemi a grosse dimensioni: il nodo centrale diventa presto un collo di bottiglia e hanno un singolo punto di fallimento.

Algoritmo di allocazione gerarchico

- Gli algoritmi di allocazione **gerarchici** prevedono una gerarchia organizzativa, così come quella delle aziende.
- Alcune macchine sono operai (worker), altre dirigenti (manager).
- Per ciascun gruppo di k worker una macchina manager (il capo dipartimento) tiene conto di chi è occupato e chi inattivo.
- Per un grande sistema ogni j capi di dipartimento si avrà un direttore che li controlla.
- E così via si possono avere "mega-direttori", ecc..
- I capi sembrano essere però un punto vulnerabile della struttura, ma quando un capo smette di funzionare, viene subito rimpiazzato da un sottoposto o da un parigrado o da un capo del capo.
- Per evitare di avere un unico capo in cima all'albero, si può troncare l'albero in cima ed avere un comitato come massima autorità.

Algoritmo di allocazione gerarchico

- Esempio di gerarchia di processori vista come gerarchia organizzativa:



- In genere i capi o i direttori tengono traccia del numero di risorse inattive che hanno al di sotto anche se può accadere che queste informazioni non siano aggiornate.
- L'algoritmo non è propriamente distribuito, ma funziona bene ed è in grado di riparsi da solo e sopravvive alle cadute a tutti i livelli

Algoritmo di allocazione euristico distribuito

- I tre algoritmi di Eager ed altri già descritti rientrano in questa categoria, nell'algoritmo più efficiente dal punto di vista del costo che hanno studiato, quando si crea un processo, la macchina manda un messaggio di sondaggio ad una scelta a caso, chiedendogli se il suo carico sta al di sotto di una certa soglia: se lo è, il processo viene spedito su quella macchina, altrimenti si sceglie un'altra macchina per rifare il sondaggio. Se dopo N tentativi non trova nessuna macchina, l'algoritmo termina ed il processo viene mandato sulla macchina originale.
- Di questo modello è stato dato un modello analitico a code, studiando questo modello si è stabilito che l'algoritmo si comporta bene ed è stabile rispetto ad un vasto insieme di parametri quali diversi valori soglia, costi di trasferimento e limiti dell'operazione di sondaggio.

Algoritmo di allocazione ad asta



- In un'altra classe di algoritmi i processori mettono all'asta i loro cicli al migliore offerente, cercando di trasformare l'elaboratore in un sistema economico in miniatura (Ferguson e altri).
- Ciascun processore pubblicizza il proprio prezzo mettendolo su un file accessibile pubblicamente in lettura. Processori diversi hanno prezzi diversi in base alle loro prestazioni (velocità, dimensione della memoria, presenza di hardware particolare ed altre caratteristiche).
- Quando un processo vuol dar vita ad un processo figlio si guarda intorno per vedere chi offre il servizio di cui ha bisogno ed in base ad un proprio criterio di valutazione può indicare come migliore, a seconda dell'applicazione, il più economico, il più veloce, o quello con il miglior rapporto prezzo/prestazioni.
- In genere i prezzi messi in rete sono quelli degli ultimi compratori.

Lo scheduling nei sistemi distribuiti

- In genere ciascun processore ha internamente la propria politica di scheduling se ha più processi da far girare. Non sempre però questo tipo di scheduling indipendente è il più efficiente.
 - Esempio: se i processi A e B girano su un processore e i processi C e D su un altro e se ciascun processore è schedulato in timesharing con un quanto di 100 msec, per cui A e C girano durante i quanti pari e B e D in quelli dispari. Se A spedisce molti messaggi o fa molte RPC su D che sfortunatamente non si trova in esecuzione perché è il turno di C: dopo 100 msec avviene un cambio di contesto e D riceve il messaggio di A, esegue il lavoro e invia la risposta, ma poiché ora sul primo processore c'è B deve aspettare altri 100 msec prima che A riceva la replica. Il risultato è che viene scambiato un messaggio ogni 200 msec. Quello che si vuole è far sì che processi che comunicano frequentemente siano schedulati contemporaneamente.
- Sistemi distribuiti e paralleli

Processore		0	1
Intervallo temporale	0	A	C
	1	B	D
	2	A	C
	3	B	D
	4	A	C
	5	B	D

Lo scheduling nei sistemi distribuiti

- Ousterhout ha proposto alcuni algoritmi basati su ciò che egli chiama **co-scheduling** che prende in considerazione gli schemi di comunicazione durante lo scheduling per assicurare che tutti i membri di un gruppo girino nello stesso momento. Il primo algoritmo usa una matrice nella quale ogni colonna rappresenta la tabella dei processi per un certo processore.
- La colonna 4 contiene tutti i processi che girano sul quarto processore, mentre la riga 3 contiene tutti i processi che girano nel terzo quanto di qualche processore.

		Processore							
Intervallo temporale		0	1	2	3	4	5	6	7
	0	X				X			
	1			X			X		
	2		X			X		X	
	3	X					X		
	4		X		X				X
	5			X		X			

Lo scheduling nei sistemi distribuiti

- L'idea è di avere tutti i processori con uno scheduling round robin e di fare in modo che tutti i processori seguano l'ordine fissato. Per comunicare ad ogni processore quando effettuare il cambio di contesto si usa un messaggio di broadcast, in modo da mantenere tutti i quanti di tempo sincronizzati.
- Una variante prevede di spezzare la matrice in righe e concatenarle in un'unica riga lunga; con k processori qualunque gruppo di k elementi consecutivi appartiene a processori diversi. Per allocare un nuovo gruppo di processi si individua una finestra di k elementi nella riga lunga, in modo che l'elemento più a sinistra sia vuoto, ma l'elemento appena fuori il limite sinistro della finestra sia pieno. Se nella finestra è presente un numero sufficiente di elementi vuoti i processi vengono assegnati agli elementi, altrimenti la finestra scorre a destra e l'algoritmo viene ripetuto. Lo scheduling viene fatto facendo partire la finestra all'estremità sinistra e muovendosi verso destra per circa l'ampiezza di una finestra per ogni quanto di tempo, facendo attenzione a non dividere i gruppi sulle finestre.

Sistemi paralleli

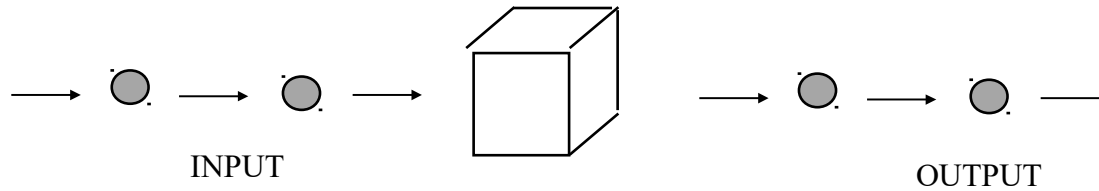
A thick, horizontal yellow brushstroke with a textured, painterly appearance, extending across the width of the slide below the title.

Il parallelismo

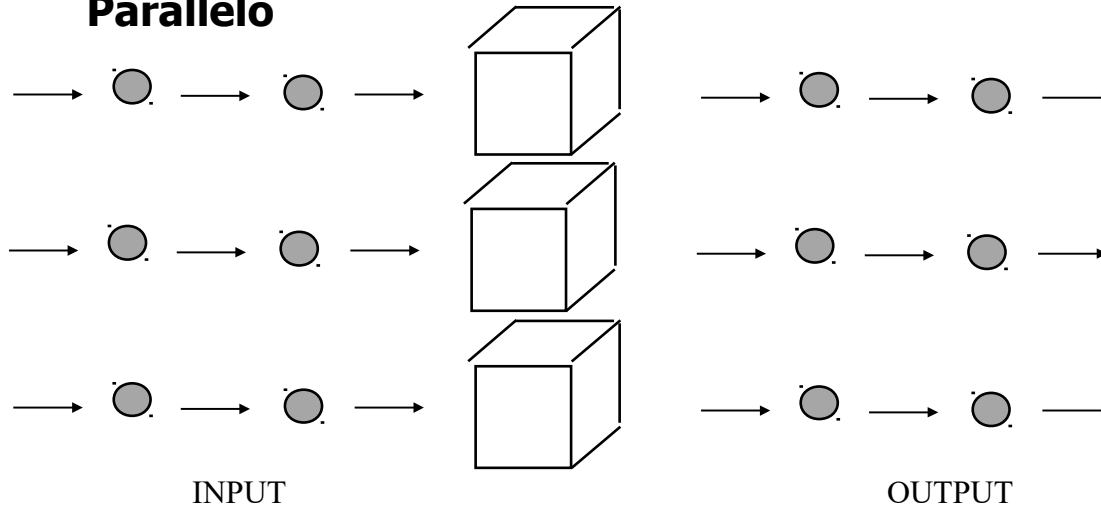
- ❑ Il **parallelismo** è un metodo per effettuare velocemente lavori lunghi e complessi.
- ❑ Un lavoro lungo può essere fatto **sequenzialmente**, cioè un passo dopo l'altro, o **decomposto** in piccoli lavori che possono essere portati avanti simultaneamente, in **parallelo**.
- ❑ Il parallelismo si attua:
 - Dividendo un lavoro in lavori più piccoli
 - Assegnando piccoli lavori a molti lavoratori che operano simultaneamente.
 - Coordinando i lavoratori.
 - Non dividendo a tal punto il lavoro da perdere più tempo ad assegnare un piccolo lavoro ad un altro che a farlo da sè.
- ❑ **Esempi di parallelismo:** costruzione di edifici, catene di montaggio ecc.

II Parallelismo

Sequenziale



Parallelo



La programmazione sequenziale

- ❑ Tradizionalmente, i programmi sono stati scritti per computer sequenziali.
- ❑ Caratteristiche della programmazione sequenziale:
 - Le istruzioni vengono eseguite una alla volta
 - Viene utilizzata una sola unità centrale di elaborazione (processor)
 - La velocità di esecuzione dipende da quanto velocemente i dati possono circolare sull'hardware
 - ❑ velocità della luce = 30 cm/nano secondi ($3 \cdot 10^8$ m/s)
 - ❑ limite del filo di rame = 9 cm/nano secondi ($9 \cdot 10^7$ m/s)
 - Le macchine più veloci eseguono circa 9-12 miliardi di istruzioni al secondo.

Applicazioni complesse

- Ci sono varie classi di problemi che richiedono elaborazioni più veloci ed efficienti:
 - problemi di simulazione e di “Modeling”
 - basati su approssimazioni successive
 - molti calcoli, molto precisi
 - problemi dipendenti dal calcolo di grandi insiemi di dati
 - elaborazione di immagini e segnali (Image Rendering)
 - database ed estrazione dati
 - problemi basati su dati molto variabili
 - modelli climatici
 - dispersione dell’inquinamento
 - modelli per superconduttori
 - visione e percezione
 - turbolenza dei fluidi
 - modelli per semiconduttori
 - sistemi di combustione
 - simulazione sistemi economici

Esempi di applicazioni



Galaxy Formation



Planetary Movments



Climate Change



Rush Hour Traffic



Plate Tectonics



Weather



Auto Assembly



Jet Construction



Drive-thru Lunch

Sistemi distribuiti e paralleli

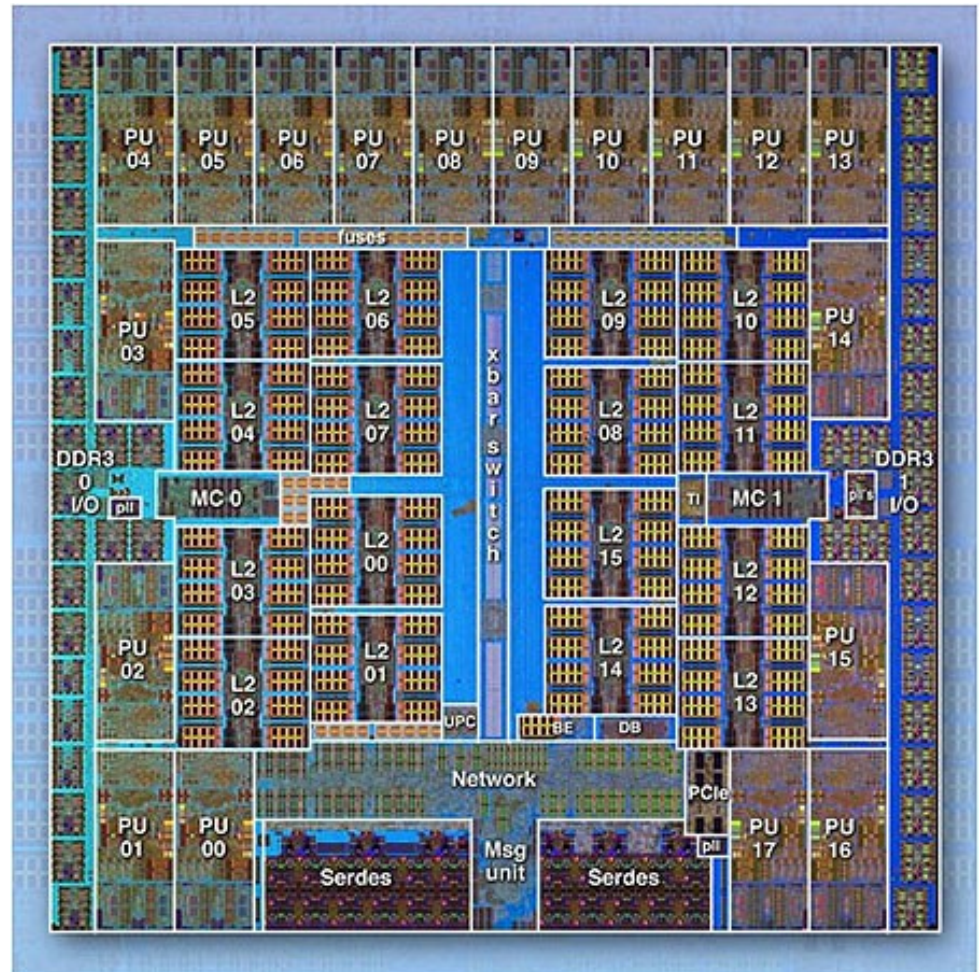
Computer paralleli

- ❑ Virtualmente tutti i computer stand-alone attualmente possono considerarsi paralleli dal punto di vista dell'hardware che possiedono:
 - Impegnare Unità funzionali multiple
 - (cache L1, cache L2, graphics processing unit (GPU), ecc.)
 - Unità/core di esecuzione multiple (CPU)
 - Threads multipli
- ❑ Nella maggior parte dei casi, i programmi seriali eseguiti su computer moderni "sprecano" la potenziale potenza di calcolo

Computer paralleli

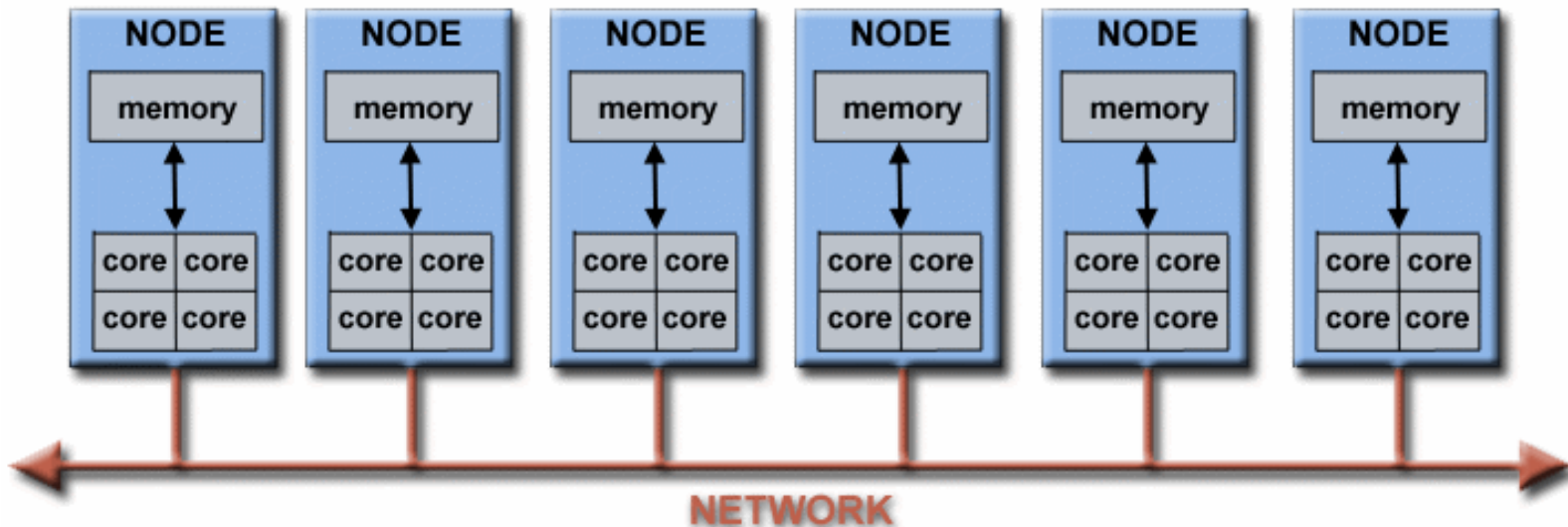
□ Esempio:

Chip IBM BG/Q con 18 core (CPU) and 16 unità Cache di tipo L2



Supercomputer paralleli

- Cluster di più grandi computer paralleli possono essere configurati tramite una rete che collega tra loro nodi formati da computer stand-alone.



Supercomputer paralleli

- Per esempio nello schema di uno dei più veloci computer paralleli al mondo (chiamato Sierra, del LLNL Lawrence Livermore National Laboratory)
 - Ogni nodo è un multi-processore parallelo esso stesso
 - Nodi multipli sono collegati tra loro con una rete Infiniband
 - Nodi special purpose, anch'essi multi-processori, vengono usati per altri scopi.

Supercomputer paralleli



compute node



login / remote partition server node



infiniband switch



gateway node

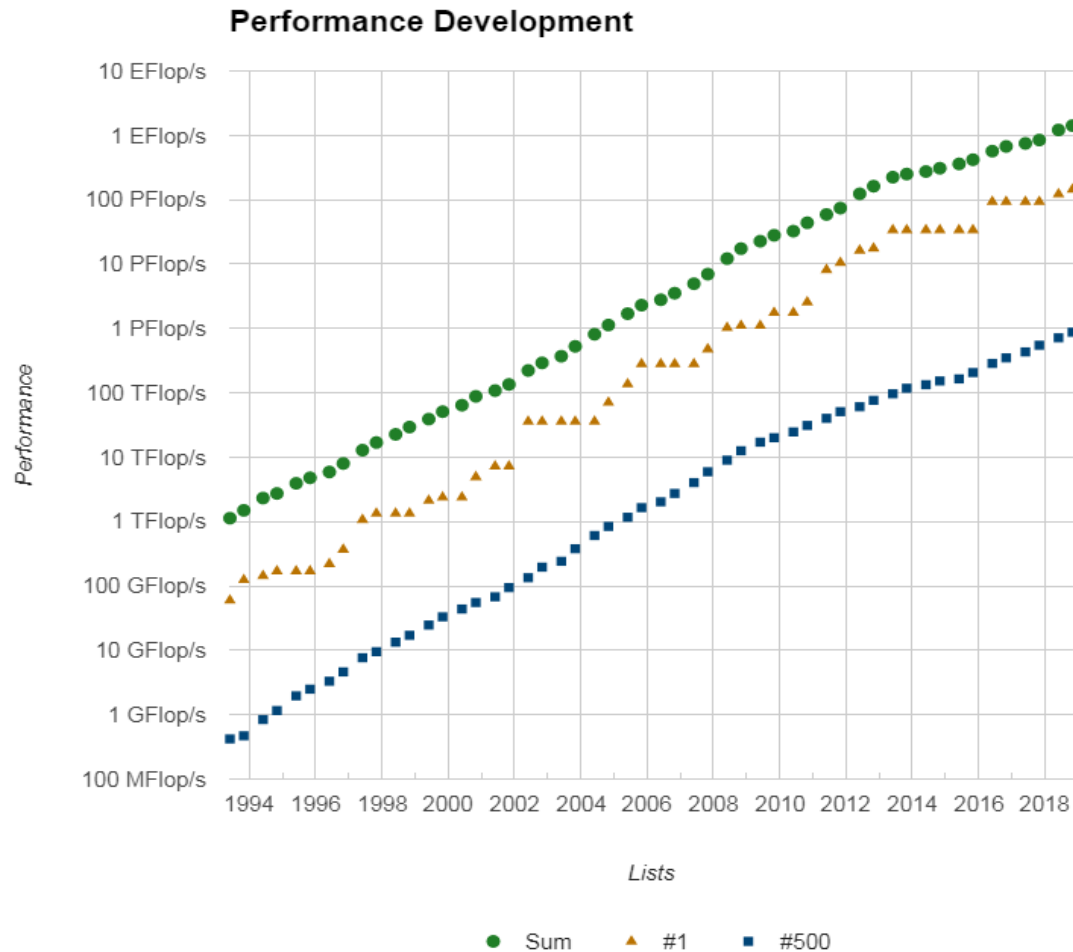


management hardware

Computer paralleli

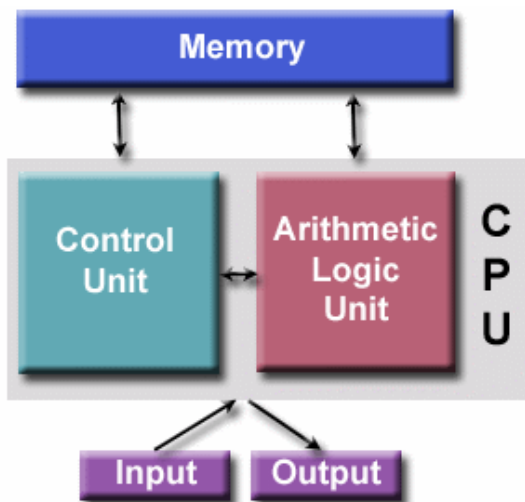
- ❑ Negli ultimi 20 anni, le tendenze indicate da reti, sistemi distribuiti e architetture di computer multiprocessore sempre più veloci (anche a livello desktop) mostrano chiaramente che il parallelismo è il futuro dell'informatica.
- ❑ Nello stesso periodo di tempo, si è registrato un aumento di oltre 500.000 volte delle prestazioni del supercomputer, senza che sia attualmente in vista alcuna fine.
- ❑ La gara è già iniziata per Exascale Computing
Exaflop = 10^{18} calcoli al secondo

Computer paralleli



Architettura di von Neumann

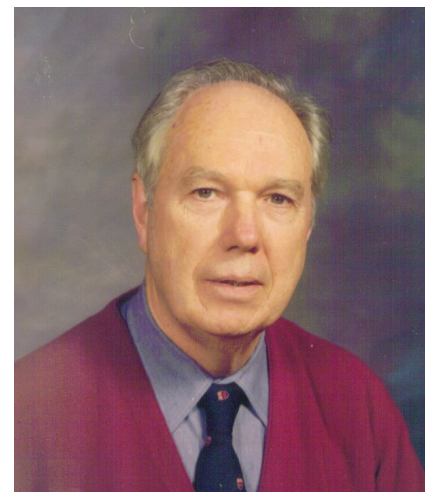
- Computer di von Neumann: Nel 1946 John von Neumann stabiliva i principi generali di progettazione di un elaboratore elettronico digitale. In conformità a tali principi è stata costruita la maggioranza dei calcolatori e probabilmente lo sarà ancora in futuro. In particolare uno di questi principi enuncia che in ogni momento all'interno di una unità di elaborazione viene svolta soltanto un'azione.



