Elementi di Programmazione Parallela

Claudia Truini
c.truini@cineca.it

Luca Ferraro
l.ferraro@cineca.it

CINECA SCAI
SuperComputing Applications and Innovation

CINECA COURSES
High Performance Computing

Sommario

- Elementi di Parallelismo
- Misura delle prestazioni parallele
- Tecniche di partizionamento
- Load Balancing
- Comunicazioni

Problema 1: Serie di Fibonacci

Calcolare e stampare i primi N elementi della serie di Fibonacci

- La serie di Fibonacci {1, 1, 2, 3, 5, 8, …} è così definita:

$$f_1 = 1; \quad f_2 = 1$$

$$f_i = f_{i-1} + f_{i-2} \qquad \forall n > 2$$

Problema 2: Serie geometrica

Calcolare la somma parziale N-sima della serie geometrica

- La serie geometrica è così definita:

$$g_1 = x, \quad g_2 = x^2, \quad g_3 = x^3, \ldots.$$

ovvero $g_i = x^i \qquad \forall i > 0$

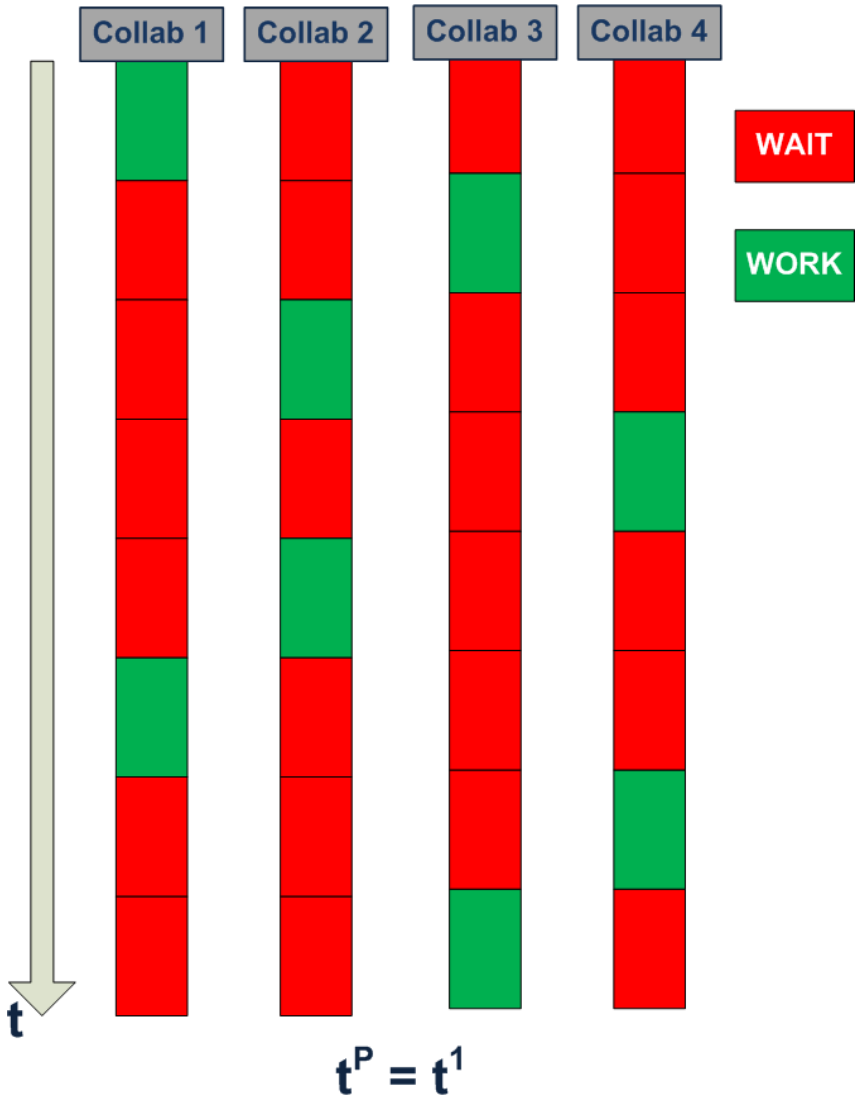
- Dobbiamo calcolare:

$$G_N = \sum_{i=1}^{N} x^i$$

Risoluzione del problema 1 con P collaboratori

Come calcolare nel minor tempo i primi N numeri di Fibonacci?

- Il generico f_i dipende dai 2 numeri precedenti della serie, dunque può essere calcolato solo dopo che siano stati determinati f_{i-1} e f_{i-2}
- Utilizzando P collaboratori:
  1. Un qualsiasi collaboratore calcola e stampa il 3° termine
  2. Un qualsiasi collaboratore calcola e stampa il 4° termine
  3. …
- Utilizzando P collaboratori, il tempo necessario all'operazione è uguale al tempo necessario ad un solo collaboratore!



Risoluzione del problema 2 con P collaboratori

$$G_N = \sum_{i=1}^{P} \left(\sum_{j=1}^{N/P} x^{\frac{N}{P}(i-1)+j} \right) = \sum_{i=1}^{P} S_i$$

dove

$$S_i = \sum_{j=1}^{N/P} x^{\frac{N}{P}(i-1)+j}$$

- Utilizzando P collaboratori:
  1. Ogni collaboratore calcola una delle P somme parziali S_j
  2. Solo uno dei collaboratori somma i P contributi appena calcolati
  3. Il tempo impiegato è uguale a 1/P del tempo che avrebbe impiegato un solo collaboratore

Benefici nell'uso di P collaboratori

Se il problema e l'algoritmo possono essere decomposti in task indipendenti:

- Il lavoro complessivo potrà essere completato in un tempo minore

  In condizioni ideali, il tempo di calcolo diventa $\mathbf{t^P=t^1/P}$, dove $\mathbf{t^m}$ è il tempo di calcolo con **m** collaboratori

- Ogni collaboratore dovrà "ricordare" meno dati

  In condizioni ideali, la quantità di dati da ricordare diventa $\mathbf{A^P=A^1/P}$, in cui $\mathbf{A^m}$ è la size dei dati nel caso con **m** collaboratori