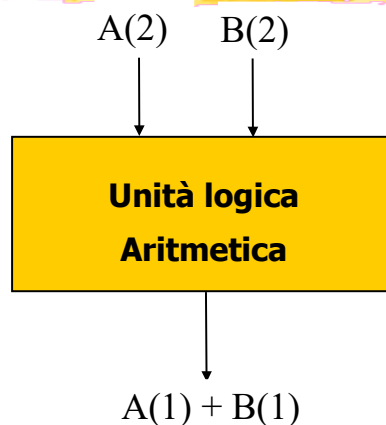
Tassonomia di Flynn

- ❑ Esistono vari metodi usati per classificare i computer, un'unica tassonomia non va bene per tutte le configurazioni
- ❑ La **tassonomia di Michael Flynn** (1966) usa la relazione tra istruzioni di programma e dati di programma.
- ❑ I quattro modelli in cui si dividono le architetture sono:

S I S D Single Instruction stream Single Data stream	S I M D Single Instruction stream Multiple Data stream
M I S D Multiple Instruction stream Single Data stream	M I M D Multiple Instruction stream Multiple Data stream



SISD



- Non è un computer parallelo
- E' un computer di von Neumann convenzionalmente seriale e scalare
- Ha un unico flusso di istruzioni
- Ad ogni ciclo di clock viene eseguita un'unica istruzione

SISD

- Ogni istruzione opera su un singolo (scalare) elemento dei dati
- Il modello è limitato dal numero di istruzioni che possono essere eseguite in una data unità di tempo
- La prestazione è misurata frequentemente in MIPS (Milioni di Istruzioni Per Secondo)
- La maggioranza delle applicazioni non rientra nella categoria dei supercomputer

□ Esempio:

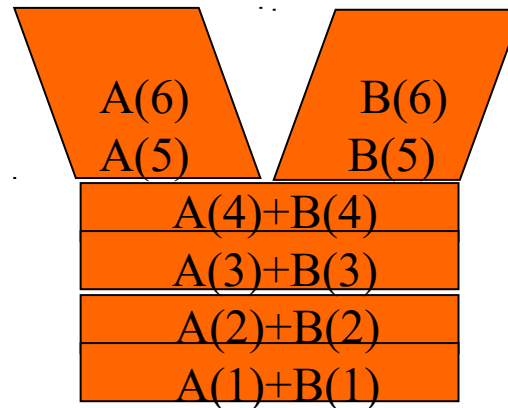
- Per calcolare la somma dei numeri a_1, a_2, \dots, a_N un processore richiede N consecutivi accessi alla memoria (ognuno per prendere un numero). Così vengono eseguite $N-1$ somme in sequenza. Il calcolo ha una complessità di $O(N)$ operazioni.

SIMD

- Architettura ancora di von Neumann ma con istruzioni potenti
- Ogni istruzione può operare su più elementi di dati
- Di solito un *host* intermediario esegue il programma logico e trasmette istruzioni agli altri processori
- Sincrono (**lockstep** = ad ogni passo tutti i processori eseguono la stessa operazione su dati diversi))
- Valutare quanto velocemente queste macchine possono eseguire istruzioni non è una buona misura delle loro prestazioni
- La prestazione è misurata in MFLOPS (Milioni di operazioni floating point per secondo)
- ✧ Esistono 2 tipi di SIMD:
 - SIMD vettoriale
 - SIMD parallelo

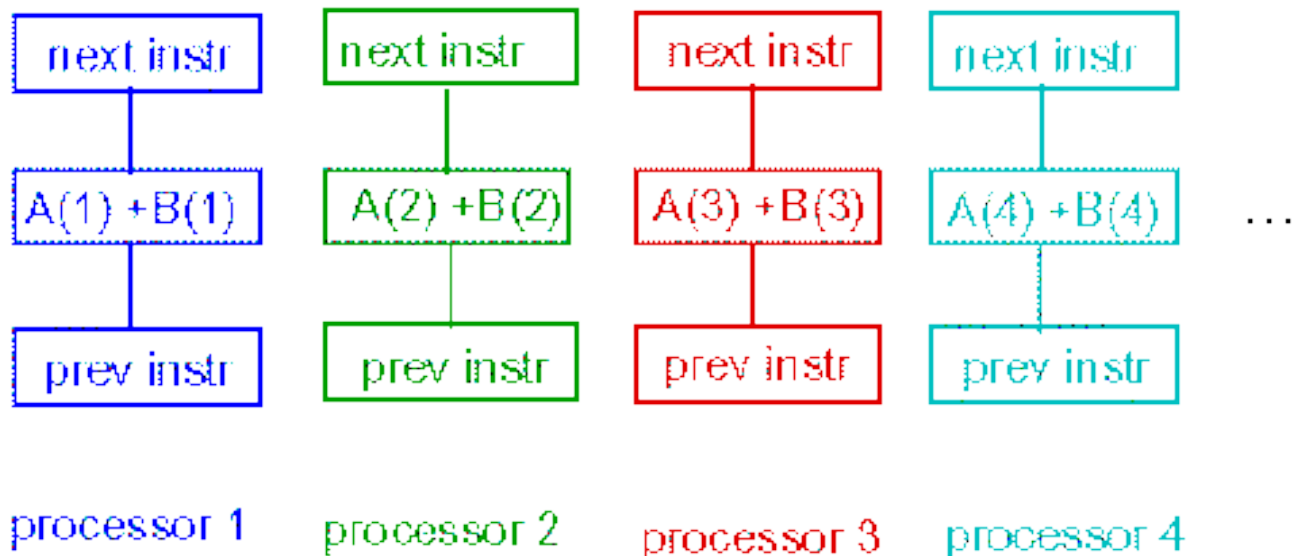
SIMD vettoriale

Confronta l'esponente
Aggiusta l'esponente
Somma la mantissa
Normalizza



- Una sola istruzione implica l'aggiornamento di molti dati.
- Un'**esecuzione scalare** agisce su singoli elementi di dati. Un'**esecuzione vettoriale** opera su un intero vettore (gruppo) di dati nello stesso istante.
- ✧ Esempi di macchine vettoriali:
 - Cray 1, NEC SX-2, Fujitsu VP, Hitachi 5820
oppure singoli processori di:
 - Cray C90, Cray2, NECSX-3, Fujitsu VP 2000, Convex C-2

SIMD parallelo



- La singola istruzione vettoriale è inviata a tutti i processori che eseguono la stessa istruzione, ma ognuno su un insieme diverso di dati.
- I processori operano in maniera sincrona, "lockstep"

SIMD parallelo

□ Vantaggi

- 👍 Permettono di operare su vettori in tempi paragonabili a quelli necessari per operare su elementi singoli (scalari)
- 👍 I cicli DO conducono a un parallelismo SIMD

```
do 100 i=1,100
  c(i)=a(i)+b(i)
100 continue
```
- 👍 La sincronizzazione non è un problema, poichè tutti i processori operano in lock-step.

□ Svantaggi

- 👎 Decisioni dentro cicli DO possono risultare penalizzanti in esecuzione poichè richiedono a tutti i processori di effettuare l'operazione controllata dalla decisione, sia che i risultati vengano effettivamente usati, sia che non si utilizzino.
- ✧ Esempi di macchine SIMD parallele:
 - Connection Machine CM-2, Maspar MP-1, MP-2

SIMD parallelo

□ Esempio:

✧ Applicazione su SIMD parallele:

Sommare 2 matrici $A+B=C$.

Se le 2 matrici A e B sono di ordine 2 avremo bisogno di 4 processori:

$$A_{11} + B_{11} = C_{11} \quad A_{12} + B_{12} = C_{12}$$

$$A_{21} + B_{21} = C_{21} \quad A_{22} + B_{22} = C_{22}$$

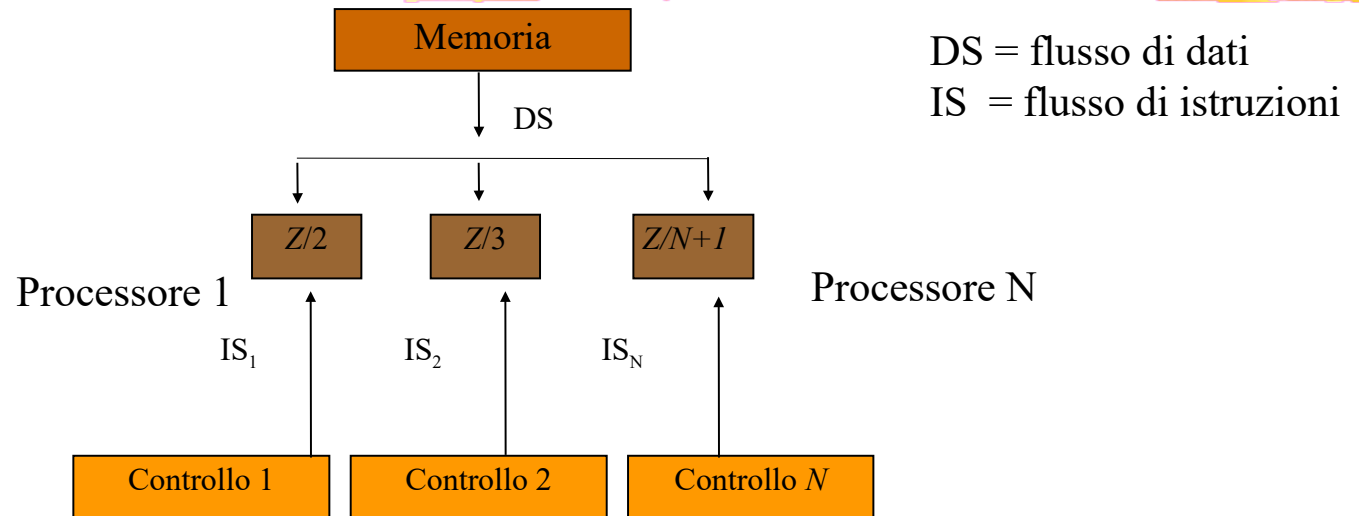
La stessa istruzione è inviata a tutti i 4 processori (somma 2 numeri) e tutti i processori eseguono l'istruzione simultaneamente. L'operazione è svolta quindi in un solo passo, mentre su macchine sequenziali occorrono 4 passi.

□ In molti problemi risolti su architetture SIMD (e MIMD) è necessario che i processori siano capaci di comunicare fra di loro per scambi di dati o risultati.

□ Questo può essere fatto in 2 modi:

- usando memoria condivisa e variabili condivise
- usando una memoria distribuita, ma una rete di interconnessione e un "message-passing".

MISD



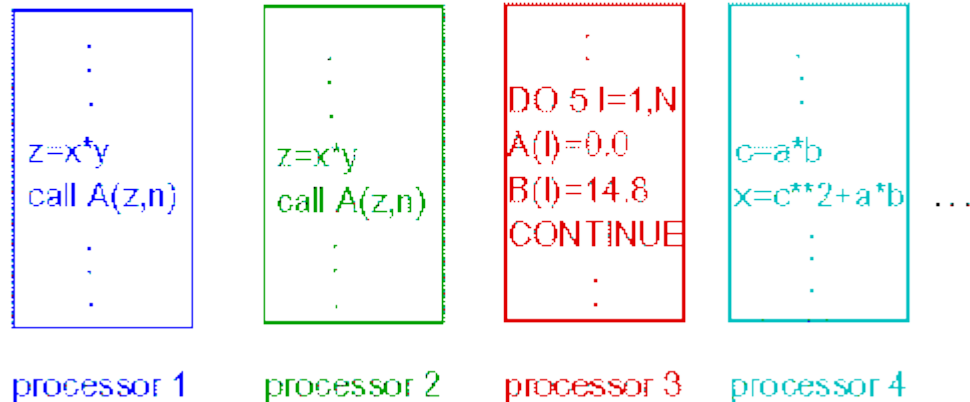
- Ci sono N flussi di istruzioni (algoritmi/programmi) e 1 solo flusso di dati. I vari processori svolgono differenti operazioni con lo stesso dato nello stesso istante.
- Macchine MISD sono usate in calcoli in cui lo stesso input deve essere sottoposto a molte diverse operazioni.
- Per loro caratteristica le macchine MISD sono molto mirate e poco duttili, per tale ragione non esistono macchine commerciali con questa architettura

MISD



- ❑ Macchine MISD sono usate in calcoli in cui lo stesso input deve essere sottoposto a molte diverse operazioni.
- ❑ Per loro caratteristica le macchine MISD sono molto mirate e poco duttili, per tale ragione non esistono macchine commerciali con questa architettura
- ❑ **Esempio:**
 - ✧ Controllare se un dato numero Z è primo.
 - ✧ Una semplice soluzione è di tentare tutte le possibili divisioni di Z .
 - ✧ Se N è il numero di processori $\rightarrow N=Z-2$. Tutti i processori prendono Z come input e tentano di dividerlo con il proprio divisore associato in questo modo è possibile stabilire in un solo passo se Z è primo o no.
 - ✧ Più realisticamente si può pensare a $N < Z-2$ e un sottoinsieme di divisori sarà assegnato ad ogni processore.

MIMD



- Questo è il modello più generale e più potente della classificazione di Flynn.
Si hanno N processori, N flussi di istruzioni e N flussi di dati.
Ogni processore opera sotto il controllo di un flusso di istruzioni inviatogli dalla propria unità di controllo, cioè ogni processore è capace di eseguire il proprio programma su differenti dati. Ogni processore opera in maniera asincrona.
- Il modello include tutte le forme di configurazioni di multiprocessori.
- Secondo il numero di unità di elaborazione funzionanti in parallelo si distinguono macchine **a grana grossa** (con poche unità piuttosto potenti), da quelle **a grana fine** (con molte unità, ciascuna di scarsa potenza) (vedi *Granularità*).

MIMD

□ Vantaggi:

- 👍 I processori possono eseguire flussi di lavori multipli simultaneamente.
- 👍 Il singolo processore può eseguire ogni operazione non curandosi di ciò che stanno facendo gli altri processori.
- 👍 Le prestazioni salgono fino all'ordine di 1000 MFLOPS.

□ Svantaggi:

- 👎 sono necessari prima un bilanciamento del carico e quindi una sincronizzazione per coordinare i processori alla fine di una struttura parallela all'interno di una singola applicazione.

□ Computer MIMD che usano la memoria condivisa vengono chiamati **MULTIPROCESSORI** o "TIGHTLY COUPLED MACHINES" (Macchine collegate solidamente). (Es. ENCORE, MULTIMAX)

□ Computer MIMD che hanno una rete di interconnessione vengono chiamati **MULTICOMPUTER** o "LOOSELY COUPLED MACHINES" (macchine collegate liberamente) All'interno di questi ultimi MULTICOMPUTER si ricordano:

- ✧ Sequent, nCUBE, intel iPSC/2, cluster di IBM RS6000, Cray C 90, Cray 2, NEC SX-3, Fujitsu VP 2000, Convex C-2, Intel Paragon, CM5, KSR-1, IBM SP1, IBMSP2

Riepilogo dei 4 modelli

Operazioni	dati	macchina	risultati
+	AB	SISD	\longrightarrow A+B
+	AB CD	SIMD	\longrightarrow A+B \longrightarrow C+D
+	AB	MISD	\longrightarrow A+B \longrightarrow A*B
+	AB	MIMD	\longrightarrow A+B
*	CD		\longrightarrow C*D

SPMD

- Il modello SIMD, già visto, è un esempio di parallelismo sincrono in cui ogni processore operava in "lockstep". Il modello SPMD ne è la relativa versione asincrona. Questo si può ottenere facendo girare lo stesso programma nei vari processori di una macchina MIMD.
- SPMD non è un modello hardware, è solo l'equivalente software del SIMD.
- Poiché un intero programma viene eseguito su dati separati è possibile che vengono intrapresi cammini diversi, ciò ci conduce ad un parallelismo asincrono. I processori per poco eseguono le stesse istruzioni in sincrono, poi vengono impegnati ad eseguire istruzioni diverse dello stesso programma.

SPMD

□ Esempio:

il programma sia:

```
.  
.   
IF X = 0  
THEN S1  
ELSE S2
```

Assumendo: $x=0$ in P1

$x \neq 0$ in P2

Mentre il processore P1 esegue S1, il processore P2 esegue S2 (cosa che non sarebbe possibile su una macchina SIMD).

Comunicazioni fra processori



- Allo scopo di coordinare le parti di lavoro che svolgono nodi multipli sullo stesso problema, occorre prevedere una qualche forma di comunicazione tra processori per:
 - trasmettere informazioni
 - sincronizzare le attività
- Il modo in cui i processi comunicano è dipendente dall'architettura di memoria, che, a sua volta, influenza il modo di scrivere programmi paralleli.

Architetture di memoria

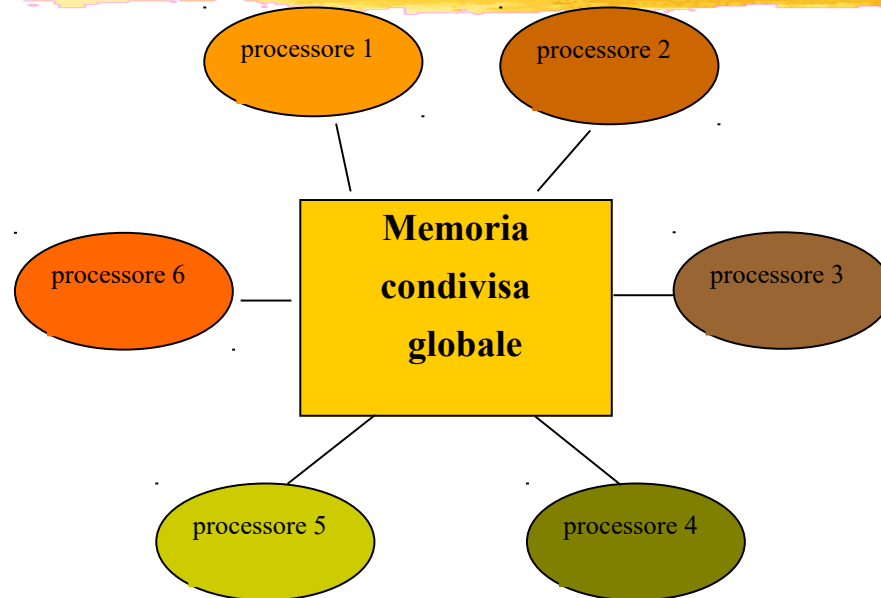


○ Le architetture primarie di memoria sono:

□ **Memoria condivisa** (Shared Memory)

□ **Memoria distribuita** (Distributed Memory)

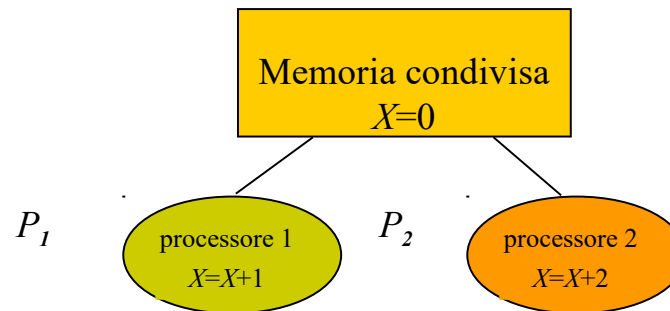
Memoria condivisa (shared memory)



- Molti processori operano indipendentemente fra loro, ma condividono le stesse risorse di memoria
- Solo un processore alla volta può accedere alla locazione di memoria condivisa
- La sincronizzazione è ottenuta controllando i lavori che "leggono da" e "scrivono su" la memoria condivisa
- La memoria condivisa risolve le comunicazioni tra processori, ma introduce il problema dell'accesso simultaneo alla stessa locazione di memoria.

Memoria condivisa

Esempio:



Casi	Sequenza di esecuzione	Risultato di P_1	Risultato di P_2	Ris. finale
a	$P_1(l+s) \rightarrow P_2(l+s)$	1	3	3
b	$P_2(l+s) \rightarrow P_1(l+s)$	3	2	3
c	$P_1(l) \rightarrow P_2(l) \rightarrow P_1(s) \rightarrow P_2(s)$	1	2	2
d	$P_2(l) \rightarrow P_1(l) \rightarrow P_2(s) \rightarrow P_1(s)$	1	2	1
e	$P_1(l) \rightarrow P_2(l) \rightarrow \begin{matrix} \nearrow P_1(s) \\ \searrow P_2(s) \end{matrix}$	1	2	?

Memoria condivisa

Supponiamo che X sia una variabile condivisa di P_1 e P_2 .

Secondo la sequenza di lettura (l) e scrittura (s) dei 2 processi il risultato finale può assumere i valori $X = 1, 2$, o 3 .

Nel caso poi che P_1 e P_2 finiscono il calcolo nello stesso tempo (caso e) un solo valore può essere memorizzato nella locazione per X .

Quando un programma parallelo con gli stessi dati di input produce differenti risultati in differenti esecuzioni siano di fronte a un problema di NON DETERMINATEZZA.

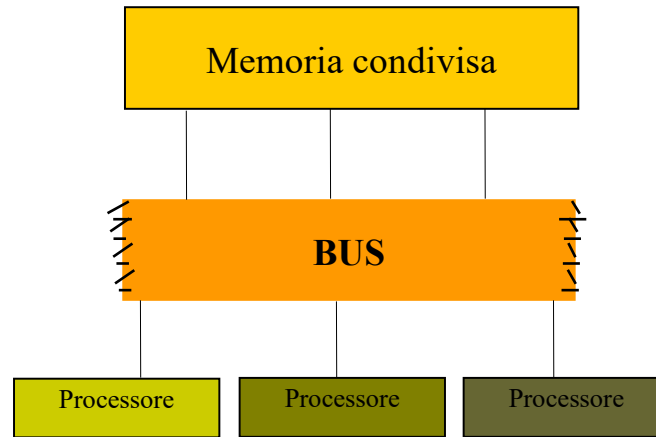
La NON DETERMINATEZZA è causata dalle "condizioni di contesa".

Una contesa avviene quando due istruzioni in processi concorrenti accedono alla stessa locazione di memoria, almeno una di queste è una istruzione di scrittura e non è garantito l'ordine di esecuzione dei processi.

Il problema della NON DETERMINATEZZA può essere risolto sincronizzando l'uso dei dati condivisi.

Nell'esempio le istruzioni $X = X + 1$ e $X = X + 2$ sono puntualmente esclusive, cioè non devono essere eseguite allo stesso istante, ciò riconduce ai soli casi **a** e **b**.

Memoria condivisa



- ❑ Computer di tipo "shared Memory" (SEQUENT, ENCORE, CRAY, CRAY-MP, CONTEX C-2, Cray C-90) sono stati implementati incorporando un "bus" veloce per connettere i processori alla memoria.
- ❑ Poichè il bus ha una banda finita, cioè ad un certo istante può essere spedito un numero finito di dati, allora non appena aumenta il numero di processori la contesa per il bus diventa un problema. Perciò diventa possibile permettere a P processori di accedere a P locazioni di memoria simultaneamente solo se P è relativamente piccolo (es. <30)

Memoria condivisa

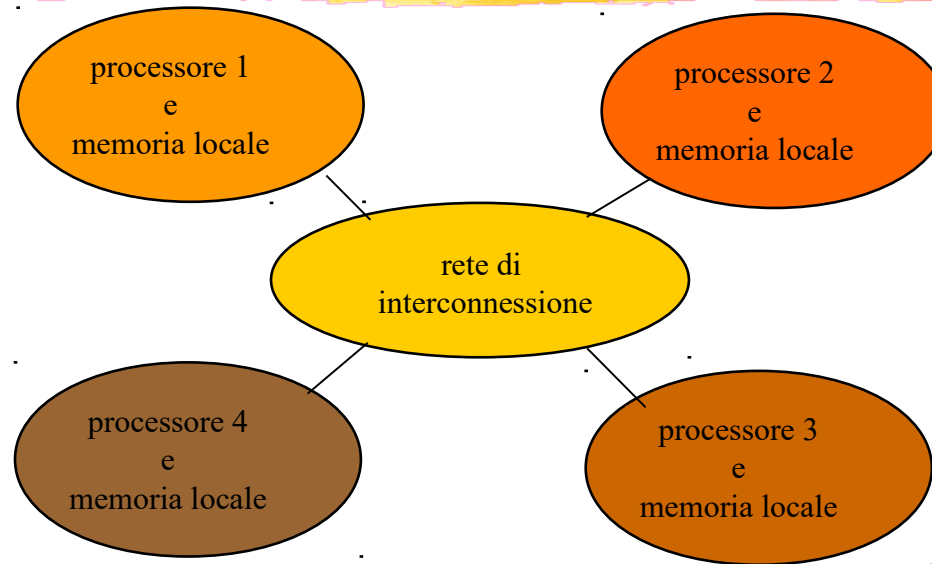
□ Vantaggi della memoria condivisa:

- 👍 E' semplice per l'utente usarla in modo efficiente
- 👍 La condivisione dei dati fra processi avviene velocemente (velocità di accesso alla memoria)

□ Svantaggi della memoria condivisa:

- 👎 Il bus ha banda limitata. Aumentare i processori senza aumentare la banda può causare gravi "colli di bottiglia"
- 👎 All'utente è data la responsabilità di specificare la sincronizzazione ("locks")

Memoria distribuita (distributed memory)



- Molti processori operano indipendentemente fra loro, ma ognuno ha la propria memoria privata.
- I dati vengono condivisi tramite una rete di comunicazione usando il *message passing* (scambio di messaggi).
- Sempre usando il *message passing* l'utente ha la possibilità di attuare le sincronizzazioni.
- Se un processore richiede un dato contenuto in un altro processore, il dato deve essere esplicitamente "passato" usando le funzioni **send** e **receive** per la comunicazione

Memoria distribuita

❏ Esempio:

P_1	P_2
receive (X, P_2)	send(X, P_1)

⇒ il valore di X è “passato” esplicitamente da P_2 a P_1

Questo modo di operare viene chiamato “**MESSAGE PASSING**”.

❏ Vantaggi della memoria distribuita

- 👍 Memoria scalabile al numero di processori
- 👍 Aumentando il numero di processori aumenta la dimensione della memoria e la banda.
- 👍 Ogni processore può rapidamente accedere alla propria memoria senza rischi di interferenza

Memoria distribuita

❑ Svantaggi della memoria distribuita

- 👉 E' difficile mappare strutture dati esistenti a questa organizzazione della memoria.
- 👉 L'utente ha la responsabilità di inviare e ricevere i dati nei vari processori.
- 👉 Per minimizzare l'overhead e la latenza nella comunicazione, i dati potrebbero essere raggruppati in grandi blocchi e spediti prima di ricevere la richiesta di un nodo.

- ❑ Esempi di computer basati su architettura a memoria distribuita sono: nCUBE Hypercube, Intel Hypercube, TMC CM-5, IBM SP1, IBM SP2, Intel Paragon.

Architetture memorie/processori

❑ Memoria distribuita

- MPP - Massively Parallel Processor

❑ Shared Memory

- SMP - Symmetric Multiprocessor

- ✧ processori identici
- ✧ uguale accesso alla memoria
- ✧ a volte chiamati UMA - Uniform Memory Access
- ✧ o CC-UMA - Cache Coherent UMA
- ✧ Cache coherent significa che se un processo aggiorna una locazione nella memoria condivisa, tutti gli altri processori vengono a conoscenza del cambiamento effettuato

- NUMA - Non Uniform Memory Access

- ✧ a volte chiamati CC-NUMA - Cache Coherent NUMA
- ✧ Spesso attuati "linkando" (collegando) due o tre SMP
- ✧ Un SMP può direttamente accedere alla memoria di un altro SMP
- ✧ Non tutti i processori hanno tempo di accesso uguale alle memorie
- ✧ Accessi alla memoria tramite link sono più lenti

Architetture memorie/processori

□ Combinazioni

○ Più SMP connessi tramite rete

- ✧ I processori comunicano con un SMP via memoria
- ✧ Richiede il message passing tra i vari SMP
- ✧ Un SMP non può accedere direttamente alla memoria di un altro SMP

○ Più processori a memoria distribuita connessi a una memoria condivisa più grande

- ✧ La piccola memoria veloce può essere utilizzata per fornire i dati ai processori, mentre la grande memoria lenta può essere usata per un "backfill" alle memorie più piccole.
- ✧ E' simile alle gerarchia:
 - registri ⇐ memoria cache ⇐ memoria principale
- ✧ Trasferimenti dalla memoria locale alla memoria condivisa dovrebbero essere trasparenti all'utente
- ✧ Probabilmente in futuro questo tipo di configurazione prevederà molti processori con la loro memoria locale che circondano una grande memoria condivisa il tutto su una stessa scheda.

Architetture

Confronti:

	CC-UMA	CC-NUMA	MPP
Architetture	di solito RISC	di solito RISC	RISC istruzioni potenti
Esempi	SMP Sun Vexx DEC SGI Challenge	SGI Origin Sequent esemplari HP DEC	Cray T3E Maspar IBM SP2
Comunicazioni	MPI shmem	MPI shmem	MPI
Scalabilità	a decine di processori	a centinaia di processori	a migliaia di processori
Svantaggi	ampiezza di banda della memoria limitata	nuova architettura	programmazione di amministratore di sistema
Disponibilità SW	molte migliaia di ISV	molte migliaia di ISV comunicazioni point-to-point	decine di ISV difficile da sviluppare e da mantenere

Paradigmi della programmazione parallela

- Ci sono molti metodi di programmare computer paralleli:
- **Message Passing**: l'utente effettua chiamate a librerie per passare esplicitamente informazioni tra i processori
- **Data parallel**: il partizionamento dei dati determina il parallelismo.
- **Memoria condivisa** (Shared Memory): molti processi condividono lo spazio di memoria comune.
- **Operazione di Memoria Remota** (Remote Memory Operation): insieme di processi in cui un processo può accedere alla memoria di un altro processo senza l'intervento di quest'ultimo.
- **Threads** (fili): un singolo processo ha più vie di esecuzione (concorrenti)
- **Modelli combinati**: composto da due o più metodi sopra citati.

Paradigmi della programmazione parallela

□ Due dei più comuni sono:

- il *message passing* e

- il *data parallel*.

✧ **N.B.** : questi modelli sono indipendenti dalla macchina e dall'architettura, ognuno di essi può essere implementato, grazie al supporto di un appropriato sistema operativo in qualsiasi hardware. Una implementazione efficace è quella che raggiunge in pieno i propri obiettivi hardware e permette all'utente di programmare facilmente.

Message passing

- ❑ Il paradigma *message passing* è così definito:
 - insieme di processi che utilizzano solo la memoria locale
 - processi che comunicano fra loro attraverso messaggi di tipo **SEND** e **RECEIVE**.
 - Il trasferimento dei dati richiede operazioni congiunte che devono essere effettuate da ogni processo: un'operazione di SEND deve avere la sua corrispettiva operazione di RECEIVE (ciò implica eventuali stati di attesa).
- ❑ La programmazione con il paradigma *message passing* viene effettuata sia attraverso *link* o costruendo chiamate a librerie che gestiscono lo scambio dati tra processori.
- ❑ Le librerie *message passing* sono disponibili per i più moderni linguaggi di programmazione (C, JAVA, FORTRAN,...).

Data parallel

- ❑ Il paradigma ***data parallel*** è così definito:
 - Ogni processo lavora su una parte distinta di una stessa struttura dati
 - Esiste uno spazio dei nomi globale
 - Generalmente è un approccio di tipo SPMD (Single Program Multiple Data)
 - I dati vengono distribuiti sui vari processori
 - Tutto lo scambio dei messaggi tra i processori è fatto in maniera trasparente al programmatore
 - In genere viene costruito sopra ad una delle più comuni librerie di *message passing*
- ❑ La programmazione con il paradigma *data parallel* viene effettuata scrivendo un programma con costrutti di tipo *data parallel* e compilandolo con un compilatore specifico.
- ❑ Il compilatore converte il programma in un codice standard e inserisce le chiamate alle funzioni di libreria del *message passing* per distribuire i dati a tutti i processi.

Implementazioni



○ MESSAGE PASSING

- **MPI** - *Message Passing Interface*

- **PVM** - *Parallel Virtual Machine*

- **MPL** - *Message Passing Library*

○ DATA PARALLEL

- **FORTRAN 90/High Performance Fortran**

MPI



- Message Passing Interface viene più comunemente chiamato **MPI**
- La definizione di *MPI* come libreria standard portabile di message-passing è stata sviluppata nel 1993 da un gruppo di rivenditori di hardware e software.
- *MPI* è disponibile sia per il linguaggio C che per FORTRAN
- *MPI* è installabile su una grande varietà di macchine parallele
- La piattaforma ottimale è un sistema a memoria distribuita (come per es. l'IBM SP)
- Tutte le comunicazioni tra processi sono fatte attraverso il *message passing*
- Il parallelismo è esplicito : è a cura del programmatore di attuare il parallelismo sul programma e di implementare i costrutti *MPI*
- Il modello di programmazione è di tipo SPMD (Single Program Multiple Data).

PVM

- Parallel Virtual Machine, comunemente chiamato **PVM**, permette ad un insieme di sistemi di computer diversi di essere visto come una singola macchina parallela.
- Originariamente *PVM* è stato sviluppato dal Oak Ridge National Laboratory nel 1989 come un pacchetto di ricerca per esplorare calcoli su reti eterogenee. Ora è disponibile come pacchetto software di pubblico dominio.
- *PVM* è un pacchetto software usato per creare ed eseguire applicazioni concorrenti o parallele.
- *PVM* opera su gruppi di computer UNIX eterogenei collegati tra loro da una o più reti.
- Tutte le comunicazioni vengono effettuate tramite *message passing*
- *PVM* è composto da 2 componenti principali:
 - il processo *PVM daemon* che gira su ogni processore
 - le routine di interfaccia di libreria che forniscono il controllo sui processori e il *message passing*.

F90/High Performance Fortran HPF

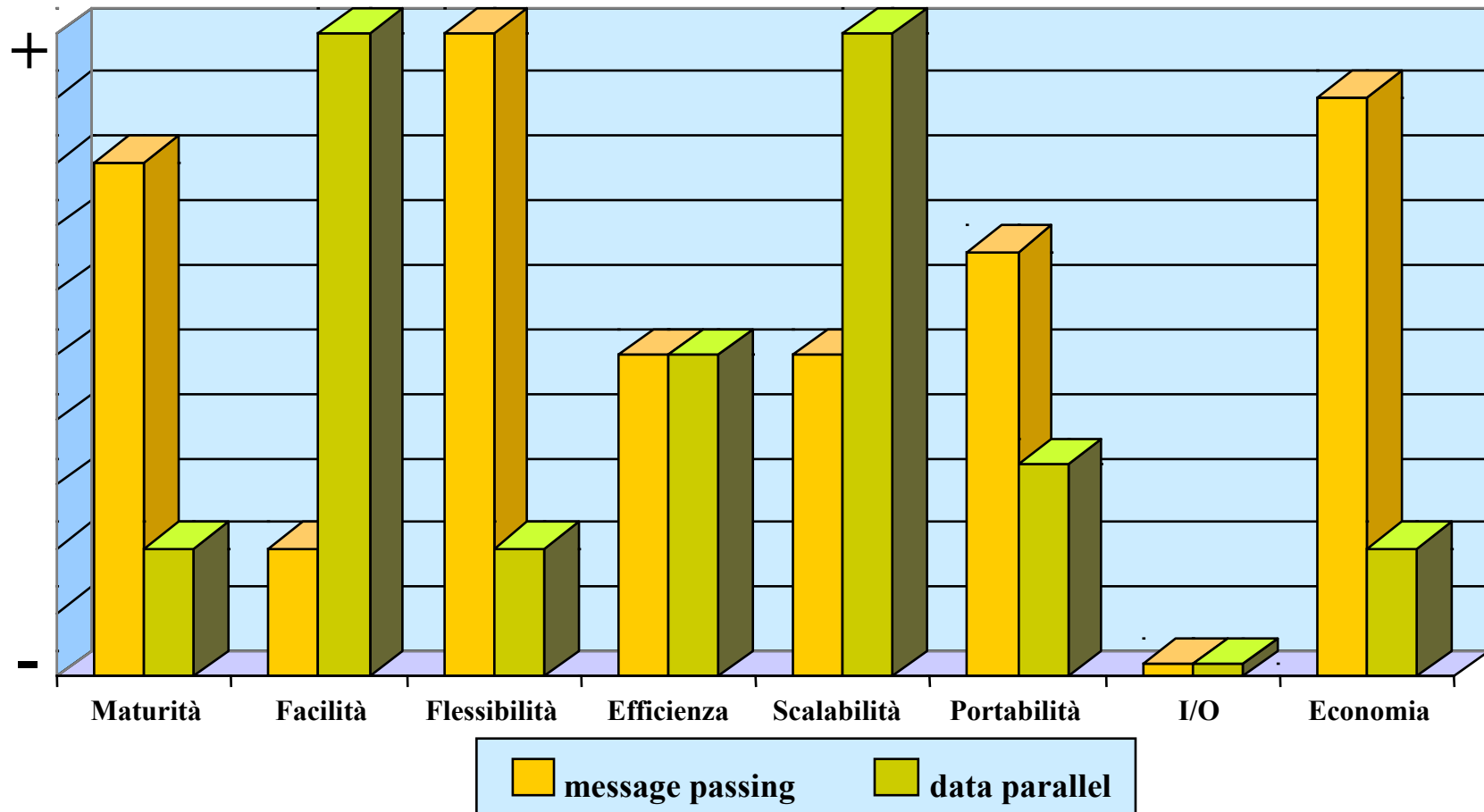


- Il *Fortran 90* (F90) è l'estensione ISO/ANSI al FORTRAN 77
- *High Performance Fortran* chiamato **HPF** è una estensione del F90 per supportare la programmazione parallela dei dati
- Le direttive del compilatore forniscono al programma le specifiche sulla distribuzione dei dati e l'allineamento
- Nuovi costrutti e funzioni intrinseche del compilatore permettono al programmatore di fare calcoli e manipolare dati con differenti distribuzioni

MPL

- La Message Passing Library viene comunemente chiamata **MPL**
- *MPL* fa parte del pacchetto IBM chiamato SP Parallel Environment
- Le routine di *message passing* sono di proprietà IBM
- *MPL* è stata ideata per fornire un insieme semplice ed efficiente di operazioni per coordinare e far comunicare i processori che agiscono su applicazioni parallele
- La piattaforma ideale per la *MPL* è un sistema a memoria distribuita (tipo IBM/SP), ma è anche ben supportata su cluster di Risc RS/6000.

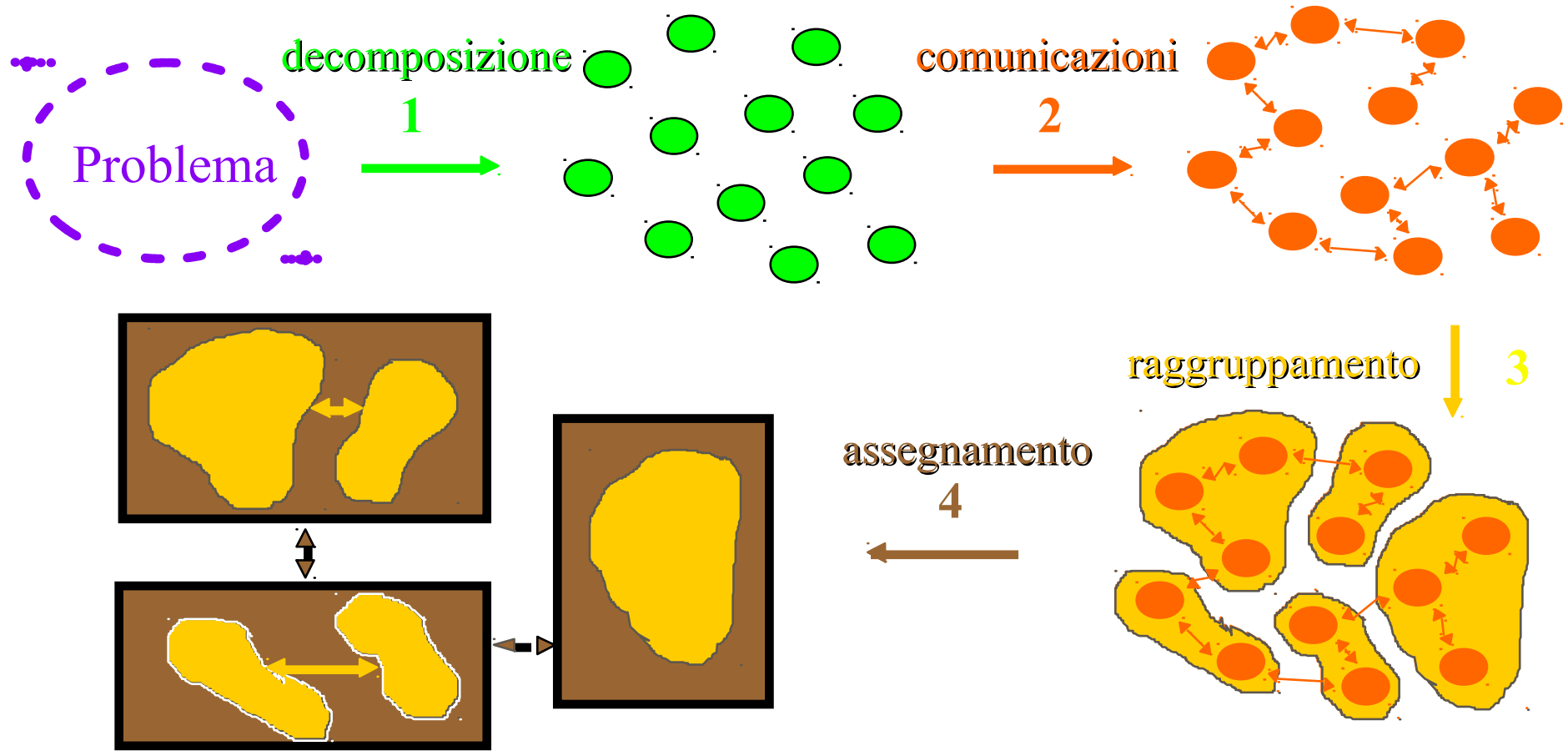
Confronto fra i due paradigmi



Analisi metodologica sulla programmazione parallela

- La metodologia più diffusa prevede i passi seguenti:
- Decomposizione (*Partitioning*):
 - i calcoli che devono essere fatti e i dati su cui opera l'algoritmo vengono decomposti in piccoli task. In tale fase non viene preso in considerazione il numero di processori che poi effettuano i calcoli, ma l'attenzione è focalizzata a riconoscere quanto più possibile i moduli parallelizzabili.
- Comunicazioni (*Communication*):
 - vengono stabilite le comunicazioni necessarie a coordinare i vari task e si decide quali sono le strutture e gli algoritmi di comunicazione.
- Raggruppamento (*Agglomeration*):
 - le strutture dei task e delle comunicazioni definite nei passi precedenti vengono valutate in termini di prestazioni e di costi. Se necessario i task vengono conglobati in task più grandi per raggiungere le prestazioni richieste o per abbassare i costi di sviluppo.
- Assegnamento (*Mapping*):
 - ogni task è assegnato al singolo processore in modo da soddisfare il più possibile due aspetti antagonisti: massimizzare l'uso dei processori e minimizzare i costi di comunicazione. L'assegnamento può essere fatto staticamente o determinato in corso di esecuzione attraverso gli algoritmi di bilanciamento del carico.