Comunicazione dei dati tra collaboratori

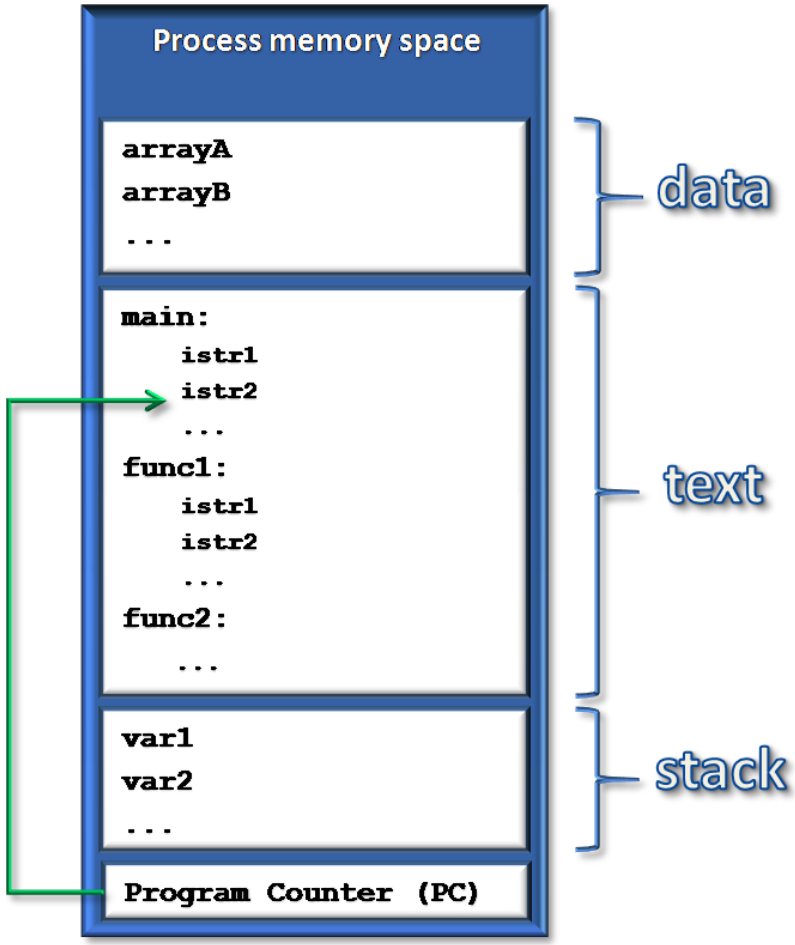
- Per una “fruttuosa cooperazione” tra P collaboratori è necessario che gli stessi possano scambiarsi dati

- *Problema 2*: Serie geometrica
  quando i P collaboratori terminano il calcolo della somma parziale di propria competenza, uno di essi
  1. richiede a tutti gli altri la somma parziale di competenza
  2. somma al proprio i P-1 risultati parziali ricevuti

- Il trasferimento di dati tra collaboratori può avvenire con:
  - la condivisione di dati (paradigma Shared Memory)
  - la *spedizione/ricezione di un messaggio (paradigma Message Passing)*:

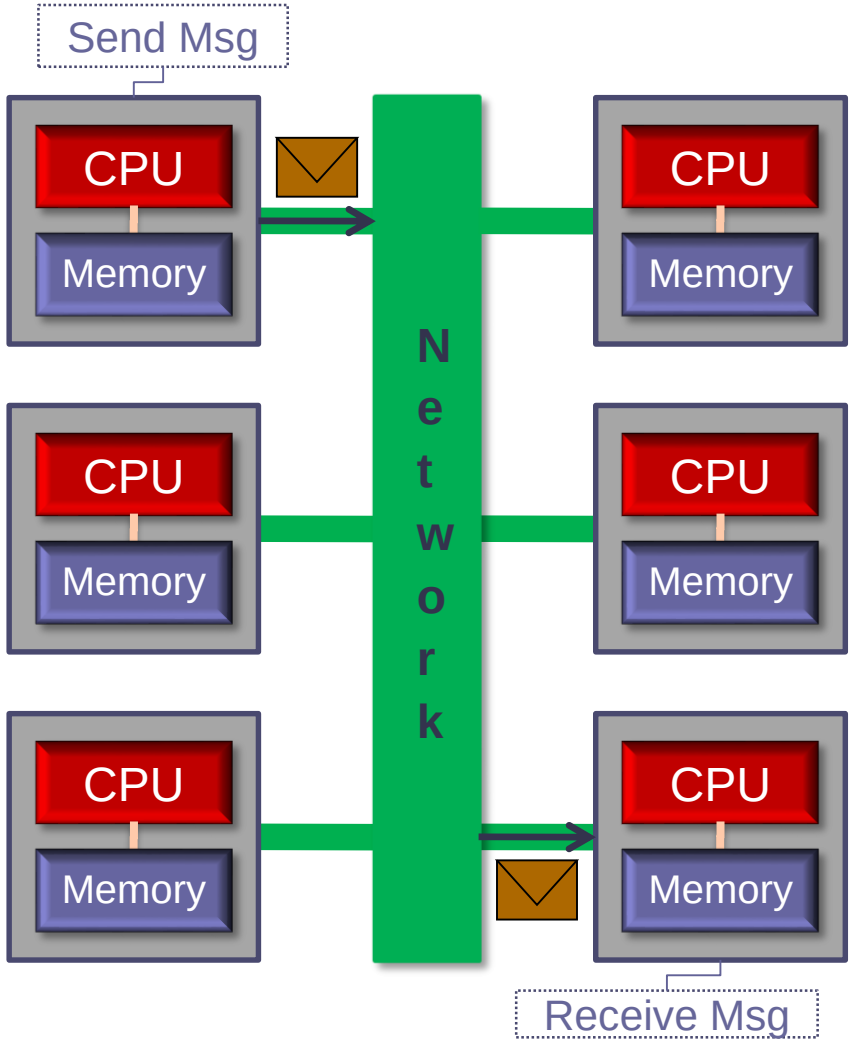
Il collaboratore nel *computing*: il processo

- Un processo è un'istanza in esecuzione di un programma
- Il processo mantiene in memoria i dati e le istruzioni del programma, oltre ad altre informazioni necessarie al controllo del flusso di esecuzione
  - Text
    istruzioni del programma
  - Data
    variabili globali
  - Stack
    variabili locali alle funzioni
  - Program Counter (PC)
    puntatore all'istruzione corrente