Il calcolo parallelo (o distribuito)



La Programmazione parallela

- ❑ La programmazione parallela comporta:
 - **Decomporre** un algoritmo o un insieme di dati in varie parti
 - **Raggruppare** le parti in lavori che possono essere portati avanti da più processori simultaneamente
 - **Assegnare** le parti di lavoro ai vari processori
 - **Coordinare** il lavoro e le comunicazioni dei processori coinvolti

- ❑ I vari tipi di programmazione parallela vengono determinati da:
 - il tipo di architettura parallela su cui si lavora
 - il tipo di comunicazione tra processori che viene utilizzata

Analisi metodologica



- ❑ Oltre ai passi suddetti occorre prevedere le seguenti fasi:
- ❑ Test
- ❑ Controllo e correzione degli errori (Debugging)
- ❑ Misura delle prestazioni

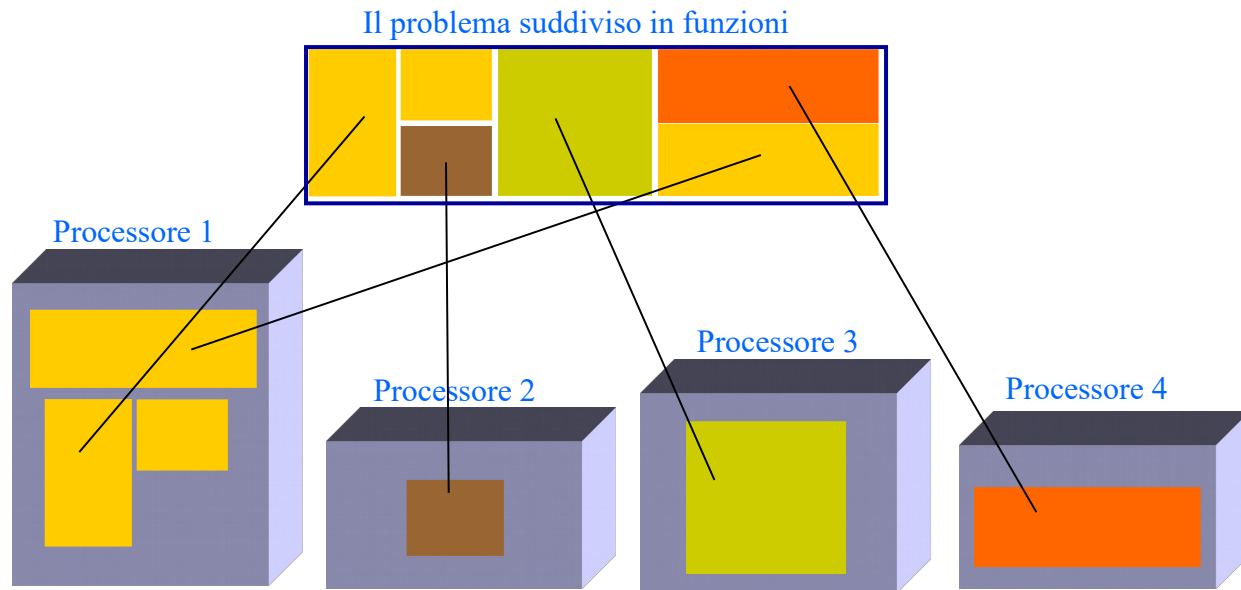
Decomposizione



- ❑ Esistono tre tipi di decomposizione di un problema in task più piccoli da effettuare in parallelo:
 - **decomposizione funzionale** (parallelismo funzionale)
 - **decomposizione dei domini** (parallelismo dei dati)
 - **decomposizione combinata** (una combinazione di entrambe)

- ❑ Una buona decomposizione suddivide in piccole parti sia la struttura dati su cui opera sia l'algoritmo da eseguire.

Decomposizione funzionale



- ❑ L'algoritmo di calcolo viene suddiviso in task disgiunti che sono in generale indipendenti dalle strutture dati e possono essere assegnati ai vari processori per esecuzioni simultanee.
- ❑ Ne è consigliato l'utilizzo in presenza di strutture non statiche o quando si possono predeterminare il numero di calcoli che devono essere fatti.

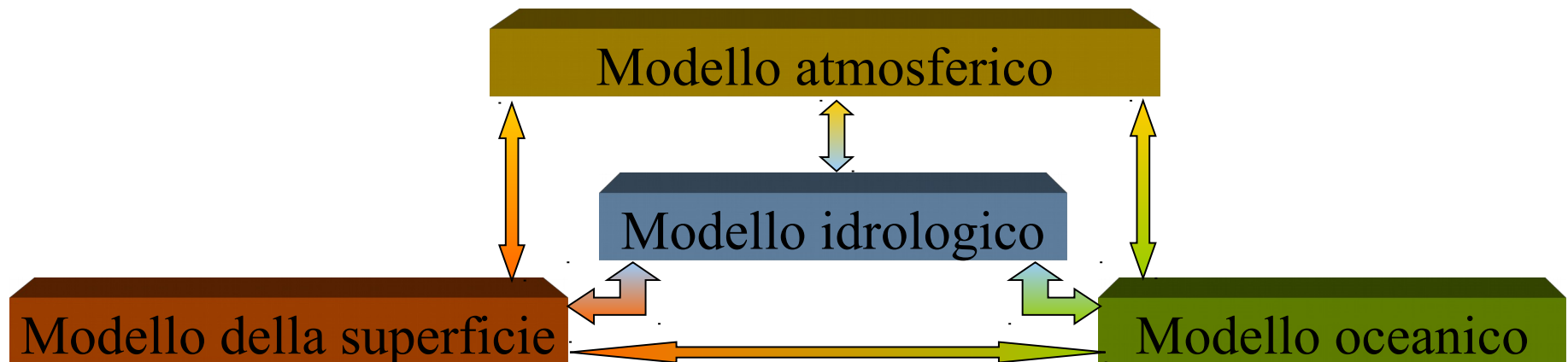
Decomposizione funzionale

- Nella fase di decomposizione funzionale si cerca di scomporre l'algoritmo di calcolo in vari task disgiunti, procedendo poi ad esaminare quali dati saranno associati a questi task. Se anche i dati sono disgiunti avremo una *decomposizione completa*, altrimenti si dovranno prevedere comunicazioni per passare i dati da un task all'altro. Se i dati sono molti bisognerà prendere in considerazione la successiva decomposizione dei domini, che potrebbe sostituire o integrare (**decomposizione combinata**) la decomposizione funzionale.

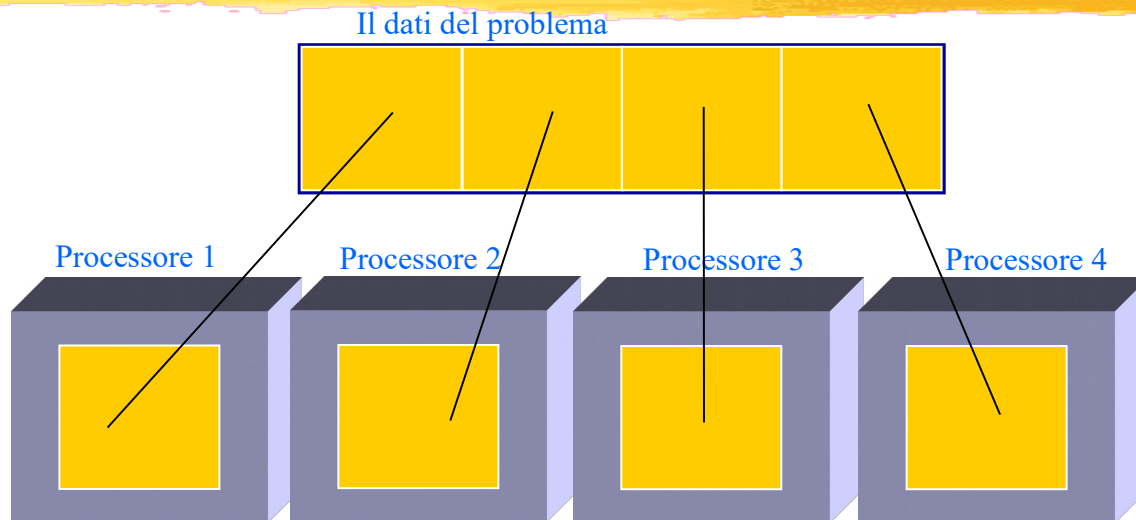
Decomposizione funzionale

□ Esempio:

- Consideriamo un problema di elaborazione di un modello climatico che può essere suddiviso in vari modelli. Ogni componente (task) può essere visto come un problema separato e quindi rianalizzato e decomposto al suo interno, anche con una decomposizione dei domini. Le frecce fra i vari componenti rappresentano scambi di dati durante i calcoli attraverso comunicazioni (per esempio la velocità del vento generata dal modello atmosferico viene usata dal modello oceanico, di rimando la temperatura superficiale del mare può essere utilizzata dal modello atmosferico).



Decomposizione dei domini



- L'attenzione è rivolta inizialmente ai dati determinandone una appropriata partizione, successivamente i dati vengono associati all'algoritmo di calcolo.
- Da usare in presenza di:
 - dati statici (factoring e risoluzione di grandi matrici o calcoli di differenze finite),
 - strutture dati dinamiche legate a singole entità, dove le entità possono essere determinate (large multi-body problems),
 - il dominio è fisso, ma il calcolo entro varie regioni del dominio è dinamico (fluid vortices models).

Decomposizione dei domini



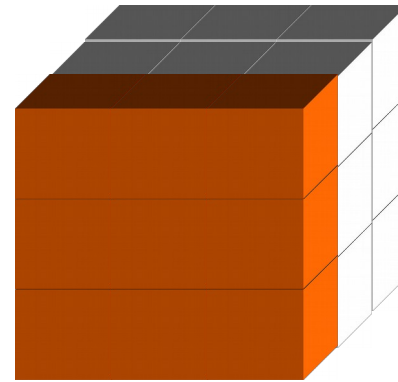
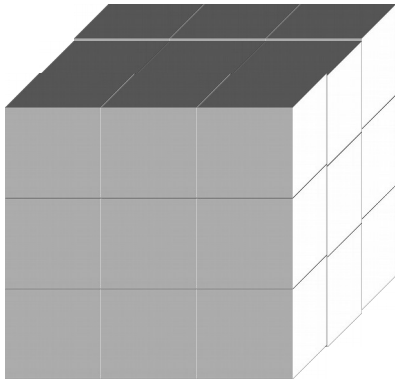
- ❑ Si partizionano i dati associati al problema in esame in blocchi di circa le stesse dimensioni e a ciascun blocco sarà associata una parte di codice che opera su di esso.
- ❑ Questa partizione determina un certo numero di task, ognuno dei quali comprende sia i dati che un certo numero di operazioni su di esse. Determinate operazioni possono però richiedere dati da vari task, determinando l'esigenza di introdurre comunicazioni tra i task.

Decomposizione dei domini

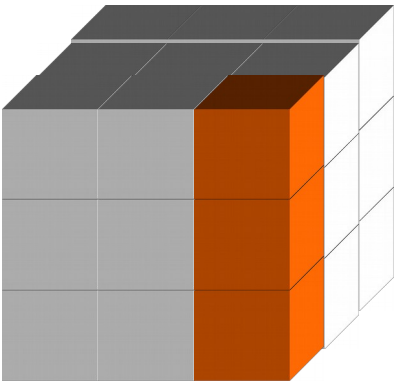
□ Esempio:

- Supponiamo di considerare un problema di analisi e calcolo di uno spazio tridimensionale di dati (modello atmosferico, elaborazione di immagini, ecc.). E' possibile decomporre lo spazio lungo gli assi cartesiani, naturalmente dobbiamo raggruppare per intervalli lungo gli assi perché una decomposizione per punti sarebbe impraticabile. Di seguito vengono illustrati tre tipi di decomposizione:
 - ✧ 1-D (decomposizione lungo un solo asse),
 - ✧ 2-D (decomposizione lungo due assi),
 - ✧ 3-D (decomposizione lungo tre assi, la più forte e la più usata perché offre una grande flessibilità specialmente nei primi stadi di analisi).

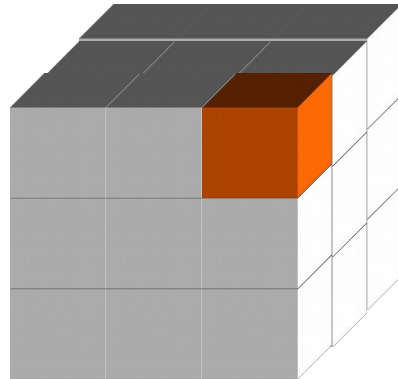
Decomposizione dei domini



1-D



2-D



3-D

Sistemi distribuiti e paralleli

Comunicazioni

- ❑ I task generati da una decomposizione verranno eseguiti in maniera concorrente, ma in generale non in maniera indipendente, infatti spesso il risultato di un calcolo eseguito da un task può essere un input di un altro task. Esiste quindi uno scambio di dati tra i vari task. Questo flusso di informazioni viene specificato nella fase di analisi chiamata comunicazioni.
- ❑ La definizione di canali di comunicazione fra i vari task comporta però due tipi di costi: il primo di natura intellettuale dovuto all'appesantimento delle procedure, il secondo più propriamente fisico per l'invio e la ricezione delle informazioni

Comunicazioni



- ❑ Le comunicazioni fra task possono essere di vari tipi:
 - **locali o globali**
 - **strutturate o non strutturate**
 - **statiche o dinamiche**
 - **sincrone o asincrone**

Comunicazioni locali

□ In comunicazioni di tale tipo il generico task richiede dati soltanto da un piccolo numero di task (i vicini).

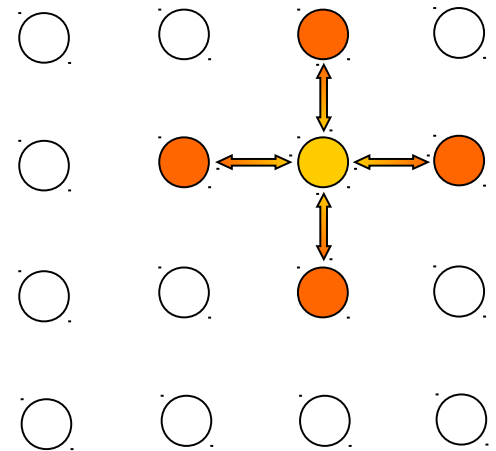
□ Esempio

○ metodo delle differenze finite

$$X_{ij}^+ = \frac{4X_{ij} - X_{i-1,j} - X_{i+1,j} - X_{i,j-1} - X_{i,j+1}}{4}$$

✧ se ad ogni nodo associamo un task si ha:

- for t=0 to t-1
- invia X_{ij} a tutti i vicini
- ricevi $X_{i-1,j}, X_{i+1,j}, X_{i,j-1}, X_{i,j+1}$ da tutti i vicini
- calcola X_{ij}^+ (usando l'equazione sopra)
- fine

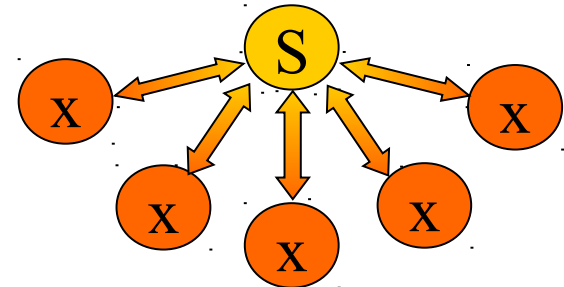


Comunicazioni globali

- In questo caso i dati vengono distribuiti tra molti task all'interno dell'algoritmo.
- Questo tipo di comunicazioni viene anche chiamato **Fattoria di processi**.
- Esempio

- metodo delle differenze finite

$$S = \sum_{i=1}^N x_i$$

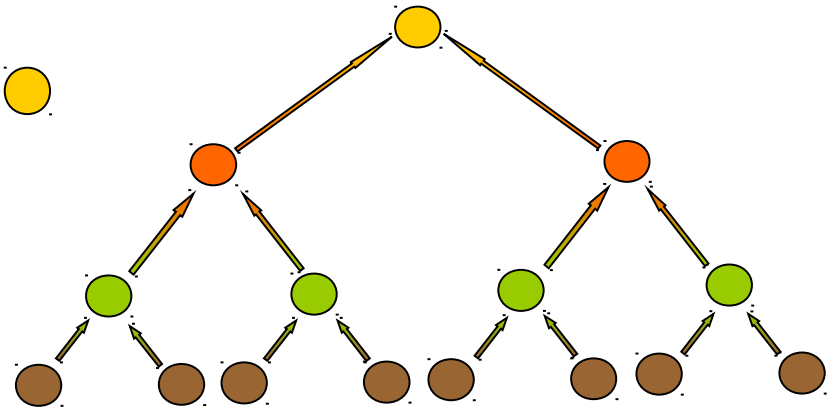
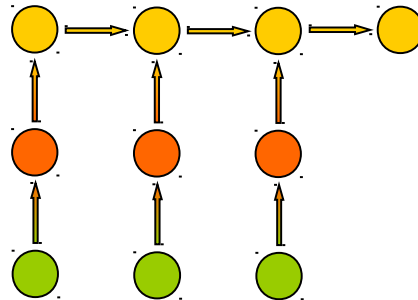
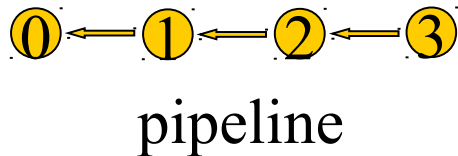


- Se tutti i task inviano i dati al master **S** e questi processa un dato alla volta, non si hanno vantaggi in quanto la scrittura di elaborazione è sequenziale e non concorrente
- E' necessario non solo distribuire il calcolo, ma anche le comunicazioni, in modo opportuno

Comunicazioni strutturate e non strutturate

- In comunicazioni **strutturate** i task eseguono calcoli parziali e la sequenza del calcolo segue la sequenza dei processori interessati, ossia l'output prodotto da un processore è input per il processore successivo nella struttura.

- Esempio:



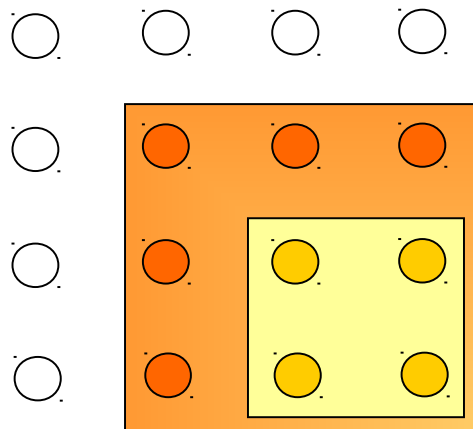
- Un caso di comunicazioni **non strutturate** si può avere quando la segmentazione del problema non è regolare e può cambiare durante il calcolo (es: metodo delle differenze finite).

Comunicazioni sincrone e asincrone

- ❑ Le comunicazioni **sincrone** sono caratterizzate dal fatto che avvengono a specifici istanti di tempo.
 - ❑ Le comunicazioni **asincrone** (più diffuse) sono comunicazioni che avvengono su richieste esplicite di uno o entrambi i processori.
 - Esempio:
 - ✧ Il classico problema del produttore e del consumatore: non si conosce quando i dati saranno necessari, quindi i dati dovranno essere richiesti esplicitamente.
 - ✧ La situazione si verifica quando si ha una struttura dati condivisa che non può essere contenuta in ciascun task, con differenti task che accedono alla struttura indipendentemente. Le soluzioni possibili sono:
 - Il singolo task effettua il calcolo con i dati a sua disposizione e richiede dati agli altri task interrompendosi periodicamente per soddisfare le richieste di dati da altri task a lui pendenti
 - Viene definito un gruppo di task a cui è affidato il compito di gestire unicamente la lettura e la scrittura dei dati
- La prima struttura è meno modulare e il tempo speso per comunicare i dati influisce sulle prestazioni. La seconda è più modulare, ma non sfrutta la località dei dati.