Parallelismo Message Passing: il processo

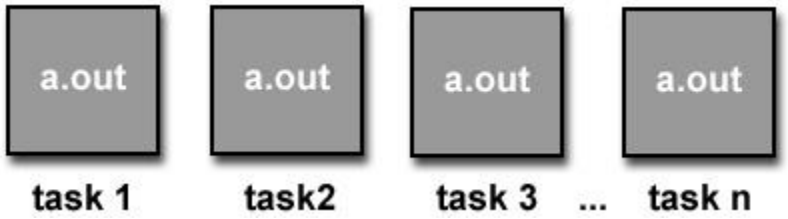
- Ogni processo svolge autonomamente la parte indipendente del task
- Ogni processo accede in lettura e scrittura ai soli dati disponibili nella sua memoria
- E' necessario scambiare messaggi tra i processi coinvolti quando
  - un processo deve accedere ad un dato presente nella memoria di un altro processo
  - più processi devono sincronizzarsi per l'esecuzione di un flusso d'istruzioni



Modello di esecuzione: SPMD

SPMD = Single Program Multiple Data

Ogni processo esegue lo stesso programma, ma opera in generale su dati diversi (nella propria memoria locale)

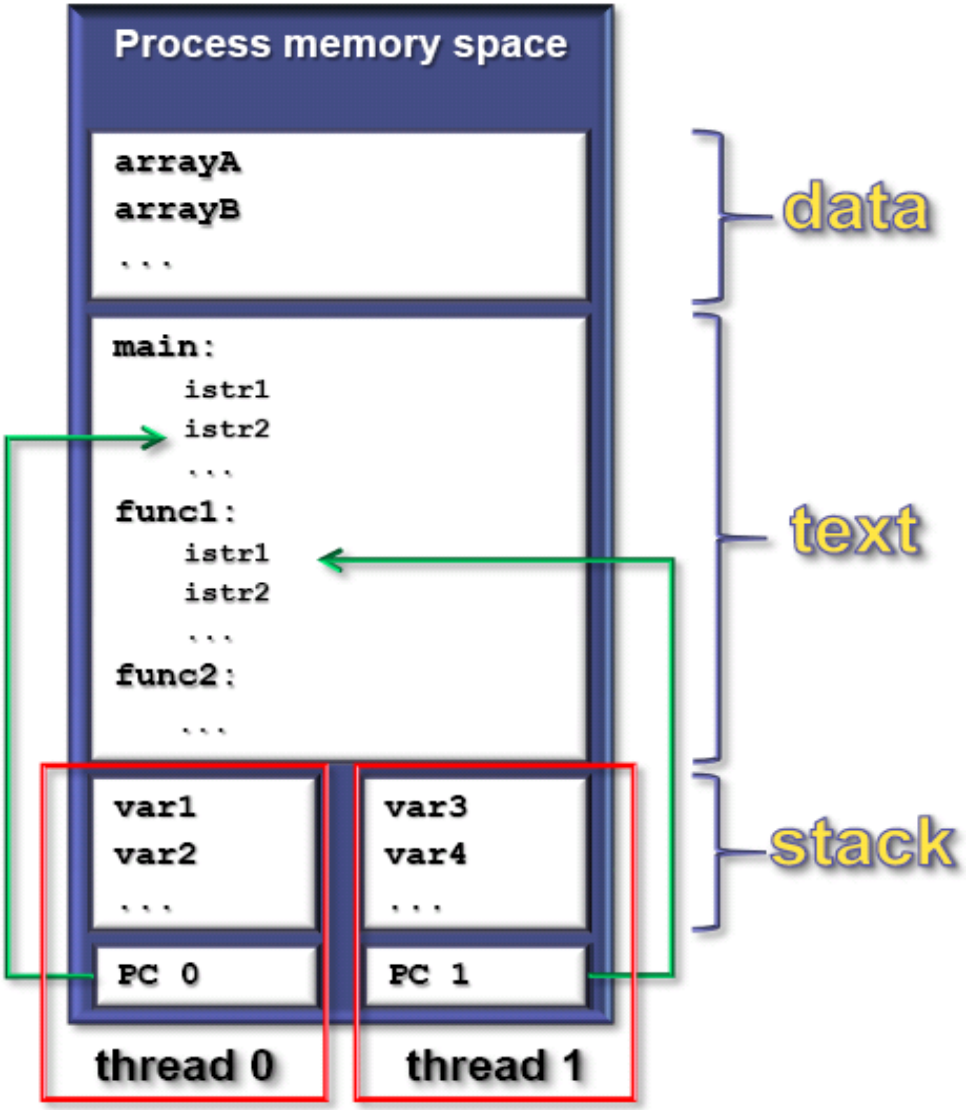


Processi differenti possono eseguire parti di codice differenti:

```
if (I am process 1)
  … do something …
if (I am process 2)
  … do something else …
```

Parallelismo Shared Memory: il thread

- Un processo può avere diversi flussi di esecuzione concorrenti (threads)
- Ogni thread ha:
  - il suo program counter (PC)
  - uno stack privato
- Le istruzioni eseguite da un thread possono accedere:
  - variabili globali (data)
  - variabili locali (stack)



Misurare le prestazioni

- **Speedup**: aumento della velocità (= diminuzione del tempo) dovuto all'uso di n processi:
  - **S(n) = T(1)/T(n)**
    - T(1) = tempo totale seriale
    - T(n) = tempo totale usando n processi
    - S(n) = n → speed-up lineare (caso ideale)
    - S(n) < n → speed-up di una caso reale
    - S(n) > n → speed-up superlineare (effetti di cache)

Misurare le prestazioni

- **Efficienza**: effettivo sfruttamento della macchina parallela:
  - **E(n)** = S(n)/n = T(1)/(T(n)*n)
  - E(n) = 1 → caso ideale
  - E(n) < 1 → caso reale
  - E(n) << 1 → ci sono problemi …
- **Scalabilità**: capacità di essere efficiente su una macchina parallela: aumento le dimensione del problema proporzionalmente al numero di processi