Raggruppamento

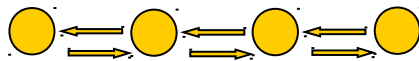
- Con il raggruppamento si combinano più task individuati nella fase di segmentazione così da ottenere un numero inferiore di task di maggiore dimensione, allo stesso tempo si indaga se sia meglio replicare i dati e/o il loro calcolo.
- **Effetti volume-superficie:**
 - se le comunicazioni per task sono poche si può aumentare il numero di comunicazioni aumentando la *granularità* della partizione.



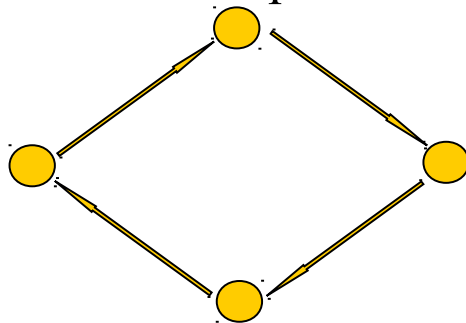
Raggruppamento

□ Calcolo replicato:

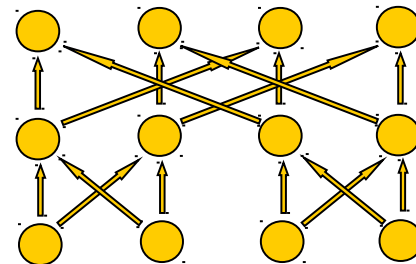
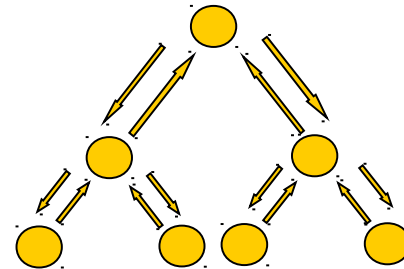
- A volte risulta conveniente il calcolo concorrente della stessa grandezza da parte di più task.
- Esempio: consideriamo il caso in cui la somma di n numeri debba essere distribuita su N task.



Calcolo replicato



anello



butterfly

Mapping

- ❑ Eseguita una opportuna decomposizione del problema ed analizzate le strutture di comunicazione si dovranno assegnare i task ai differenti processori (mapping).
- ❑ L'obiettivo del mapping è minimizzare il tempo di esecuzione.
 - Principi:
 - ✧ I task che vengono seguiti in maniera concorrente sono posti su processori differenti.
 - ✧ I task che comunicano frequentemente risiedono sullo stesso processore.
- ❑ Metodi impiegati:
 - Load balancing (bilanciamento del carico)
 - Task scheduling (metodo di schedulazione dei lavori)

Misura delle prestazioni

- Due importanti misure della qualità di un algoritmo parallelo sono:
 - lo "speedup"
 - l'efficienza

Definizione di SPEEDUP e di EFFICIENZA

- Sia T_s il tempo necessario per far girare il più veloce algoritmo seriale su un processore e sia T_p il tempo necessario per far girare l'equivalente algoritmo parallelo su N processori, si definisce

$$\text{SPEEDUP} = T_s / T_p$$

- mentre l'efficienza dell'algoritmo parallelo sarà data da:

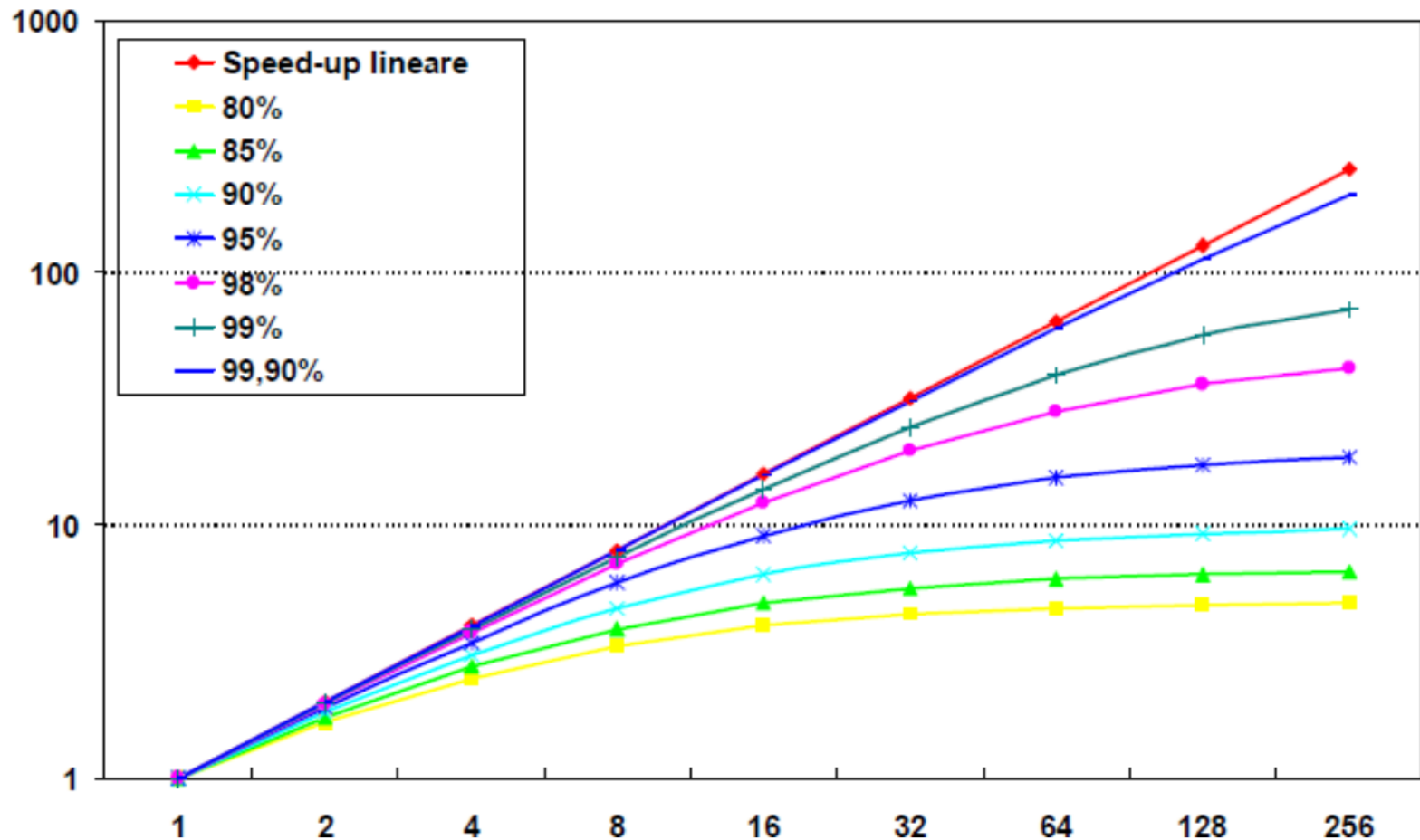
$$\text{EFFICIENZA} = \text{SPEEDUP} / N$$

Speedup ed efficienza

Un'altra definizione corrente di SPEEDUP è la seguente:

- ❑ il tempo necessario per far girare un algoritmo parallelo su un solo processore diviso per il tempo necessario a far girare lo stesso algoritmo su N processori.
- ❑ Qualsiasi definizione si usi l'ideale è di produrre uno "speedup" lineare cioè produrre uno speedup uguale a N , l'efficienza in questo modo sarà uguale a 1 cioè al 100%.
- ❑ Nella realtà però lo speedup è sempre inferiore ad N come 1 resta un limite superiore per l'efficienza.

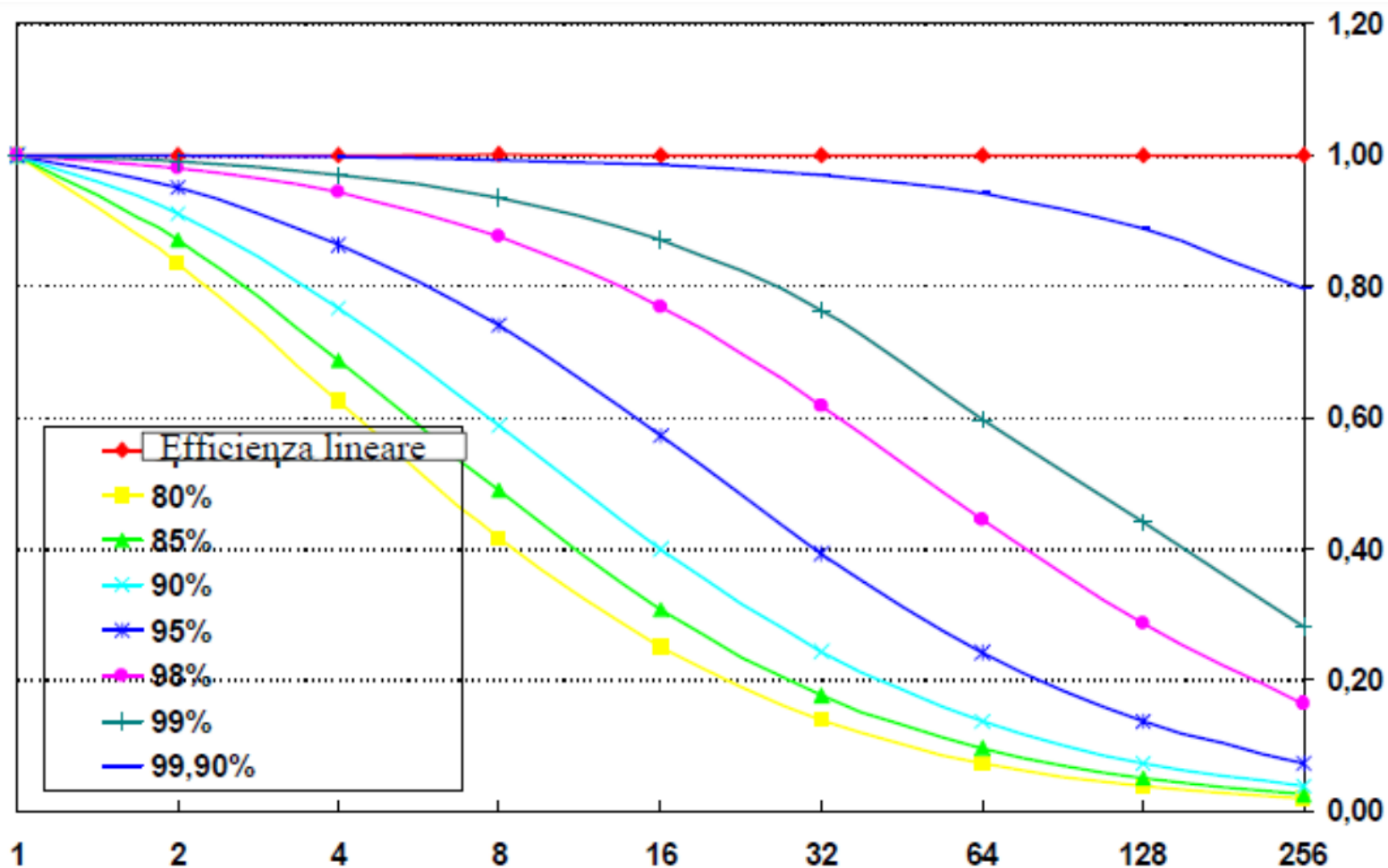
Speedup



Sistemi distribuiti e paralleli

128

Efficienza



Sistemi distribuiti e paralleli

Fattori che limitano lo speedup

❑ Overhead del software

- Maggior codice prodotto nella versione parallela di un programma.

❑ Bilanciamento del carico

- Lo speedup è generalmente limitato dalla velocità del nodo più lento. Quindi è importantissimo accertarsi che ogni nodo (processore) abbia da effettuare la stessa mole di lavoro, cioè che tutto il sistema sia bilanciato nel carico.

❑ Overhead della comunicazione

- Durante la comunicazione i processori non portano avanti i calcoli quindi questo degrada lo speedup .
- Per questo motivo chi costruisce algoritmi paralleli dovrebbe cercare di mantenere una grana più grossa possibile.
- Gli effetti della comunicazione tra processori sullo speedup si riducono, in termini relativi, all'aumentare della dimensione della grana.

Parametri e prestazioni

- Un altro importante parametro dipendente dalla macchina è la frazione seguente:

$$t_{comm}/t_{calc}$$

dove t_{comm} è il tempo necessario a trasferire una “parola” tra 2 nodi e t_{calc} è il tempo necessario ad effettuare una operazione in virgola mobile.

La legge di Amdahl

□ La legge di Amdahl stabilisce che lo speedup di un algoritmo parallelo è effettivamente limitato dal numero di operazioni che devono essere effettuate sequenzialmente, cioè dalle sue frazioni seriali

□ Sia S l'ammontare del tempo speso (da un processore) nelle parti seriali del programma e P l'ammontare del tempo speso (da un processor) nelle parti del programma che potrebbero essere fatte in parallelo

cioè $T_{\text{seq}} = S + P$ e $T_{\text{par}} = S + \frac{P}{N}$
dove N è il numero di processori

Allora lo speedup = $\frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{S + P}{S + \frac{P}{N}}$

Legge di Amdahl

□ Esempio

Se abbiamo un programma che contiene 100 operazioni ognuna delle quali impiega 1 unità di tempo e se 80 operazioni possono essere fatte in parallelo ($P=80$), mentre 20 devono essere fatte sequenzialmente ($S=20$), allora usando 80 processori si ha:

$$\text{speedup} = 100 / (20 + 80/80) = 100/21 < 5$$

cioè è possibile avere uno speedup di solo 5, non importa quanti siano i processori disponibili!

Frazione seriale

- Se definiamo la frazione seriale F come:

$$F = \frac{S}{S+P} \quad P = \frac{S}{S+P}$$

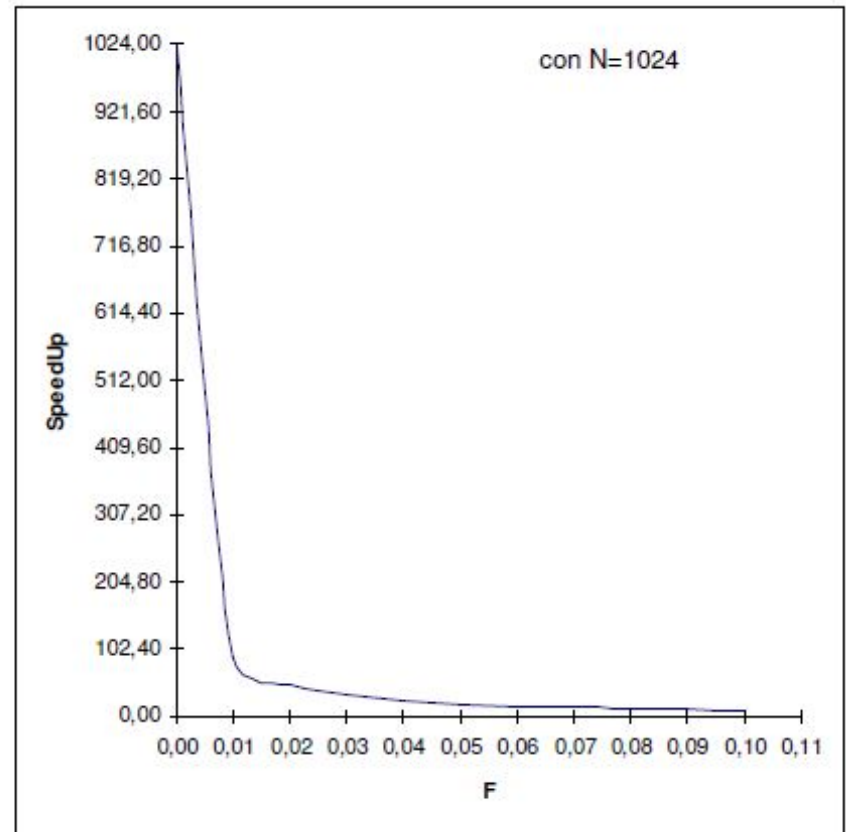
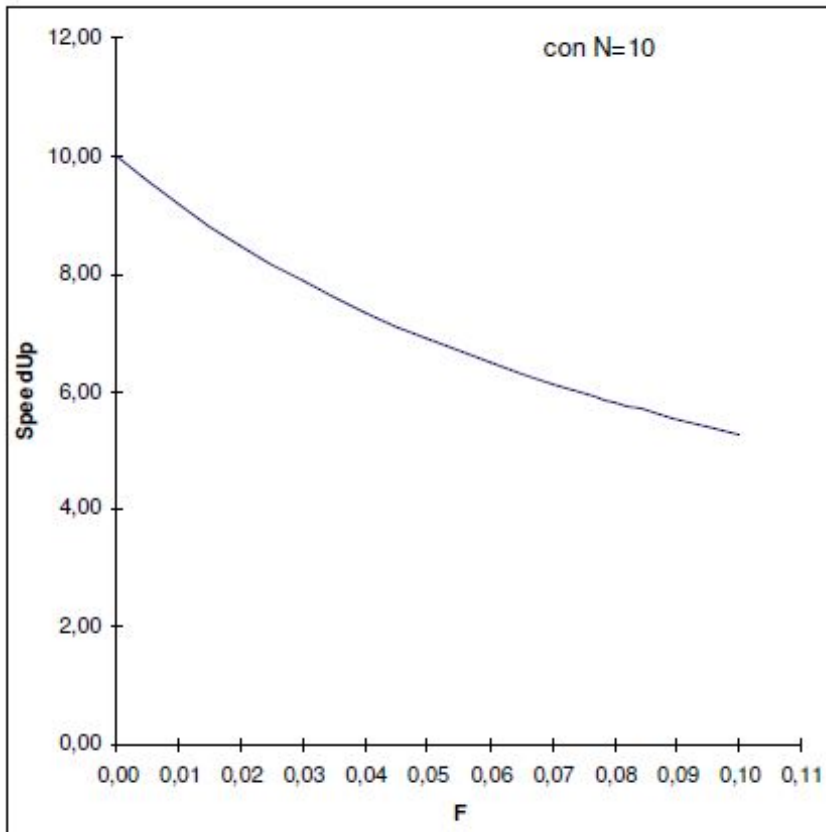
Allora l'equazione può essere scritta come:

$$\text{speedup} = \frac{N}{F + \frac{F}{N}}$$

- così se $F = 0$ (cioè non ci sono parti seriali) allora lo speedup = N (che è il valore attuale)
- se $F = 1$ (cioè è completamente seriale) allora lo speedup = 1 (solo 1 processore può essere usato)

Speedup e frazione seriale

- Se consideriamo l'andamento dello speedup in funzione della frazione seriale F per $N=10$ e per $N=1024$ si ha:



Speedup e frazione seriale

Cioè se l'1% di un programma parallelo è formato da codice seriale, il massimo speedup che si può ottenere usando 10 processori è 9, mentre usando 1024 processori è solo 91.

- ❑ **N.B.** Quindi la legge di Amdahl dice come la frazione seriale F ponga un forte limite allo speedup all'aumentare del numero dei processori.
- ❑ Comunque molte applicazioni parallele richiedono frazioni seriali molto piccole <0.001 .

Speedup e frazione seriale

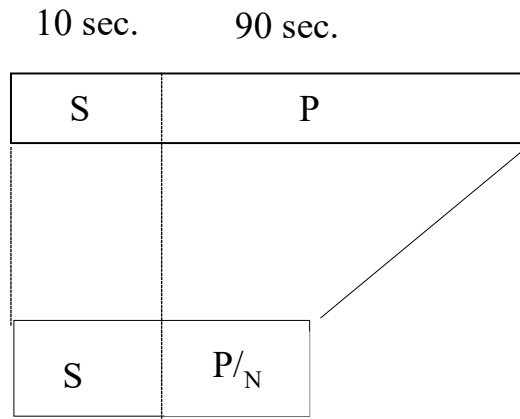
- Se noi abbiamo un programma che impiega T_{seq} secondi a girare di cui S secondi nella parte del programma che deve essere fatta serialmente e P ($=T_{seq} - S$) secondi nella parte che può essere fatta in parallelo

cioè $T_{seq} = S + P$ e se $F = \frac{S}{T_{seq}}$ è la frazione seriale

allora noi potremo risolvere lo stesso problema su una macchina parallela, cioè la dimensione del problema è fissata, quindi lo sviluppo può essere calcolato dalla legge di Amdahl così :

$$\text{Speedup} = \frac{S + P}{S + \frac{P}{N}} = \frac{1}{F + \frac{1-F}{N}}$$

Speedup e frazione seriale



- S è lo stesso poichè la dimensione del problema è fissa, se aumentiamo la dimensione del problema S potrebbe aumentare.

Speedup e frazione seriale

Comunque potremmo dire che

- il "run time" è fisso cioè la dimensione del problema non è fissa ma aumenta in proporzione a N .

E

- la frazione seriale F non è costante ma diminuisce se diminuisce la dimensione del problema cioè S è costante.

- Quindi la legge di Amdahl dice che la frazione seriale F pone un forte vincolo allo speedup all'aumentare del numero di processori.

Speedup e frazione seriale

- ❑ Poichè molti programmi paralleli contengono una certa parte di codice sequenziale la conclusione della legge di Amdahl è che non conviene costruire sistemi con un grande numero di processori, poichè non si riuscirà a produrvi un sufficiente speedup
- ❑ La legge di Amdahl è valida per problemi in cui la frazione seriale F non varia con la dimensione del problema cioè quando la dimensione del problema aumenta, il tempo T_{seq} e S aumentano, lasciando però $F = S/T_{seq}$ costante.
- ❑ Applicazioni che parallelizzano bene sono quelle con frazioni seriali molto piccole.