Legge di Amdhal

- Quale è lo speed-up massimo raggiungibile?
- Tempo seriale: $T = F + P$
  - F = tempo della parte seriale (non parallelizzata o non parallelizzabile)
  - P = tempo della parte parallela
- Tempo con n processi:

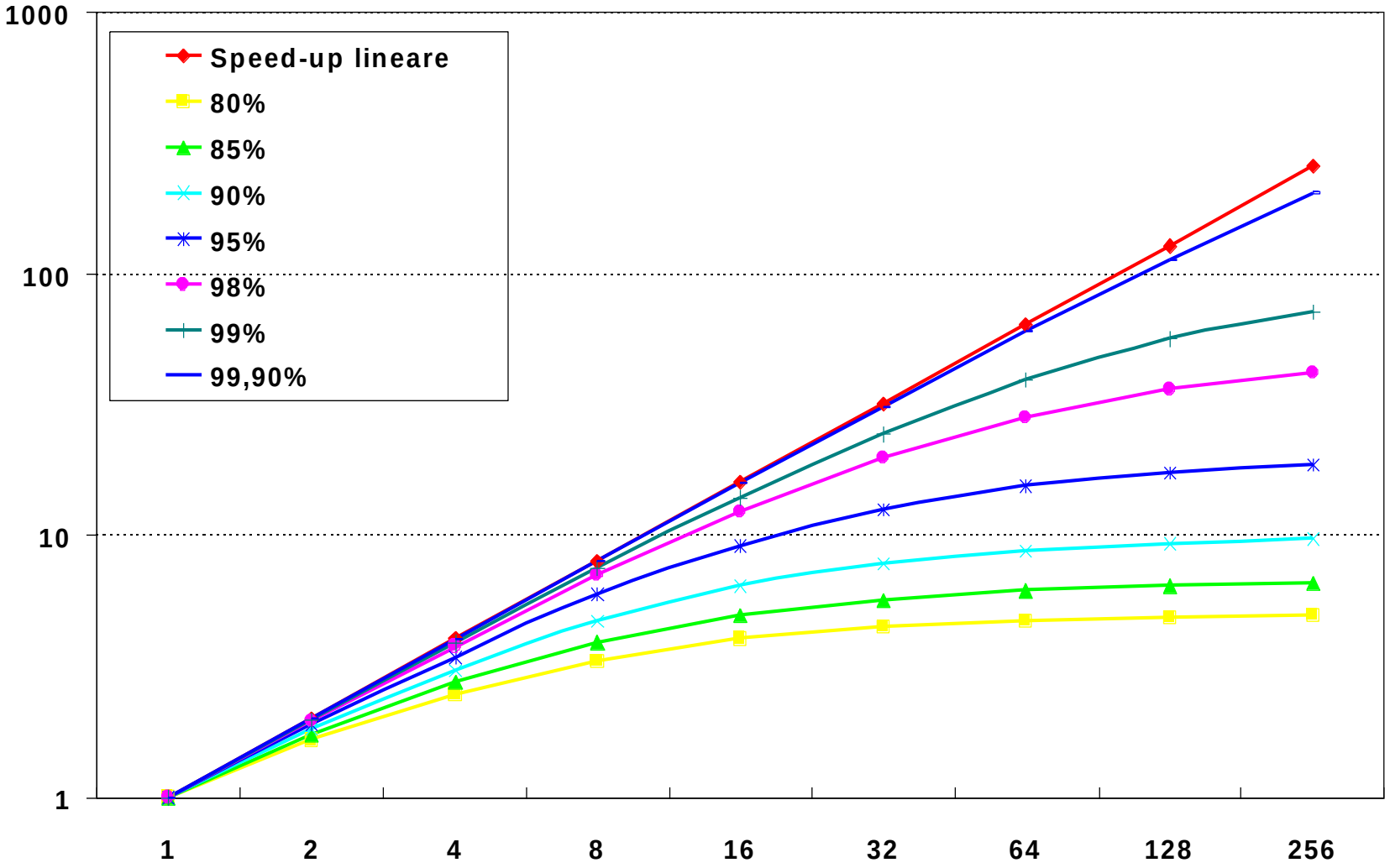
$$T(n) = F + P/n$$

- Speed-up su n processi:

$$S(n) = T(1)/T(n) = (F + P)/(F+P/n)$$

  - Per n grande $S(n) = (F + P)/F$
  - esiste un limite asintotico allo speed-up!

Legge di Amdhal: Speedup

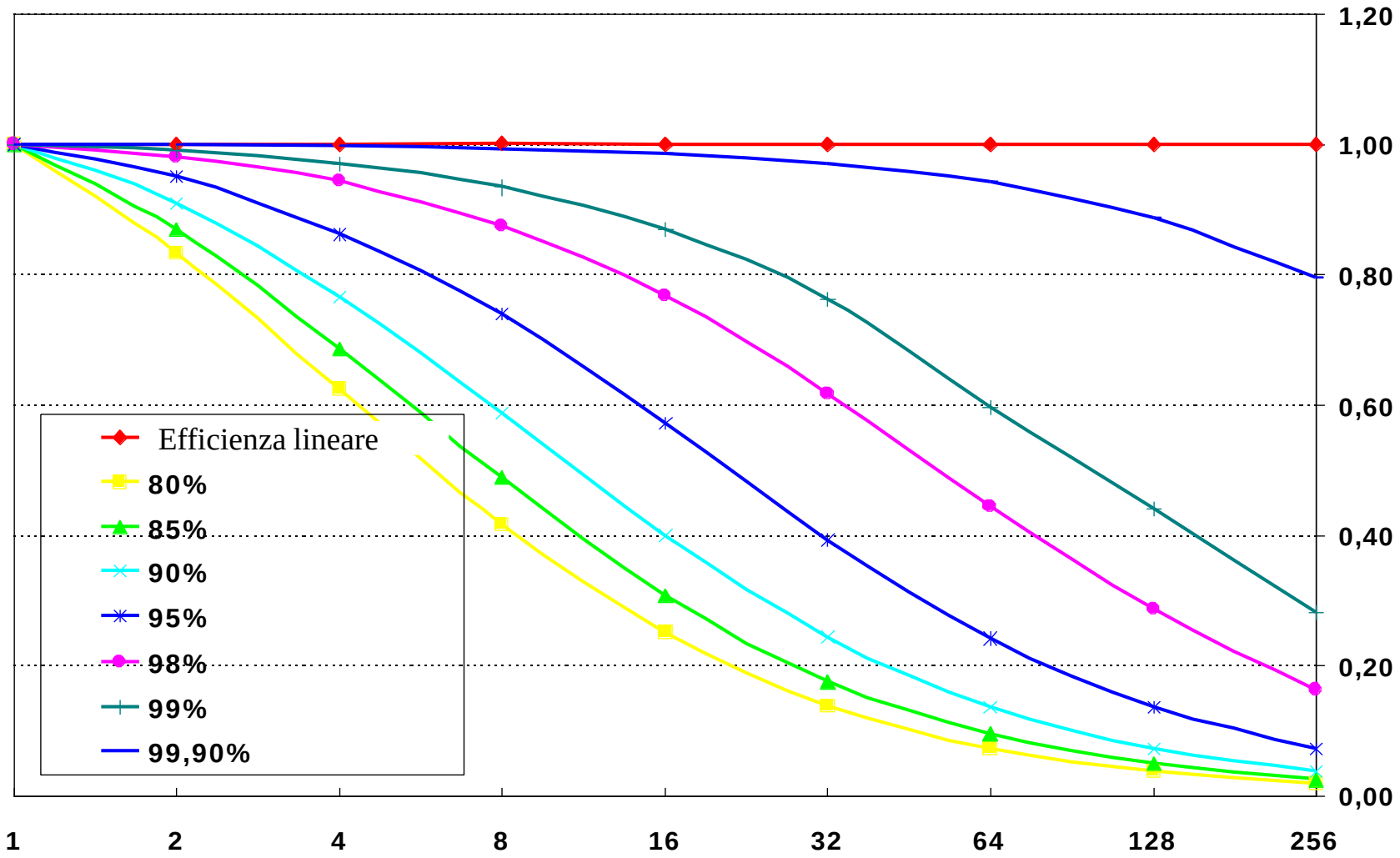


Legge di Amdhal: Speedup

- Massimo speed-up raggiungibile in funzione della percentuale della parte parallele

	1	2	4	8	16	32	64	128	256
80%	1.0	1.7	2.5	3.3	4.0	4.4	4.7	4.8	4.9
85%	1.0	1.7	2.8	3.9	4.9	5.7	6.1	6.4	6.5
90%	1.0	1.8	3.1	4.7	6.4	7.8	8.8	9.3	9.7
95%	1.0	1.9	3.5	5.9	9.1	12.6	15.4	17.4	18.6
98%	1.0	2.0	3.8	7.0	12.3	19.8	28.3	36.2	42.0
99%	1.0	2.0	3.9	7.5	13.9	24.4	39.3	56.4	72.1
99.9%	1.0	2.0	4.0	7.9	15.8	31.0	60.2	113.6	204.0
Ideale	1.0	2.0	4.0	8.0	16.0	32.0	64.0	128.0	256.0

Legge di Amdhal: Efficienza



Legge di Gustafsson

- Quale è la dimensione massima del problema?
- Tempo di esecuzione con n processi:

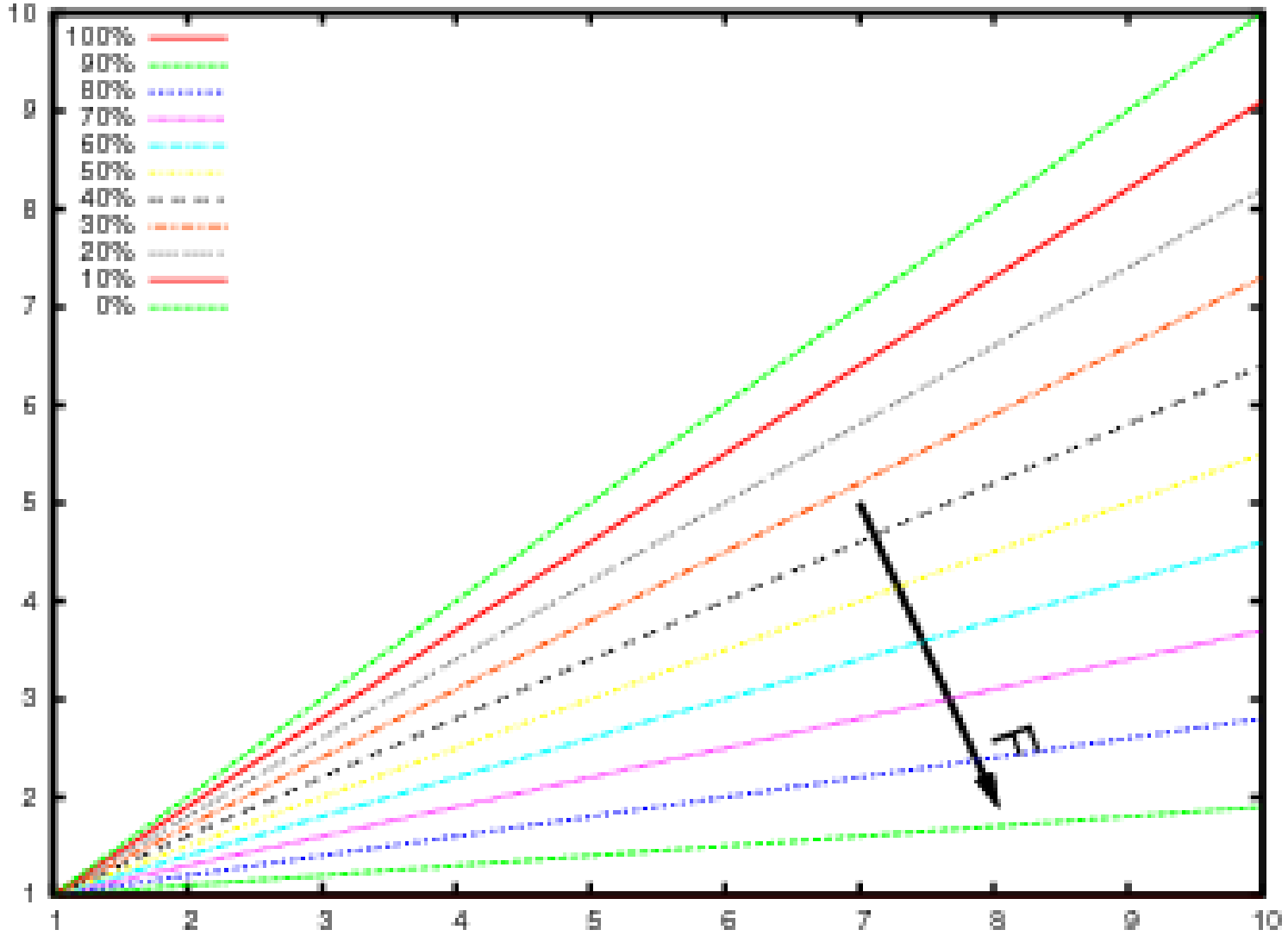
$$T(n) = F + P$$

  - F = tempo della parte seriale (non parallelizzata o non parallelizzabile)
  - P = tempo della parte parallela con n processi
- Speed-up con n processi:

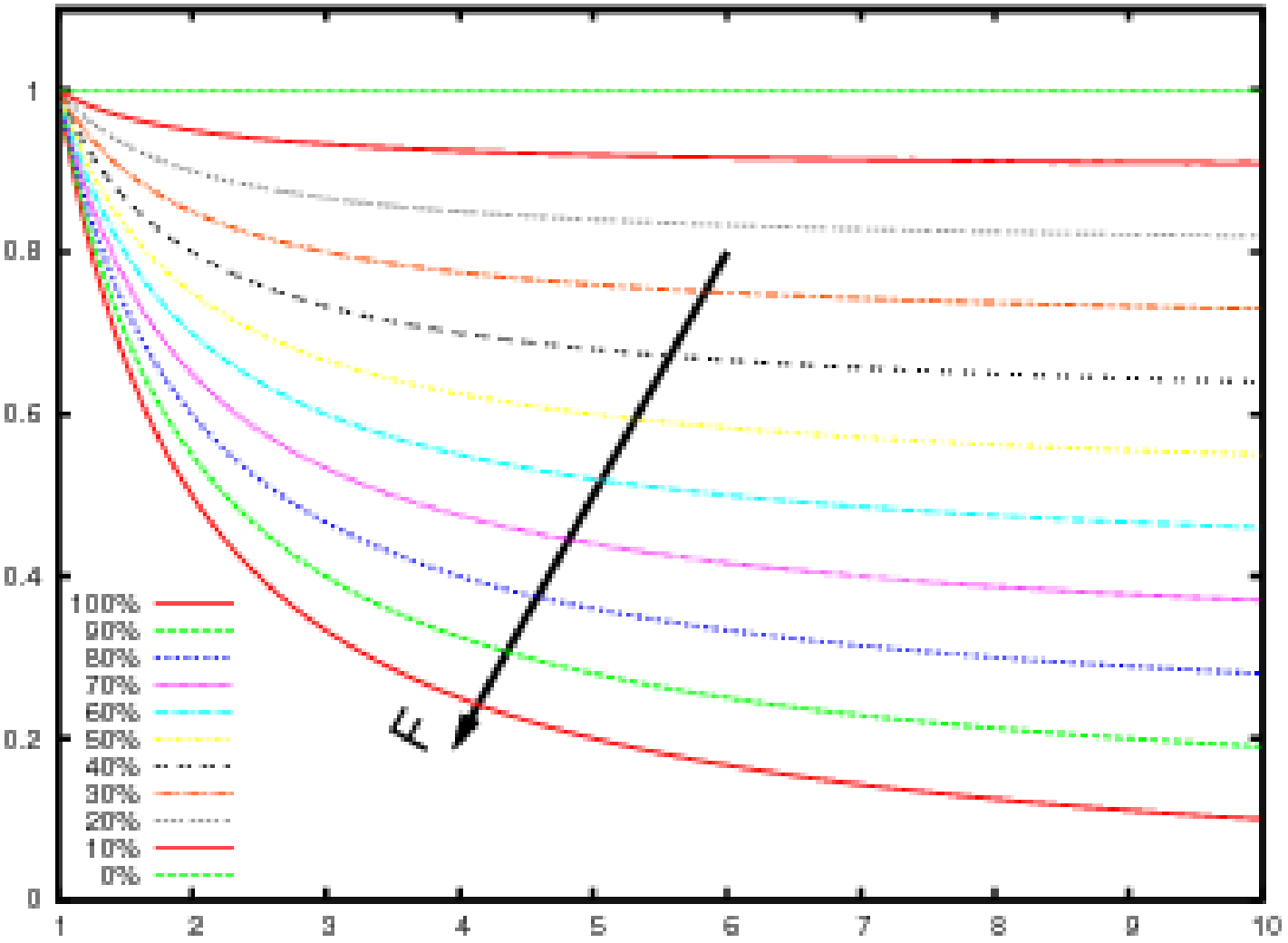
$$S(n) = T(1)/T(n) = (F + nP)/(F+P)$$

  - Ponendo P=1, $S(n) = (F + nP) = F + n(1 -F)$
  - per n grande NON esiste un limite asintotico

Legge di Gustafsson: Speedup



Legge di Gustafsson: Efficienza



Strong *vs* Weak Scaling

- Fixed-size e scaled-size sootolienano i due aspetti fondamentali della scalabilità in contesto HPC
  - aumento del numero di processori
  - aumento della taglia del problema
- Strong Scaling: scalabilità a taglia fissa
  - all'aumentare del numero di processi, voglio risolvere il problema più velocemente
- Weak Scaling: scalabilità a taglia fissa relativa
  - all'aumentare del numero di elaboratori paralleli, quanto efficientemente risolvo problemi più grandi

Modelli di scaling avanzati

- Limiti dei modelli di scaling semplificati
  - si assume che il tempo della parte parallela scali linearmente con il numero di processori
  - si trascurano i tempo di comunicazione, gestione, sincronizzazione e coordinamento addizionali
- La realtà è più complessa
  - i migliori algoritmi seriali sono di solito poco parallelizzabili
  - algoritmi che scalano bene sono solitamente inefficienti
  - la tipologia di parallelizzazione può dipendere dalla taglia del problema e dal numero di collaboratori (processi/thread)
- Per una valutazione 'onesta'
  - miglior algoritmo seriale *vs* miglior algoritmo parallelo