Uso della frazione seriale nella misura delle prestazioni

- Dati la frazione seriale F e il numero di processori N si può calcolare lo speedup previsto usando l'equazione:

$$\text{speedup} = \frac{1}{F + \frac{F}{N}}$$

- Se poi facciamo girare il programma e troviamo lo speedup reale dalla:

$$\text{speedup} = \frac{T_{seq}}{T_{par}}$$

- potremo rigirare lo speedup = $\frac{1}{F + \frac{F}{N}}$ per trovare la frazione seriale reale F :

$$F = \frac{1 - \frac{1}{\overline{\text{speedup}}}}{\overline{N}}$$

- Il valore di F è utilissimo poichè l'eq. $T_{\text{par}} = S + \frac{P}{N}$ è ideale, in effetti si assume che tutti i processori lavorino a pieno nello stesso tempo cioè siano perfettamente bilanciati.
- L'"overhead" nella comunicazione e nella sincronizzazione dei processori non è incluso.
- Gli oggetti del bilanciamento del carico sono probabilmente mostrati da un cambiamento irregolare di F quando N aumenta.
- L'"overhead" della comunicazione e sincronizzazione tende ad aumentare all'aumentare di N .
- Siccome aumentando l'"overhead" diminuisce lo "speedup" il valore di F pian piano aumenta all'aumentare di N .
- Così un lento aumentare di F è un avvertimento che la grana è troppo piccola.

Esempio

- Consideriamo i risultati seguenti:

N	Speedup	Efficienza	F
2	1.95	97	0.024
3	2.88	96	0.021
4	3.76	94	0.021
8	6.96	87	0.021

- Senza guardare la frazione seriale F non possiamo dire se i risultati sono buoni o no.

- Perchè l'efficienza diminuisce?

Siccome F è circa costante possiamo concludere che questo è dovuto al parallelismo limitato del programma.

Il bilanciamento del carico

- ❑ Il problema del bilanciamento del carico si riferisce alla distribuzione dei vari task ai vari processori in modo tale da assicurare l'esecuzione parallela nel minor tempo possibile.
- ❑ Se i tasks non vengono distribuiti in modo bilanciato si può verificare il caso in cui si resti in attesa della fine di un task quando tutti gli altri hanno già finito.
- ❑ Esistono due tipi di bilanciamento del carico:
 - **Assegnazione statica dei task**
 - **Assegnazione dinamica dei task**

Assegnazione statica dei task

- ❑ Assegnazione statica dei task significa che la distribuzione dei lavori ai vari processori viene effettuata una volta per tutte fino alla fine del tempo di esecuzione.
- ❑ Questo può essere fatto in diversi modi:
 - il programmatore usa il buon senso per bilanciare il sistema, per esempio assicurando un egual numero di operazioni sui dati in ogni processore, o decidendo quali task vengano affidati ai singoli processori, assicurando in questo modo un bilanciamento piuttosto grossolano.
 - viene utilizzato un algoritmo, per es. la bisezione, per trovare una buona distribuzione di task ai processori prima di far girare l'algoritmo parallelo.

Assegnazione dinamica dei task

- Inizialmente viene effettuata una distribuzione di task (o di dati) tra i processori, poi se il sistema incontra degli sbilanciamenti viene fatta una ridistribuzione per ottimizzare i tempi.
- I bilanciatori dinamici del carico monitorizzano continuamente il carico nei vari processori in modo da ridistribuire i lavori non appena incontrano uno sbilanciamento di un certo "livello".

Esempio

- Per esempio, se consideriamo che in un sistema a 2 processori il carico dopo un certo tempo di esecuzione sia il seguente:

A : 100

B : 60

- cioè sia presente uno sbilanciamento, allora il bilanciatore del carico deciderà di far transitare 20 unità di carico dal processore A al processore B

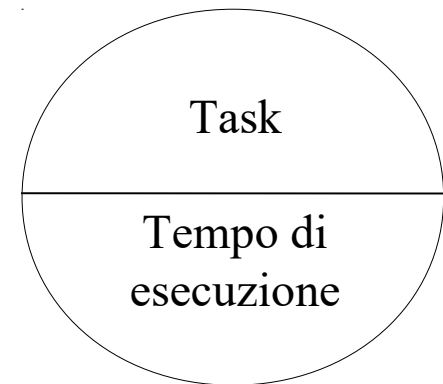
	prima	DLB	dopo
A	100	20	80
B	60	20	80

- Senza invocare il bilanciatore del carico avremo dovuto attendere l'esecuzione di 100 unità di carico, con il bilanciatore basta attendere l'esecuzione di 80

Rappresentazione grafica dei task

- Un grafo di task è un grafo in cui ogni nodo rappresenta un task che deve essere eseguito.

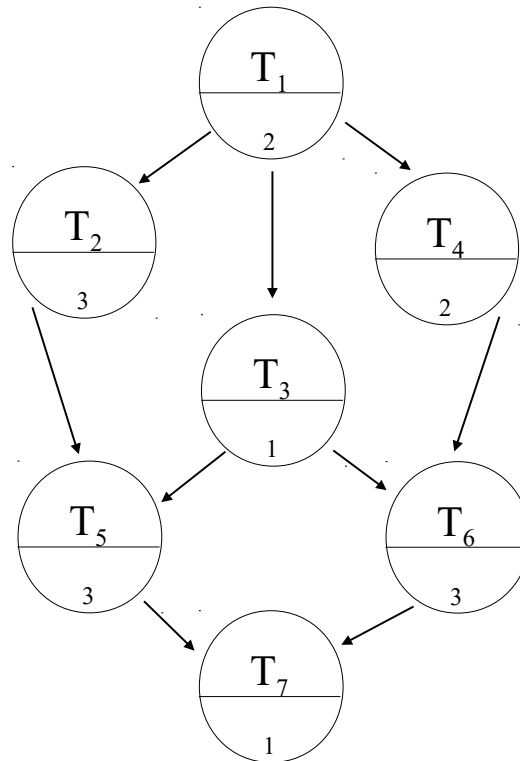
- Ogni nodo generalmente è così fatto



- Un arco diretto dal nodo T_i al nodo T_j indica che il task T_i deve essere completato prima che inizi il task T_j .

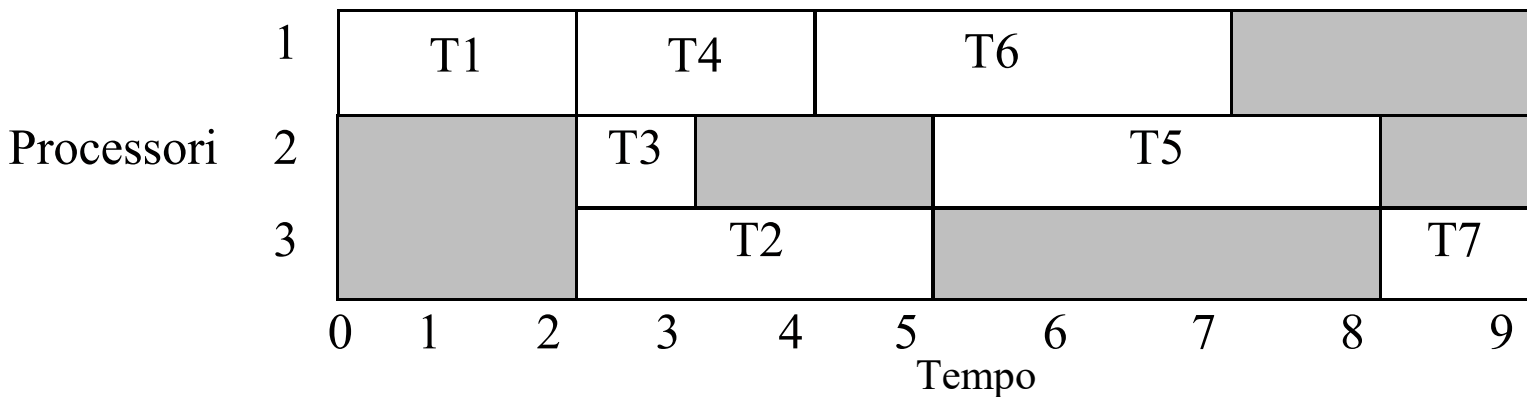
Esempio

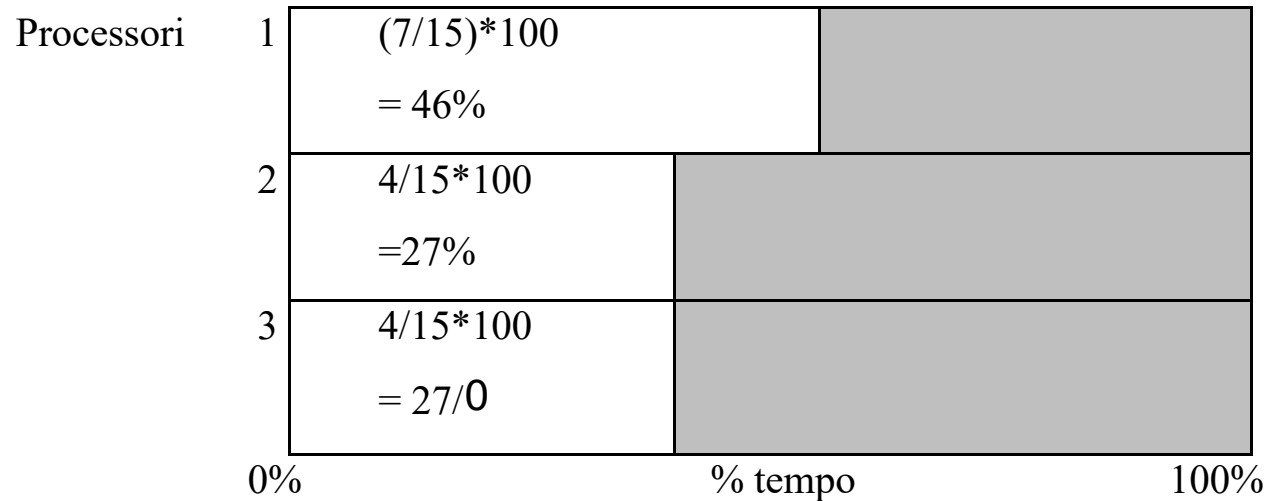
□ Consideriamo il grafico seguente:



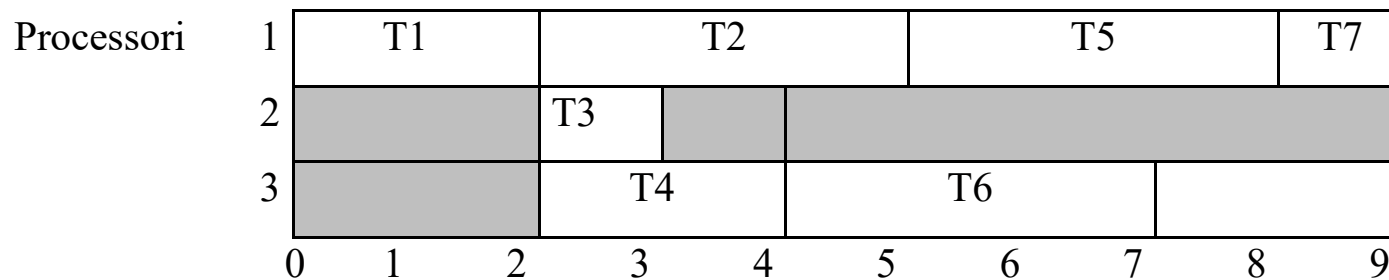
Assegnano questi task ad un sistema di 3 processori.

- L'assegnazione di task ai processori è anche illustrata da un grafico di Gontt che indica il tempo che ogni task spende in esecuzione su quel certo processore (i grafici di Gontt vengono anche usati per illustrare l'utilizzo di ogni processore, cioè in percentuale rispetto al tempo spesso nell'eseguire i task).
- Un grafico di Gontt per il grafico di task visto sopra è il seguente:

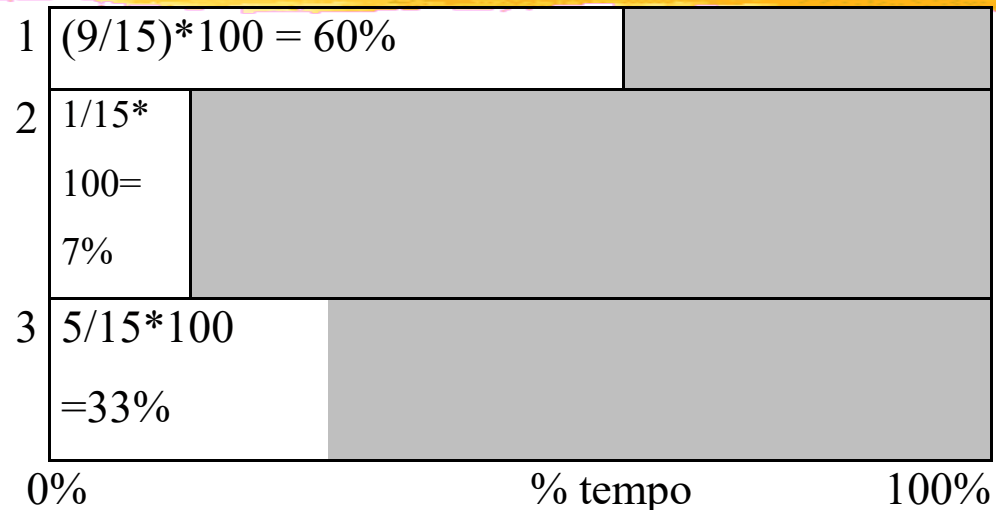




l'assegnazione ottima dei task può non essere unica, infatti

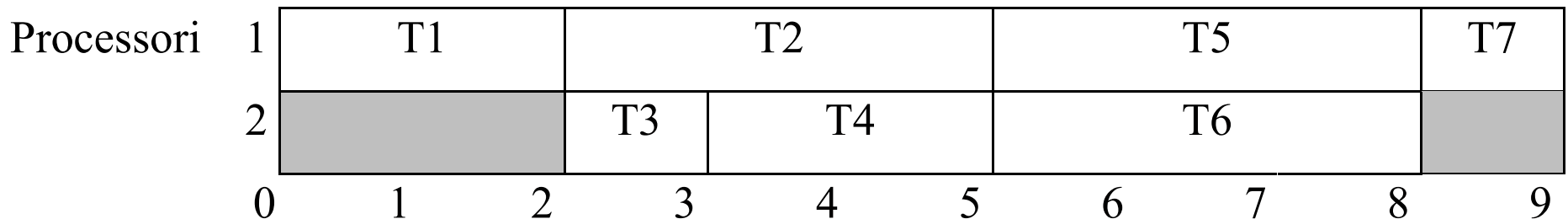


Processori



in cui il tempo totale di esecuzione è di nuovo di 9 unità

Per questo particolare esempio il più alto grado di efficienza può essere ottenuto usando solo 2 processori, infatti:





che ha un tempo di esecuzione totale uguale a quello con 3 processori.

L'efficienza nel caso dei 2 processori è quindi:

$$\text{efficienza} = \text{speedup} / N = (\text{Tempo tot. Seriale su 1 proc.} / \text{Tempo tot. su N proc.}) / N = \\ = (15/9) / 2 = 15/18$$

in termini di percentuali: $\text{efficienza} = (15/18) * 100 = 83\%$

mentre con 3 processori è:

$$\text{efficienza} = ((15/9) / 3) * 100 = (15/27) * 100 = 55\%$$

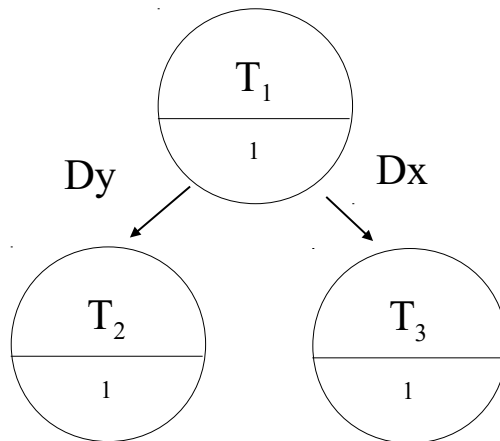
Nell'esempio sopra il tempo di esecuzione di ogni task è conosciuto prima di fare l'assegnamento, questo potrebbe non essere realistico, e si ignorano eventuali variazioni al tempo di esecuzioni dei task.

Ritardi della comunicazione nel parallelismo

Nell'esempio fatto prima venivano completamente ignorate tutte le comunicazioni tra task dipendenti.

Ma in molti problemi i ritardi della comunicazione possono avere sensibili effetti su come i task vengono assegnati ai processori.

Consideriamo per es. il grafico ad albero seguente:



$D_y > D_x$
il tempo sequenziale è di 3 unità

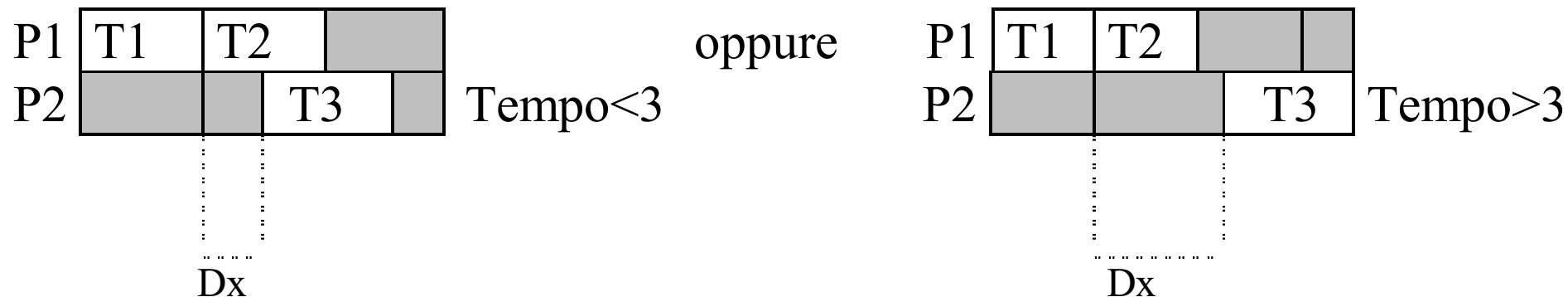
dove D_x è il ritardo della comunicazione nel passaggio dei dati/risultati tra T_1 e T_3 .


- Se noi abbiamo 2 processori, ignorando il ritardo della comunicazione, la schedulazione ottima sarebbe:

P1	T1	T2
P2		T3

0 1 2

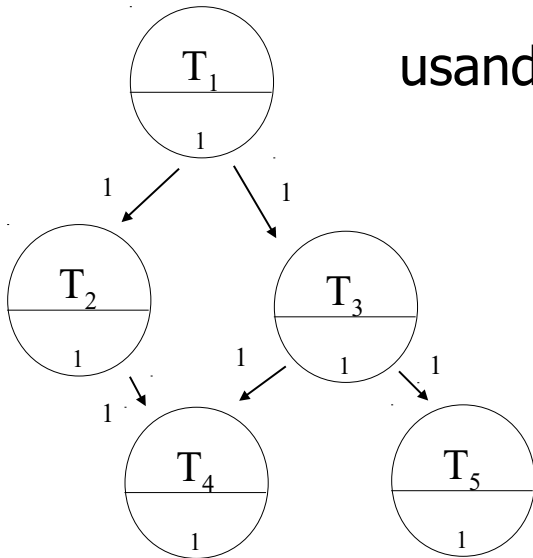
- Considerando anche i ritardi si hanno 2 possibilità:



- 
- ❑ Nel secondo caso addirittura il parallelismo implica un tempo (>3) maggiore di quello sequenziale.
 - ❑ Quindi esisterà uno scambio tra il vantaggio di massimizzare il parallelismo rispetto a minimizzare il ritardo della comunicazione.
 - ❑ Questo tipo di problema viene chiamato "max"-mini" per processi paralleli.
 - ❑ Un metodo che cerca di risolvere il problema "max"-mini" parte la duplicazione di task dove è necessario per ridurre il tempo di comunicazione e massimizzare il parallelismo allo stesso tempo.

Esempio

Se abbiamo il seguente grafico di task



usando 2 processori posso avere la seguente distribuzione

P1	T1	T2			T5
P2			T3	T4	
	0	1	2	3	4 5

Ma se duplichiamo il task T1 in entrambi i processori

P1	T1	T2		T4
P2	T1	T3	T5	
	0	1	2	3 4

otteniamo un tempo di esecuzione totale di 4 unità, questo perché l'istante di partenza del task T3 si riduce da 2 a 1.

Algoritmi per assegnazione dei task

- ❑ I primi tre algoritmi riportati sotto sono algoritmi statici, cioè algoritmi che prevedono il bilanciamento del carico prima dell'esecuzione e in cui il carico resta inalterato durante tutto il tempo di vita del programma.
 - **Algoritmo statico ottimale**
 - **Metodo di Bisezione**
 - **Algoritmo "greedy"** (avido, goloso)
- ❑ Alcune tecniche di bilanciamento del carico dinamico utilizzano i seguenti algoritmi:
 - **Cambio di dimensioni**
 - **Decomposizione sparsa**

Algoritmo statico ottimo

Passo 1

Questo algoritmo produce un partizionamento ottimo di un programma di tipo **pipeline** in una rete lineare di processori

Passo1

Disegna un grafico a "strati" in cui ogni strato corrisponde ad un processore ed ogni nodo (i,j) in uno strato corrisponde ad una sottocatena di task da i a j .