Tecniche di Partizionamento

- Si divide il problema in n parti, ciascuna parte viene assegnata ad un processo differente
- Distribuzione del lavoro tra più soggetti:
  - tutti eseguono le stesse operazioni, ma su un sottoinsieme di dati
  - approccio SIMD (Single Program Multiple Data)
    - → partizionamento dei dati
- Distribuzione delle funzioni tra più soggetti:
  - non tutti eseguono le stesse operazioni
  - approccio Task Parallel
    - → decomposizione funzionale

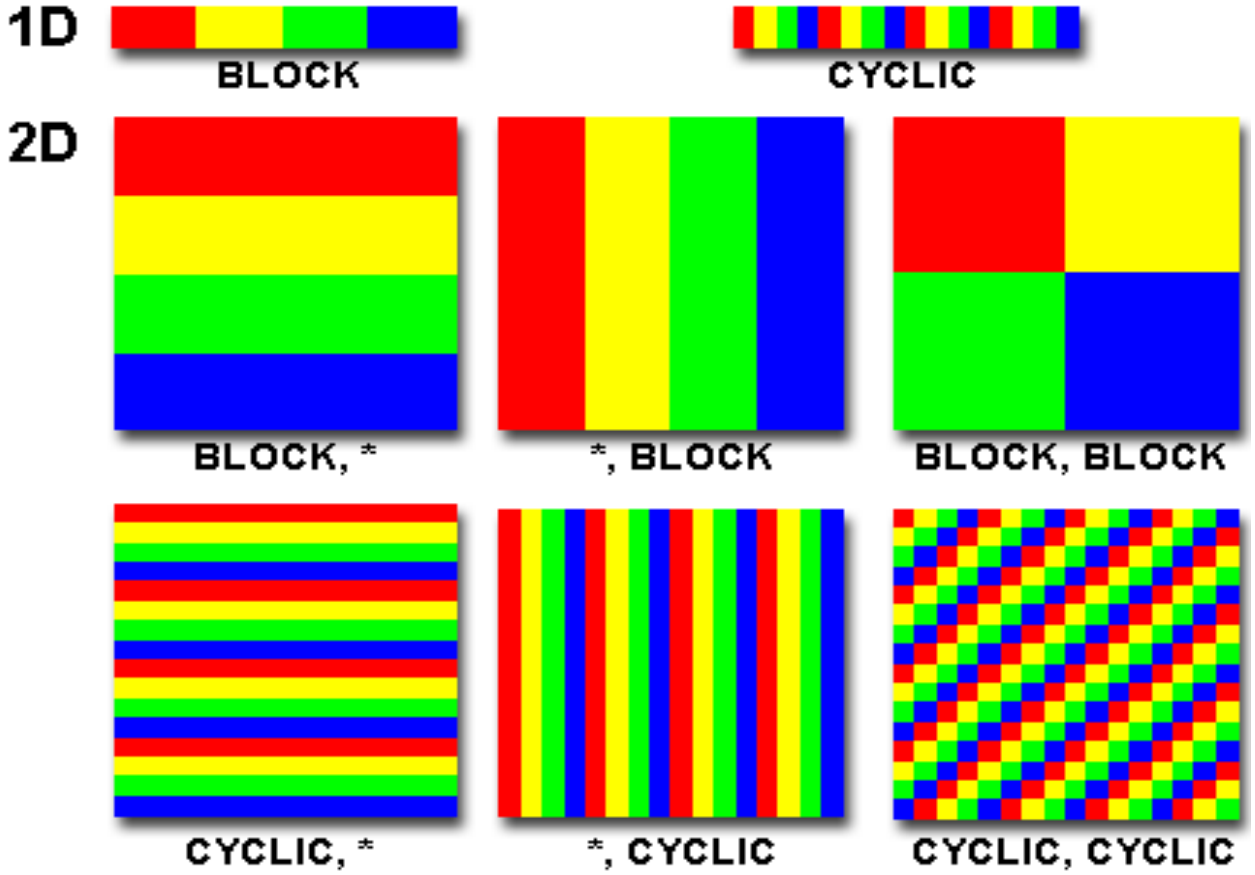
Partizionamento dei dati

- Insieme (ordinato) di dati da elaborare tutti allo stesso modo
- Decompongo i dati associati al problema
- Ogni processo esegue le stesse operazioni su una porzione dei dati

- PRO: scalabile con la quantità di dati
- CONTRO: vantaggiosa solo per casi sufficientemente grandi

Partizionamento dei dati: esempio

- Vari modi possibili per partizionare i dati
- Ogni processo lavora su una porzione di dati

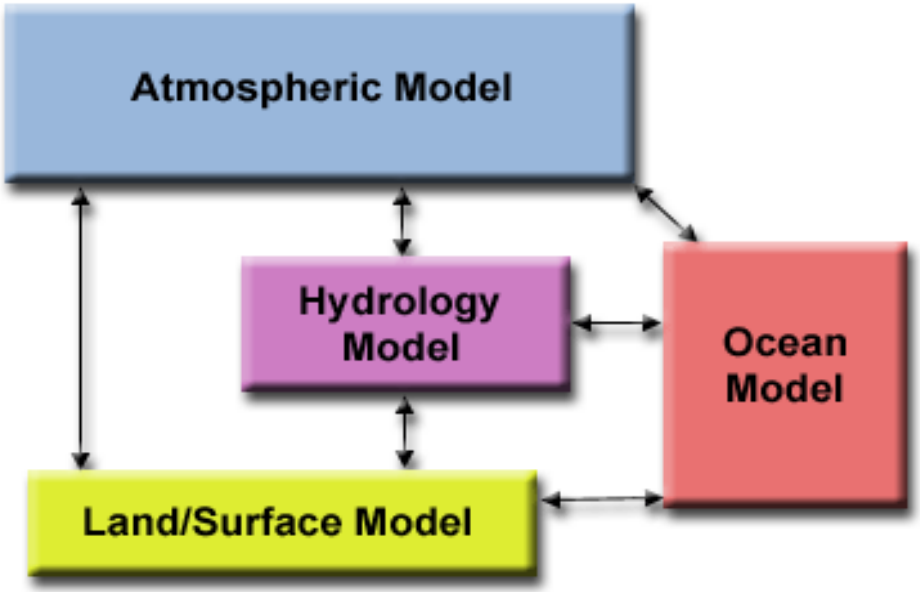


Decomposizione funzionale

- Insieme di elaborazioni differenti ed indipendenti
- Decompongo il problema in base al lavoro che deve essere svolto
- Ogni processo prende in carico una particolare elaborazione
- PRO: scalabile con il numero di elaborazioni indipendenti
- CONTRO: vantaggiosa solo per elaborazioni sufficientemente complesse