Siano N il numero dei processori e m il numero dei task.

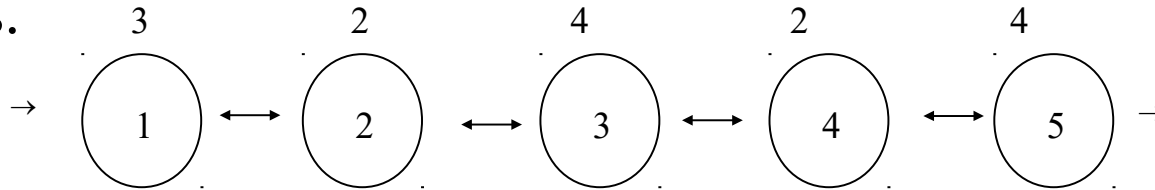
Lo strato 1 contiene nodi $(1,j)$ dove $j=1, \dots, m-(N-1)$

Gli strati $k=2, \dots, N-1$ contengono nodi (i,j) dove $i=k, \dots, m-(N-k)$,
 $j=i, \dots, m-(N-k)$

Lo strato N contiene nodi (i,m) dove $i=N, \dots, m$

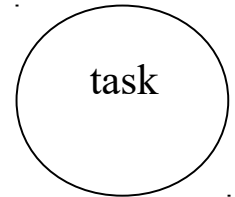
Passo 1

es.



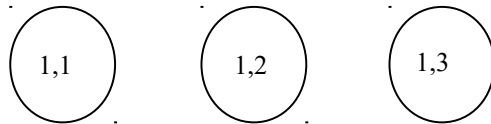
dove:

tempo richiesto
per il task



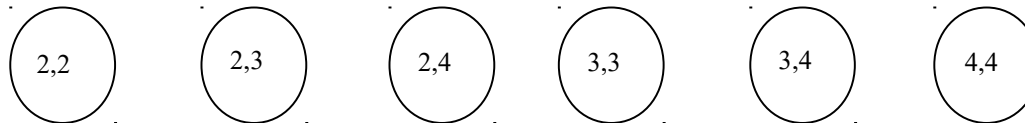
cioè $m=5$, $N=3$ Su 3 processori da:

P_1



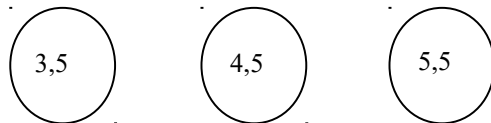
$j=1, \dots, 5-(3-1)=1, \dots, 3$

P_2



$i=2, \dots, 5-(3-2)=2, \dots, 4$
 $j=i, \dots, 4$

P_3



$i=N, \dots, m=3, \dots, 5$

Passo 2



Passo 2

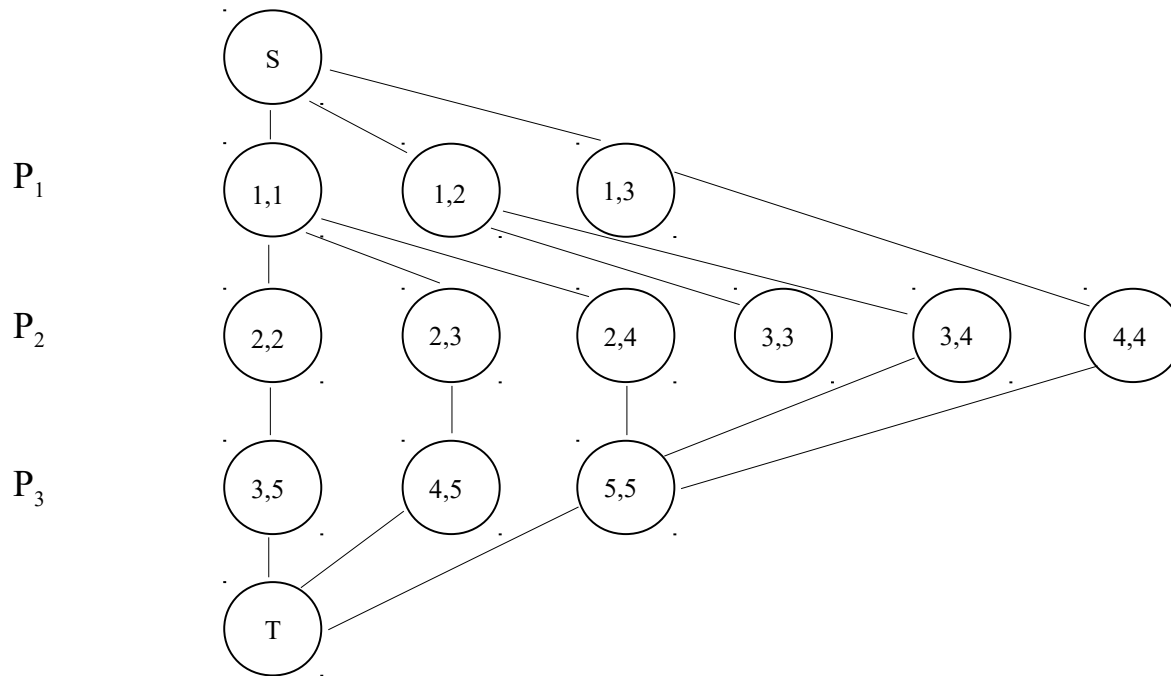
Un nodo (i,j) a livello $k=1,N-1$ è connesso a tutti i nodi $(j+1,q)$, per ogni q del livello $k+1$

Tutti i nodi $(1,j)$ nel primo strato sono connessi al nodo di partenza S

Tutti i nodi (i,m) dello strato finale sono connessi al nodo di terminazione T

Passo 2

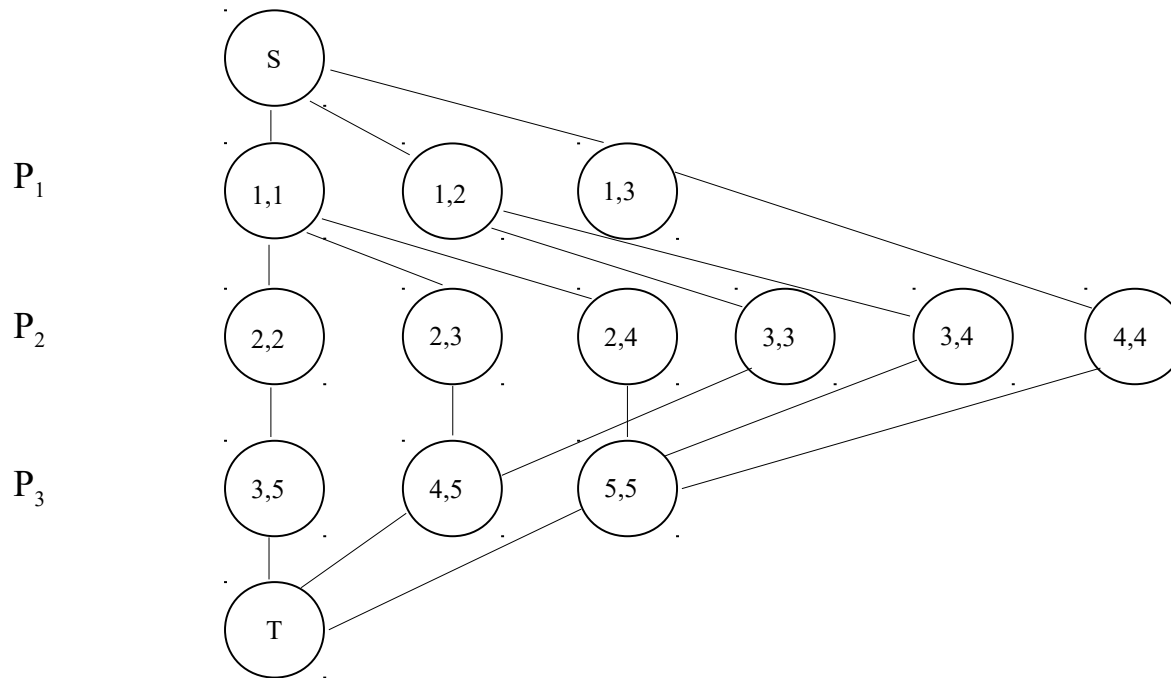
Es:



Ogni cammino che collega S a T corrisponde ad un assegnamento di Task ai processori.

Passo 2

Es:



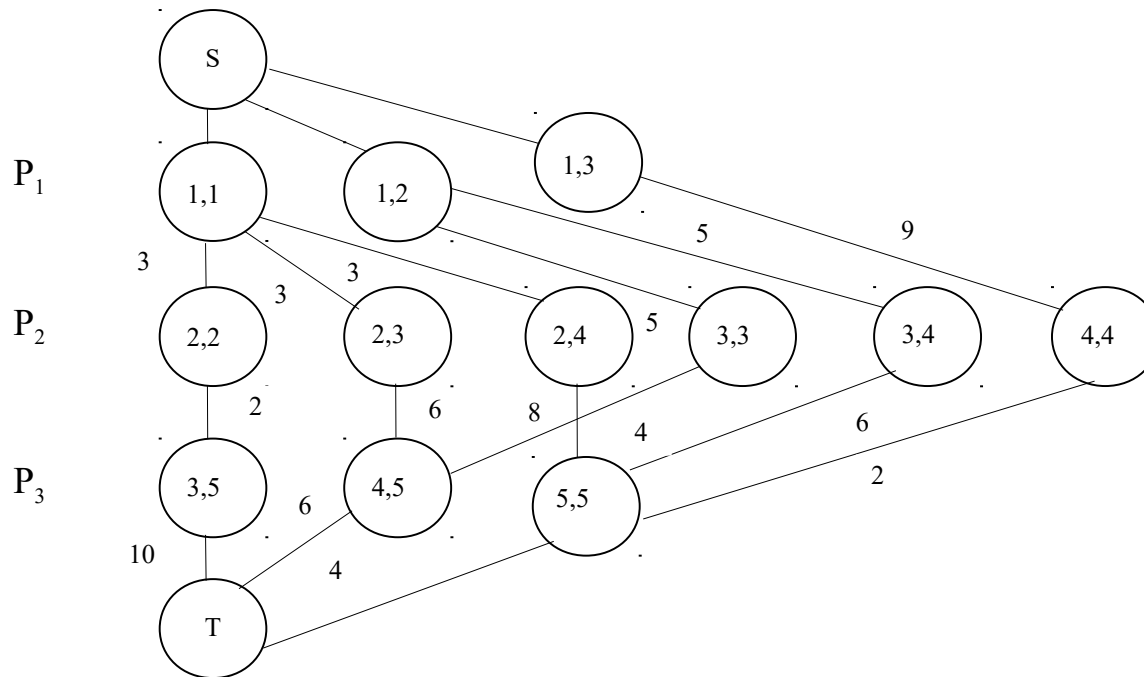
Ogni cammino che collega S a T corrisponde ad un assegnamento di Task ai processori.

Passo 3

Passo 3

Nello strato $k=1,\dots,N$ ogni arco in più dal nodo (i,j) è pesato con il tempo richiesto dal processore k a processare i task da i a j (e questo contribuisce al tempo di calcolo totale sul processore k)

es.



Passo 3



- ❑ I cammini in questo grafico rappresentano ogni possibile assegnamento di sottocatene contigue e il peso dell'arco più pesante in un cammino corrisponde al tempo richiesto al processore caricato più pesantemente.
- ❑ Quindi per trovare l'assegnamento ottimo occorre cercare il cammino in cui l'arco più pesante ha il peso minimo rispetto agli altri cammini, questo cammino è chiamato **cammino critico**.

Passo 4

Passo 4

Per trovare il cammino critico occorre operare così:

I nodi al primo strato vengono etichettati $L(i) = 0$, tutti gli altri nodi sono etichettati inizialmente a $L(i) = \infty$

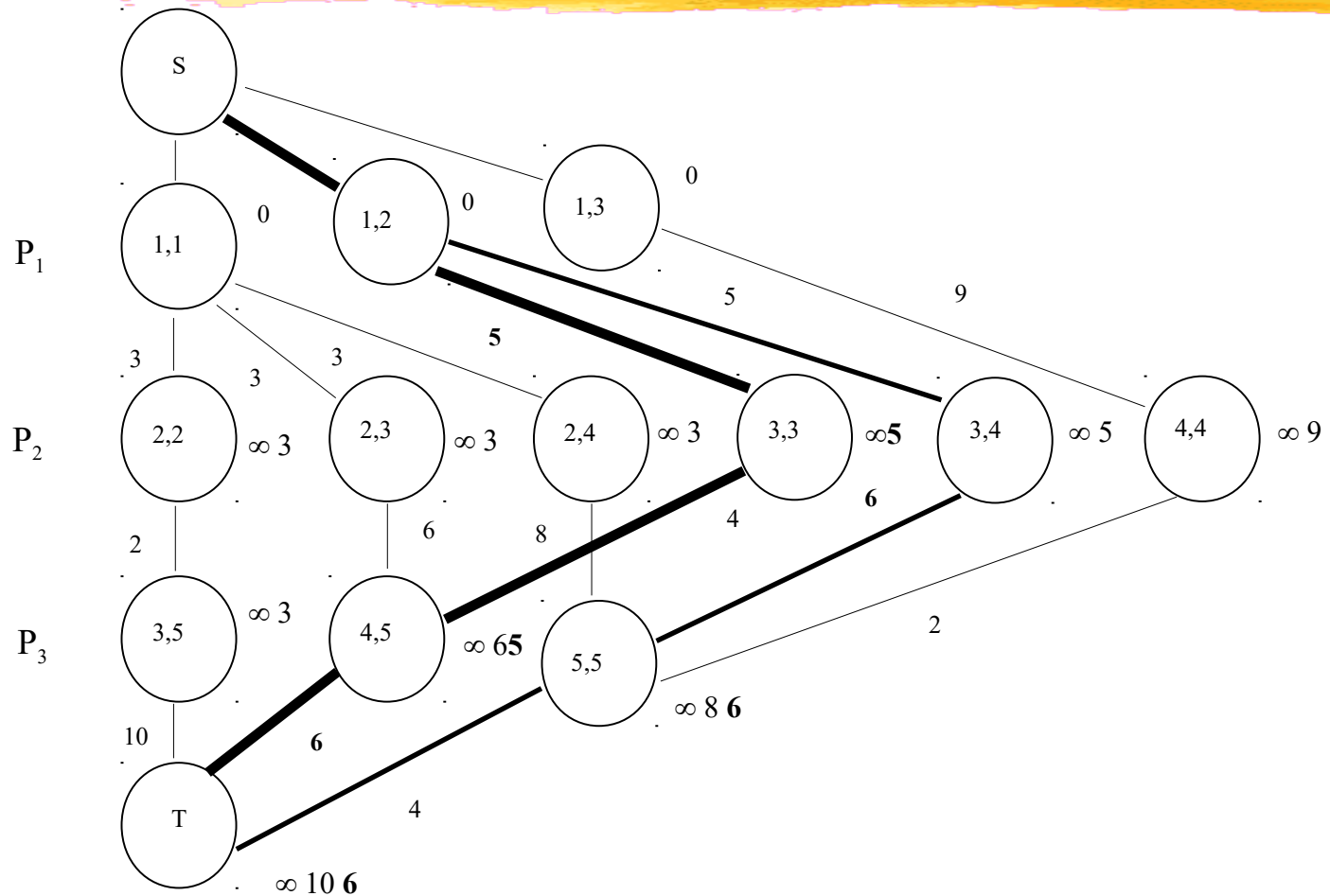
Cominciando dall'alto e a discendere esaminiamo ogni arco e che connette il nodo a (sopra) al nodo b (sotto) e rimpiazziamo l'etichetta $L(b)$ con

$\min [L(b), \max (W(e), L(a))]$

dove $W(e)$ è il peso dell'arco e .

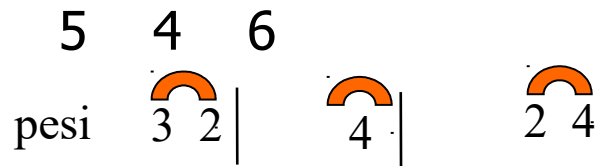
Se occorre esaminare molti archi, mettiamo in evidenza l'arco che contribuisce al valore finale dell'etichetta $L(b)$.

Passo 4



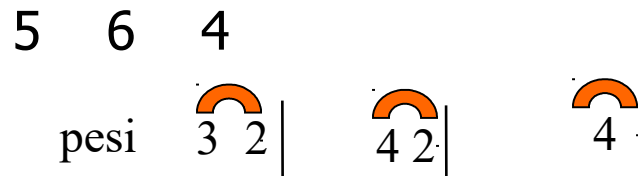
Il cammino critico viene trovato cominciando dal nodo T e seguendo gli archi marcati (se si sono)

cioè



processori	task
P1	1,2
P2	3
P3	4,5

oppure



processori	task
P1	1,2
P2	3,4
P3	5

Complessità algoritmo statico ottimo

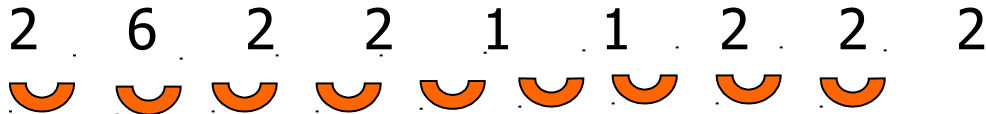
- ❑ Spesso questo metodo di trovare la soluzione ottima di un problema assegnato è troppo oneroso.
- ❑ Se il numero di nodi per strato è $O(m^2)$ e ci sono n strati allora il numero di nodi nel grafo è $O(m^2n)$. Inoltre il numero di archi uscenti da un nodo è al più m quindi la complessità totale dell'algoritmo statico ottimo è $O(m^3n)$.
- ❑ Vediamo quindi, 2 algoritmi non ottimi ma meno onerosi.

Metodo di bisezione

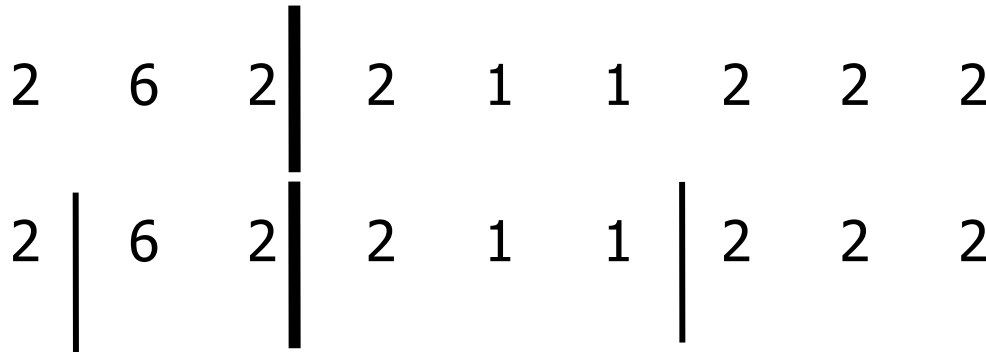
- ❑ La catena di m task viene divisa in 2 gruppi tali che la differenza tra la somma dei tempi di esecuzione di ogni gruppo sia minima.
- ❑ I 2 gruppi vengono quindi ricorsivamente partizionati tante volte quanto si desidera.
- ❑ Il numero di pezzi in cui una catena può essere partizionata è 2^k dove k è la profondità della partizione (cioè è il numero di livelli)
- ❑ Questo algoritmo è usatissimo in problemi in cui il numero di processori è una potenza di 2.

Esempio

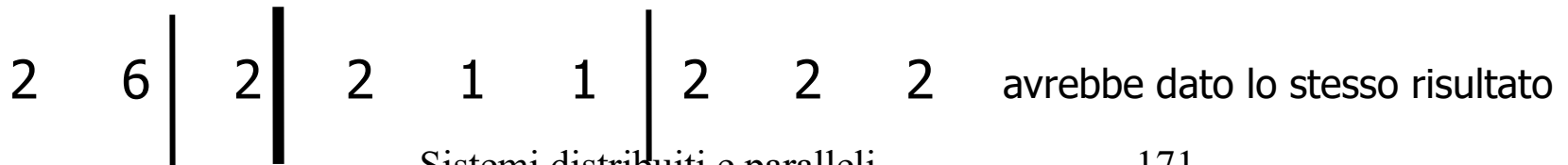
Se noi consideriamo la seguente catena di 9 task:



e fissiamo $N = 4$ (numero di processori)



il caricamento più pesante prevede 8 unità. Anche la partizione



Complessità algoritmo di bisezione

- Il tempo richiesto dall'algoritmo è $O(m \log_2 N)$ poichè non avremo più di $\log_2 N$ livelli di partizione e in ogni livello è richiesto al più un accesso al peso del task.

La soluzione ottima dell'esempio sopra (ricavabile "a mano" o tramite il precedente "algoritmo statico ottimo") è :

2 | 6 | 2 2 1 1 | 2 2 2 che da un

Caricamento massimo di 6 e non 8 unità.

Algoritmo "greedy"

Algoritmo "greedy" (goloso)

Sia W_{sum} il peso totale di tutti i task, W_{max} sia il peso del task più pesante e W_{opt} sia il peso della sottocatena più pesante in una partizione ottima.

Si può dimostrare che W_{opt} è sempre limitato dalla:

$$\frac{W_{\text{sum}}}{N} \leq W_{\text{opt}} \leq \frac{W_{\text{sum}}}{N} + W_{\text{max}} \quad (*)$$

dove N è il numero di processori.

Questa è la base dell'algoritmo.

Scegliamo un peso di prova W compreso nella (*) e tentiamo di partizionare la catena di task in sottocatene in cui il peso di ogni sottocatena sia $\leq W$.