Decomposizione funzionale: esempio

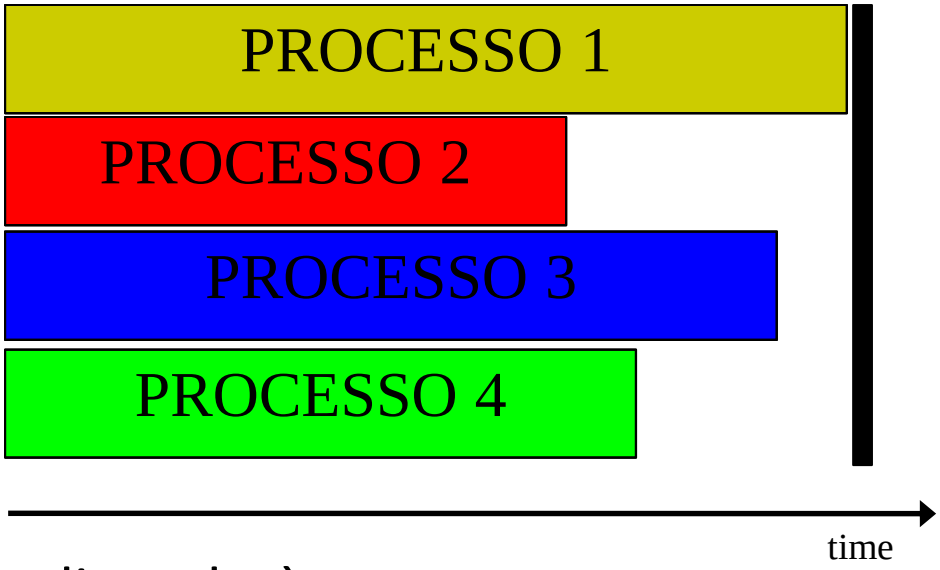
- Modello climatico



Ogni componente del modello può essere considerata come un processo separato. Le frecce rappresentano gli scambi di dati tra le componenti durante il calcolo

Load balancing

- È importante per le performance dei programmi paralleli
- Distribuire il carico di lavoro in modo da minimizzare i tempi morti (tempi di attesa) dei processi
- Esempio con 4 processi
  - tempo seriale = 40
  - tempo parallelo (teorico) = 10
  - tempo parallelo (reale) = 12
    → 20% più lento
  - La velocità dell'esecuzione dipenderà dal processo più lento



Load balancing: assegnazione statica

- Partizionare ugualmente il lavoro di ogni processo
- Per operazioni su array/matrici dove ogni processo svolge operazioni simili, distribuisco uniformemente i dati sui processi
- Per cicli dove il lavoro fatto in ogni iterazione è simile, distribuisco le iterazioni uniformemente sui processi

Load balancing: assegnazione dinamica

- Alcuni partizionamenti anche se uniformi risultano sempre non bilanciati
  - Matrici sparse
  - Metodi con griglie adattative
  - …
- In questi casi solo l'**assegnazione dinamica** può riuscire a bilanciare il carico computazionale
  - divido la regione in tante parti (maggiori del numero di processi)
  - ogni processo prenderà in carico una parte
  - quando un processo termina una parte, ne prende in carico una nuova

Load balancing: confronti

- Assegnazione statica:
  - si usa nella domain decomposition
  - ☺ in genere semplice, proporzionale al volume
  - ☹ soffre di possibili sbilanciamenti

- Assegnazione dinamica:
  - ☺ può curare problemi di sbilanciamento
  - ☹ introduce un overhead dovuto alla gestione del bilanciamento

Comunicazione dei dati

- Sono necessarie?
  - dipende dal problema
    - no, se il problema è “imbarazzantemente parallelo”
    - si, se il problema richiede dati di altri processi
- Quanto comunicare?
  - dipende dal problema e da come viene partizionato
- Tutte le comunicazioni implicano un overhead

...e la rete?

- Fare attenzione anche alla rete di comunicazione
- Quando si pianifica il pattern di comunicazione tra task, e quando si disegnano in dettaglio le comunicazioni, ci sono diversi fattori da tenere in considerazione:
  - **Latenza** (**L**): tempo necessario per "spedire" un messaggio vuoto (tempo di start-up)
  - **Bandwidth** (**B**): velocità di trasferimento dei dati (Mb/sec.)
  - Relazione tra latenza e bandwidth
    - grossa bandwidth e bassa latenza:
      - il migliore dei mondi possibili ☺
    - piccola bandwidth e grande latenza:
      - il peggiore dei mondi possibili ☹
  - **Tempo di comunicazione** (**T**) di un messaggio di N MB:

    T = L + N/B

Legge di Amdhal

- Quale è lo speed-up massimo raggiungibile?
- Tempo seriale: T = F + P
  - F = parte seriale (non parallelizzata/parallelizzabile)
  - P = parte parallela
  - C(n) = comunicazione, sincronizzazione etc….
  - T(n) = F + P/n + C(n)
  - S(n) = T(1)/T(n) = (F + P)/(F+P/n+C(n))

  - per n grande S(n) = (F + P)/(F+C(n))
  - Il limite asintotico allo speed-up è peggiore!
  - ☺ Però potrebbe essere possibile nascondere comunicazione nel calcolo ….

Granularità

- Parallelismo a grana fine
  - Pochi calcoli tra le comunicazioni

  → rapporto tra il calcolo e le comunicazioni è piccolo
  - Può essere facile bilanciare il carico
  - Overhead di comunicazioni

- Parallelismo a grana grossa
  - Molti calcoli tra le comunicazioni

  → rapporto tra il calcolo e le comunicazioni è grande
  - Può essere difficile bilanciare il carico
  - Probabile aumento delle performance