Se troviamo una partizione questa viene chiamata "greedy" (golosa). Se esiste una partizione "greedy" possiamo ridurre il peso e tentare di trovare un'altra volta una partizione "greedy" più accurata relativa a quel peso. Se invece non troviamo la partizione "greedy" dobbiamo incrementare il peso.

Algoritmo "greedy"

La selezione del peso di prova è attuata effettuando una ricerca binaria nella :

$$\frac{V_{sum}}{N} \leq W \leq \frac{V_{sum}}{N} + W_{max} .$$

Algoritmo "greedy"

$$W_{basso} = W_{sum}/N$$

$$W_{alto} = W_{sum}/N + W_{max}$$

$$\text{repeat } W = \frac{V_{basso} + W_{alto}}{2}$$

$$\text{if } \text{partizione}(W) \text{ then } W_{alto} = W$$

$$\text{else } W_{basso} = W$$

$$\text{until } (W_{max} - \frac{V_{sum}}{N} < \epsilon)$$

end

dove:

- ◀ W_{max} è il peso della sottocatena più pesante
- ◀ $\text{partizione}()$ è una funzione che restituisce:
 - True se è possibile trovare una partizione "greedy"
 - False altrimenti
- ◀ ϵ indica l'accuratezza richiesta.

Algoritmo "greedy"

Function partizione(W): boolean;

begin

 i = 1; j = 1; p = 1;

 while p <= N do begin

 repeat

 j = j + 1

 until (peso della sottocatena [i..j] > W) or (j = m + 1)

 if j = m (cioè tutti i task sono stati assegnati) then

 assegna la sottocatena [i..m] al processore p

 return True

 stop

 endif

 assegna la sottocatena [i..j-1] al processore p

 i = j

 p = p + 1

endwhile

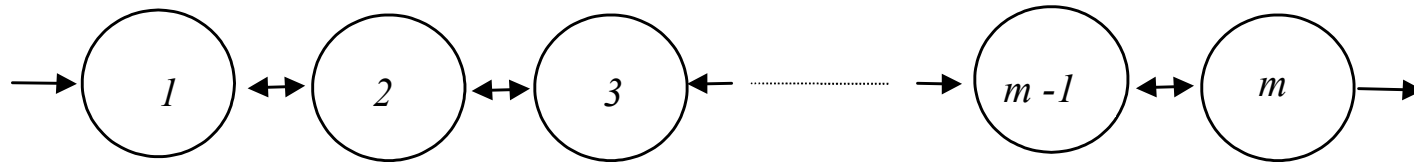
return False

stop

end

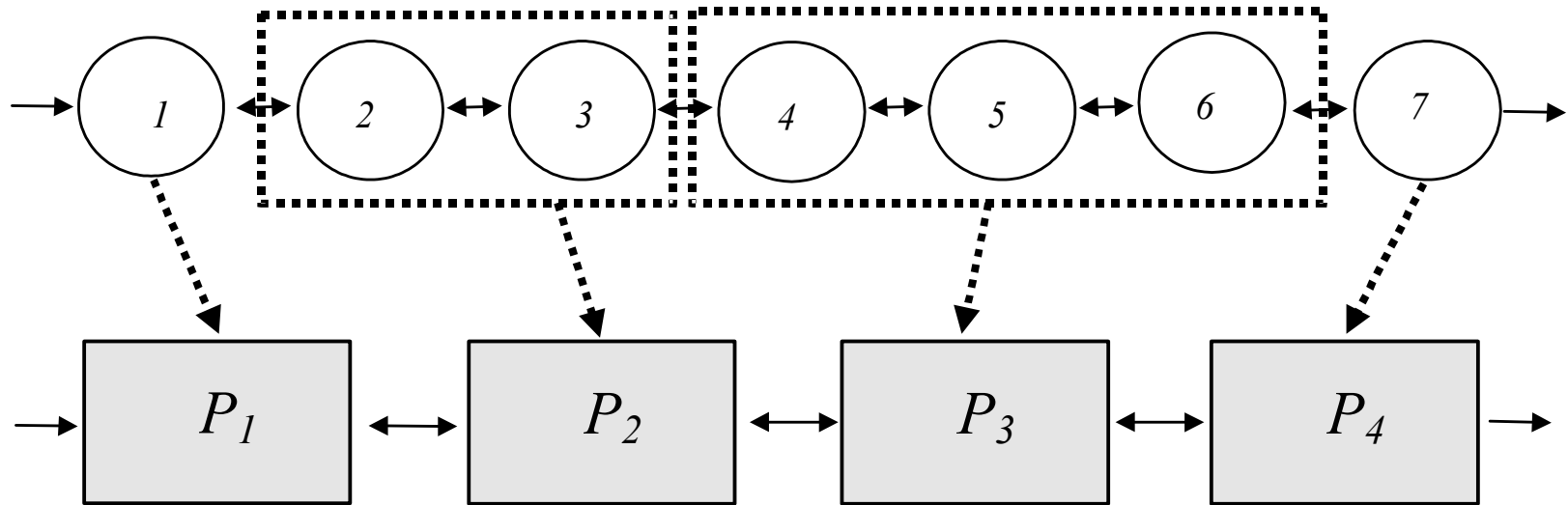
Algoritmi dinamici

- Assumiamo che la rete di processi sia organizzata come un array lineare contenente N processori e assumiamo che il problema del partizionamento (mappatura, assegnamento) sia composto da m task di calcolo numerati da $1, \dots, m$ che possono essere così rappresentati:



- Notare la dipendenza di un task dall'altro.
- Il calcolo di ogni task è diviso in passi e il task i richiede dati dal task $i-1$ e dal task $i+1$ al passo $s-1$ per poter fare i calcoli richiesti al passo s .
- Quindi bisogna assegnare m task agli N processori ($N \leq m$) in modo tale che ogni processore abbia assegnata una sottocatena di moduli contigui e venga minimizzato il carico del processore più caricato.

□ Per esempio:



□ Il tipo di formulazione fornisce il partizionamento di algoritmi di tipo pipeline su reti lineari.

Cambio di dimensioni

- Occorre fare qualche premessa.

Per ogni valore di i ci saranno 2^{d-1} cambi dove i è il contatore d'iterazione e d è la dimensione dell'ipercubo.

L' esempio nelle slide seguenti illustra bene questo algoritmo.

Dopo d iterazioni tutti i processi avranno un caricamento uniforme (somma del potenziale lavoro che devono fare) dato da:

$$W = \frac{\sum_{i=1}^N w_i}{N}$$

cioè la media di tutti i pesi dei processi.

Tutti i processori sono coinvolti nello scambio di dati per ogni valore di i , cioè ogni processore scambia dati con i suoi vicini nella dimensione i .

- Se il processore è rappresentato da *inode* allora i suoi vicini nella *i*-esima dimensione si trovano invertendo (da destra verso sinistra) l'*i*-esimo bit di *inode*:
- se *inode* = 2 (010) e *d* = 3
 - ✧ *i* = 1 inverte il 1° bit \Rightarrow 011 = 3
 - ✧ *i* = 2 inverte il 2° bit \Rightarrow 000 = 0
 - ✧ *i* = 3 inverte il 3° bit \Rightarrow 110 = 6
- Nel linguaggio C questo può essere espresso così:
 - $\text{vicino} = \text{inode} \wedge (1 \ll (i-1))$
- Per rilocare lavoro da un processore ad un altro occorre comunicare solo i dati associati ad un task. (Questo è applicabile ad algoritmi partizionati in cui tutti i processori fanno girare più o meno lo stesso codice, ma il singolo processore è responsabile solo di un sottoinsieme di dati)
- Il costo dello spostamento di un task (relativo alla comunicazione) è molto piccolo rispetto al tempo di esecuzione del task in loco, cioè senza muoverlo.

Algoritmo del cambio di dimensioni

```
for  $i = 1$  to  $d$   
  invia caricomio ai processori vicini nella dimensione  $i$   
  ricevi caricosuo dai processori vicini nella dimensione  $i$   
   $media = (caricomio + caricosuo) / 2$   
   $caricomio = media$   
endfor
```

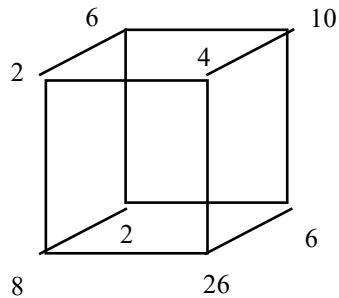
dove:

d	è la dimensione dell'ipercubo
<i>caricomio</i>	è la somma del potenziale lavoro nel processore, cioè è una stima di quanto lavoro dovrà essere fatto
<i>caricosuo</i>	è simile a <i>caricomio</i> ma si riferisce ai processori vicini

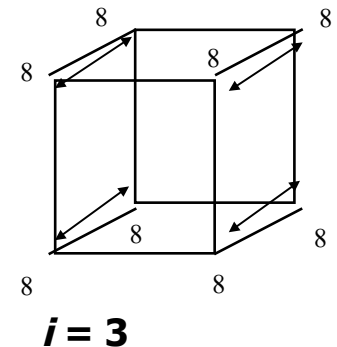
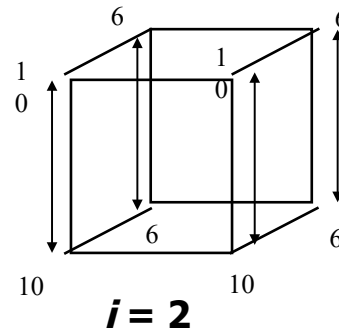
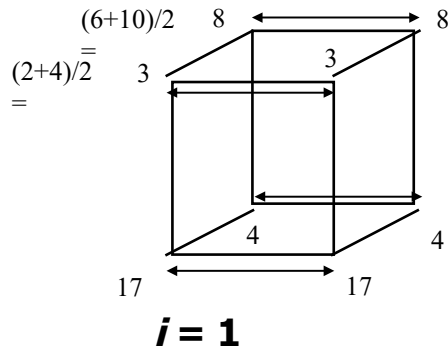
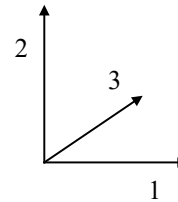
Esempio cambio di dimensioni

Esempio:

Consideriamo la configurazione iniziale, con $d = 3$



nelle 3 dimensioni



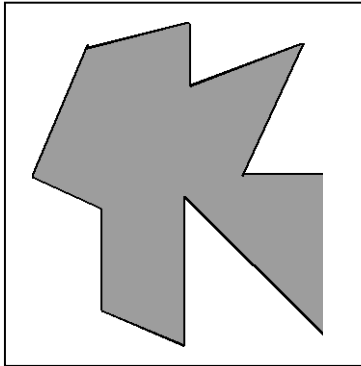
Decomposizione sparsa dei domini



- ❑ In alcune applicazioni in cui il carico nella rete varia molto da un passo ad un altro, usare un bilanciamento di carico dinamico può causare tempi di esecuzione totale lunghi quanto quelli in cui il bilanciatore non viene usato (poiché il bilanciatore viene invocato quasi ad ogni passo).
- ❑ La decomposizione sparsa non è una tecnica di bilanciamento dinamico del carico, ma una tecnica di decomposizione di domini che è perseguibile in problemi in cui il carico tra i processi può variare moltissimo, per esempio in una simulazione dinamica o nella costruzione di modelli per regioni irregolari.

Esempio decomposizione sparsa dei domini

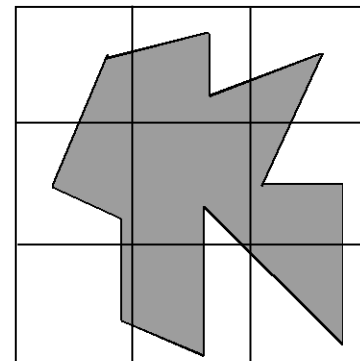
Consideriamo un dominio rettangolare che contiene qualche regione irregolare:



Normalmente suddividiamo il rettangolo in N^2 aree e assegnamo un processore ad ogni area (N^2 processori).

Cioè ogni processore ($0 \div 8$) è responsabile della propria area. In questo modo qualche processore avrà molto più lavoro di altri. Infatti:

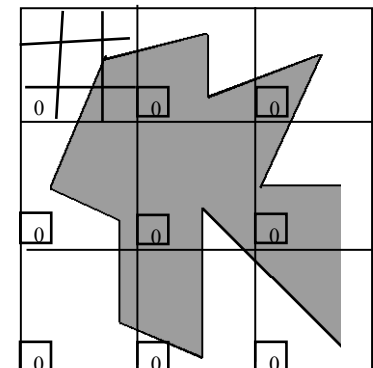
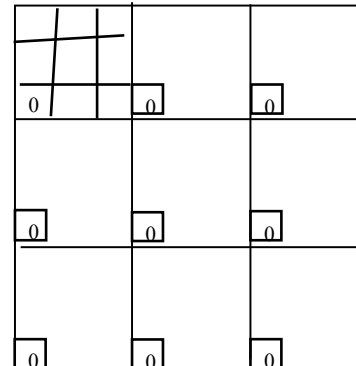
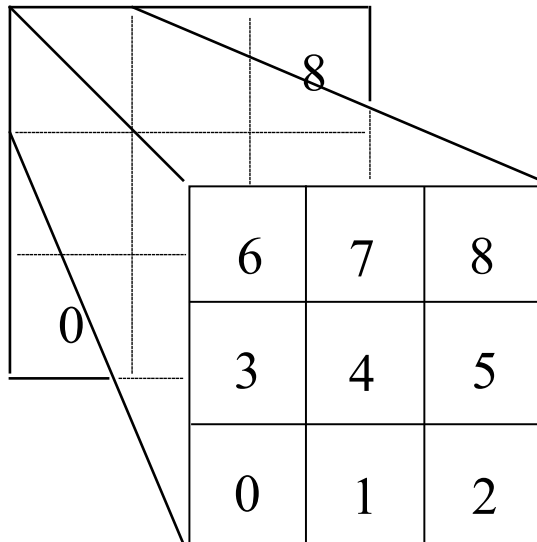
6	7	8
3	4	5
0	1	2



La decomposizione sparsa implica la decomposizione di ognuno degli N^2 rettangolini (**templates**) in un altro insieme di N^2 rettangolini chiamati **granules**, cioè:

Ad ogni processore viene assegnato un **granule** per ogni **template**.

Per esempio il processore 0 sarà responsabile di:



In questo modo il caricamento dei processori dovrebbe essere approssimativamente bilanciato.

Granularità

- Per coordinare il lavoro tra i vari processori coinvolti a risolvere lo stesso problema viene richiesta una qualche forma di comunicazione. Il rapporto tra calcolo e comunicazione è chiamato **granularità**.

Parallelismo a grana fine

- Tutti i task eseguono poche istruzioni tra 2 cicli di comunicazioni
- Il bilanciamento del carico è facilitato
- Il rapporto tra calcolo e comunicazione è basso
- Implica un forte overhead delle comunicazioni e scarse opportunità di migliorare le prestazioni
- Se la grana è troppo fine è possibile che l'overhead richiesto dalle comunicazioni e dalle sincronizzazioni tra i task sia più dispendioso del calcolo.

Granularità

Parallelismo a grana grossa

- ❑ Caratterizzato da lunghi calcoli e da un gran numero di istruzioni eseguite tra 2 cicli di comunicazioni
- ❑ Il rapporto tra calcolo e comunicazione è alto
- ❑ Può portare buone opportunità nel miglioramento delle prestazioni
- ❑ E' difficile bilanciare il carico in modo efficiente
- ❑ Il tipo di granularità da scegliere dipende dall'algoritmo e dall'hardware su cui questo "gira".
- ❑ Nella maggioranza dei casi l'overhead da pagare nelle comunicazioni e nelle sincronizzazioni è così alto rispetto alla velocità di esecuzione che risulta conveniente avere una grana piuttosto grossa.

Dipendenza dei dati

- ❑ Esiste una dipendenza dei dati quando abbiamo un uso condiviso della stessa locazione di memoria.
- ❑ Una dipendenza dei dati spesso impedisce una esecuzione parallela

Esempio 1:

```
DO 500 J = INIZIO, FINE  
    A(J) = A(J-1) * 2.0  
500 CONTINUE
```

Se il task2 ha A(J) e il task1 ha A(J-1), il valore di A(J) dipende da:

- memoria distribuita:
 - ✧ il task2 ottiene il valore di A(J-1) dal task1 (comunicazione)
- ❑ memoria condivisa:
 - ✧ se il task2 legge A(J-1) prima (o dopo) che il task1 lo ha aggiornato (sincronizzazione)

Esempio 2:

task1 task2

$x = 2$ $x = 4$

:

$y = x^{**}2$ $y = x^{**}3$

Il valore di y dipende da:

- memoria distribuita:

 - ✧ se e/o quando il valore di x viene comunicato tra i 2 task

- memoria condivisa:

 - ✧ quale task modifica per ultimo il valore di x .

Tipi di dipendenza dei dati

- ❑ Dipendenza dal flusso:
 - Il task2 usa una variabile calcolata dal task1. Il task1 deve memorizzare o inviare la variabile prima che il task2 la utilizzi
- ❑ Dipendenza dall'output:
 - Sia il task1 che il task2 calcolano la stessa variabile e il task2 deve memorizzare o inviare il suo valore dopo il task1
- ❑ Dipendenza dal controllo:
 - L'esecuzione del task2 dipende da un comando condizionale (if,...) del task1. Il task1 deve completare prima una decisione che può far partire l'esecuzione del task2.

Come gestire la dipendenza dei dati:

- memoria distribuita:
 - ✧ stabilire punti di sincronizzazione per passare i dati richiesti
- memoria condivisa:
 - ✧ sincronizzare tra i vari task le operazioni di lettura/scrittura.

Modelli di comunicazione e banda

- Per qualche problema aumentare il numero dei processori può far:
 - diminuire il tempo di esecuzione per il calcolo
 - ma anche aumentare il tempo di esecuzione per la comunicazione
- Il tempo richiesto dalla comunicazione dipende dai parametri di banda del sistema.

Ad esempio il tempo t necessario per inviare W parole tra 2 processori è dato da:

$$t = L + W / B$$

dove: L = latenza

e B = velocità del flusso di bit sull'hardware espressa in parole/secondo.

- I modelli di comunicazione influiscono anche sul rapporto tra calcolo e comunicazione.
- Per esempio in comunicazioni del tipo "gather-scatter" (raggruppa-suddividi) tra un singolo processore e N altri processori, peserà più un incremento della latenza che se gli N processori comunicassero solo con i loro vicini più prossimi.

I/O parallelo



Le operazioni di I/O vengono generalmente viste come inibitori del parallelismo.

I sistemi di I/O parallelo sono non sempre disponibili.

In ambienti in cui tutti i processori vedono lo stesso insieme di file, le operazioni di scrittura consteranno in sovrascritture dei file.

Le operazioni di lettura sono gestite più o meno bene a seconda dell'abilità del fileserver a soddisfare richieste contemporanee di lettura delle stesse informazioni.

L'I/O che deve essere effettuato su una rete (non locale) può causare gravi "colli di bottiglia".

I/O parallelo

Qualche consiglio:

- Ridurre il più possibile le operazioni di I/O.
- Confinare l'I/O di un job a specifiche porzioni seriali.
 - ✧ Per esempio il task1 potrebbe leggere un file di input e poi comunicare i dati richiesti ad altri task oppure il task1 potrebbe effettuare una operazione di scrittura dopo aver ricevuto i dati richiesti da tutti gli altri task.
- Creare nomi di file unici per ogni file di I/O di un task.
- Per sistemi a memoria distribuita con insiemi di file condivisi effettuare l'I/O in locale, non sui file condivisi.
 - ✧ Per esempio ogni processore può usare una directory /tmp, questo è generalmente più efficiente che effettuare l'I/O sulla rete in una certa home directory.

Configurazione della macchina



- ❑ Per ottenere prestazioni parallele ottime occorre adattare i propri algoritmi all'architettura e alla configurazione del sistema su cui si lavora.
- ❑ A tal scopo occorre conoscere alcuni parametri di ambiente quali:
 - numero ottimo dei nodi (processori) da usare
 - dimensioni della memoria
 - organizzazione della cache
 - carico del sistema.

Interruzioni e controlli



- Nella programmazione parallela è generalmente a carico del programmatore la gestione di eventi quali:
 - interruzioni macchina
 - interruzioni di task
 - checkpointing (punti di controllo)
 - ripartenza

Lo stallo

- Lo "stallo" descrive la condizione in cui 2 (o più) processi attendono un evento o un messaggio l'uno (gli uni) dall'altro (dagli altri).
- L'esempio più banale di stallo è quello di 2 processi che sono stati programmati per leggere/ricevere l'uno dall'altro prima di scrivere/inviare.

Esempio:

task1

:

$x = 4$

mittente = task2

receive(mittente, y)

destinatario = task2

send(destinatario, x)

$z = x + y$

:

task2

:

$y = 8$

mittente = task1

receive(mittente, x)

destinatario = task1

send(destinatario, y)

$z = x + y$

- E' quindi a carico del programmatore evitare situazioni di stallo controllando bene i flussi tra i vari task sia in fase progettuale che di sviluppo.

Debugging



- Effettuare il debugging di programmi paralleli è molto più complesso che per programmi seriali.
- Un po' di software per il debugging parallelo comincia ad essere disponibile, ma molto resta ancora da fare.
- Nello sviluppo dei programmi si consiglia di utilizzare una organizzazione a moduli.
- Occorre inoltre prestare attenzione ai dettagli della comunicazione quanto ai dettagli del calcolo.

Monitoraggio e analisi delle prestazioni



- Come per il debugging, il monitorare e analizzare l'esecuzione di programmi paralleli è molto più complesso che per programmi seriali.
- Esistono "pacchetti" per il monitoraggio e l'analisi dell'esecuzione di un programma.
- Un settore delicato resta quello della scalabilità.