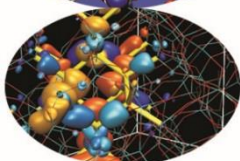
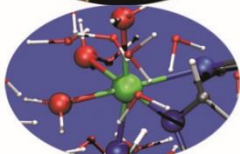
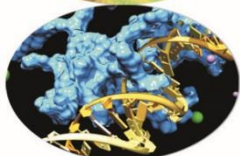
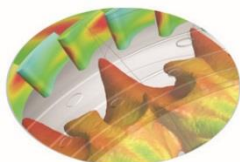


Elementi di Programmazione Parallela

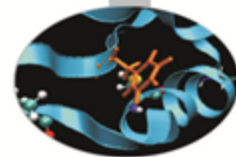


Claudia Truini
c.truini@cineca.it

Vittorio Ruggiero
v.ruggiero@cineca.it

Isabella Baccarelli
i.baccarelli@cineca.it

Sommario



Analysis & design delle applicazioni parallele

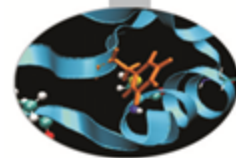
Tecniche di partizionamento

Load balancing

Comunicazioni

Misura delle prestazioni parallele

Primi passi: analizzare il problema



Prima di iniziare a scrivere un codice parallelo:

- ✦ Capire il problema che si vuole parallelizzare
- ✦ Capire il codice seriale (se esiste)
- ✦ Capire se il problema può essere parallelizzato:

- ✦ Problema parallelizzabile: esempio

- ✦ Serie geometrica:

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

- ✦ I calcoli sono indipendenti → possono essere calcolati in parallelo

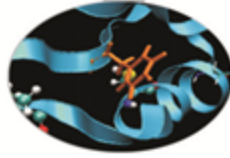
- ✦ Problema non parallelizzabile: esempio

- ✦ Successione di Fibonacci:

$$F(n) = F(n-1) + F(n-2) \quad F(1) = 1 \quad F(2) = 1$$

- ✦ I calcoli sono dipendenti: per calcolare il valore $F(n)$ devo conoscere i due valori precedenti $F(n-1)$ e $F(n-2)$ → non possono essere calcolati in parallelo

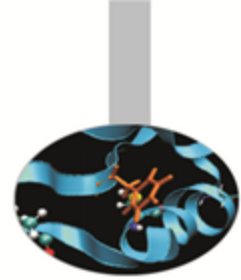
Primi passi: costruire l'applicazione parallela (1)



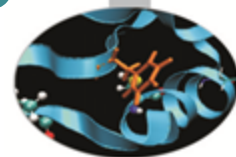
Se il codice può essere parallelizzato:

- ✦ Utilizzare ‘profilers’ (ad es. gprof) per identificare le parti maggiormente pesanti dal punto di vista computazionale (hotspots) e su quelle focalizzare l’attenzione
 - ✦ generalmente nelle applicazioni tecnico-scientifiche la maggior parte del tempo è spesa in poche sezioni di codice (spesso *loop*)
- ✦ Identificare i possibili “colli di bottiglia” (bottlenecks): provare a eliminarli o ridurli
- ✦ Identificare inibitori al parallelismo (ad. es. dipendenza dei dati)
- ✦ Studiare differenti algoritmi
- ✦ Utilizzare, se possibile, librerie parallele ottimizzate (PETSc, MKL, AMD, etc.)

Primi passi: costruire l'applicazione parallela (2)

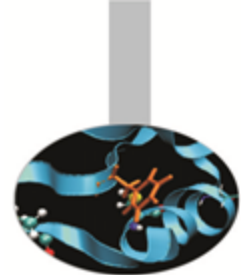


- ⌚ Discretizzare il dominio usando una mesh o particelle
- ⌚ Dividere il lavoro (e la memoria) tra gli N processi
- ⌚ Distribuire i dati in modo da
 - ⌚ bilanciare il carico di lavoro di ogni processo, misurato come tempo di calcolo (Load Balancing)
 - ⌚ ridurre le comunicazioni tra i processi
- ⌚ Gestire le comunicazioni e la sincronizzazione tra i processi paralleli



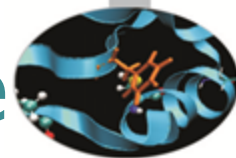
- ☛ Si divide il problema in n parti, ciascuna parte viene assegnata ad un processo differente
- ☛ Distribuzione del lavoro tra più soggetti:
 - ☛ tutti eseguono le stesse operazioni, ma su un sottoinsieme di dati
 - ☛ → partizionamento dei dati
- ☛ Distribuzione delle funzioni tra più soggetti:
 - ☛ non tutti eseguono le stesse operazioni
 - ☛ → decomposizione funzionale

Partizionamento dei dati

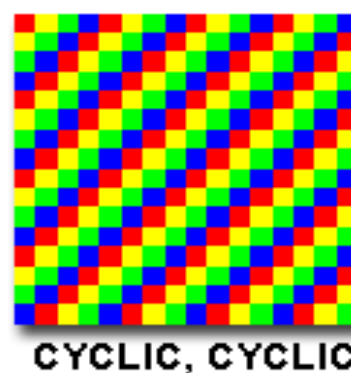
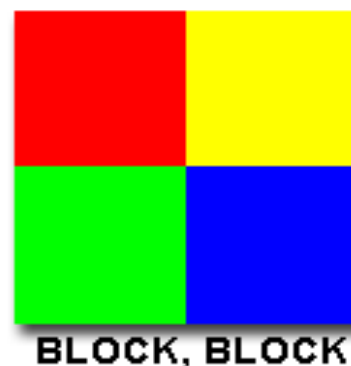
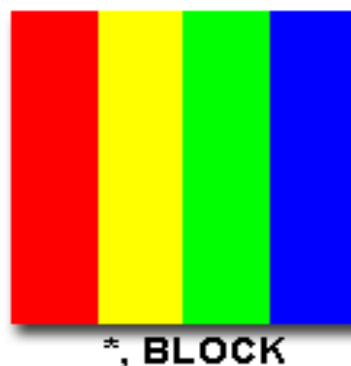
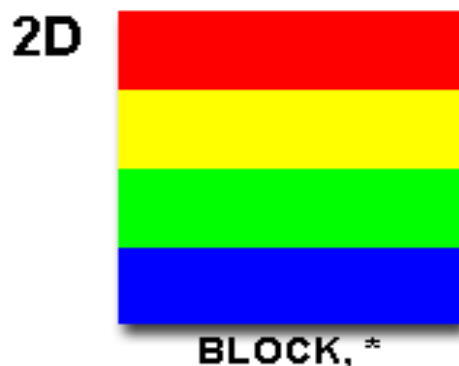


- ⌚ Insieme (ordinato) di dati da elaborare tutti allo stesso modo
- ⌚ Decompongo i dati associati al problema
- ⌚ Ogni processo esegue le stesse operazioni su una porzione dei dati (→ SPMD)
- ⌚ **PRO:** scalabile con la quantità di dati
- ⌚ **CONTRO:** vantaggiosa solo per casi sufficientemente grandi

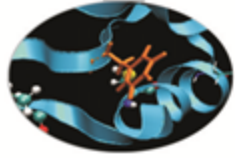
Partizionamento dei dati: esempio con griglie strutturate



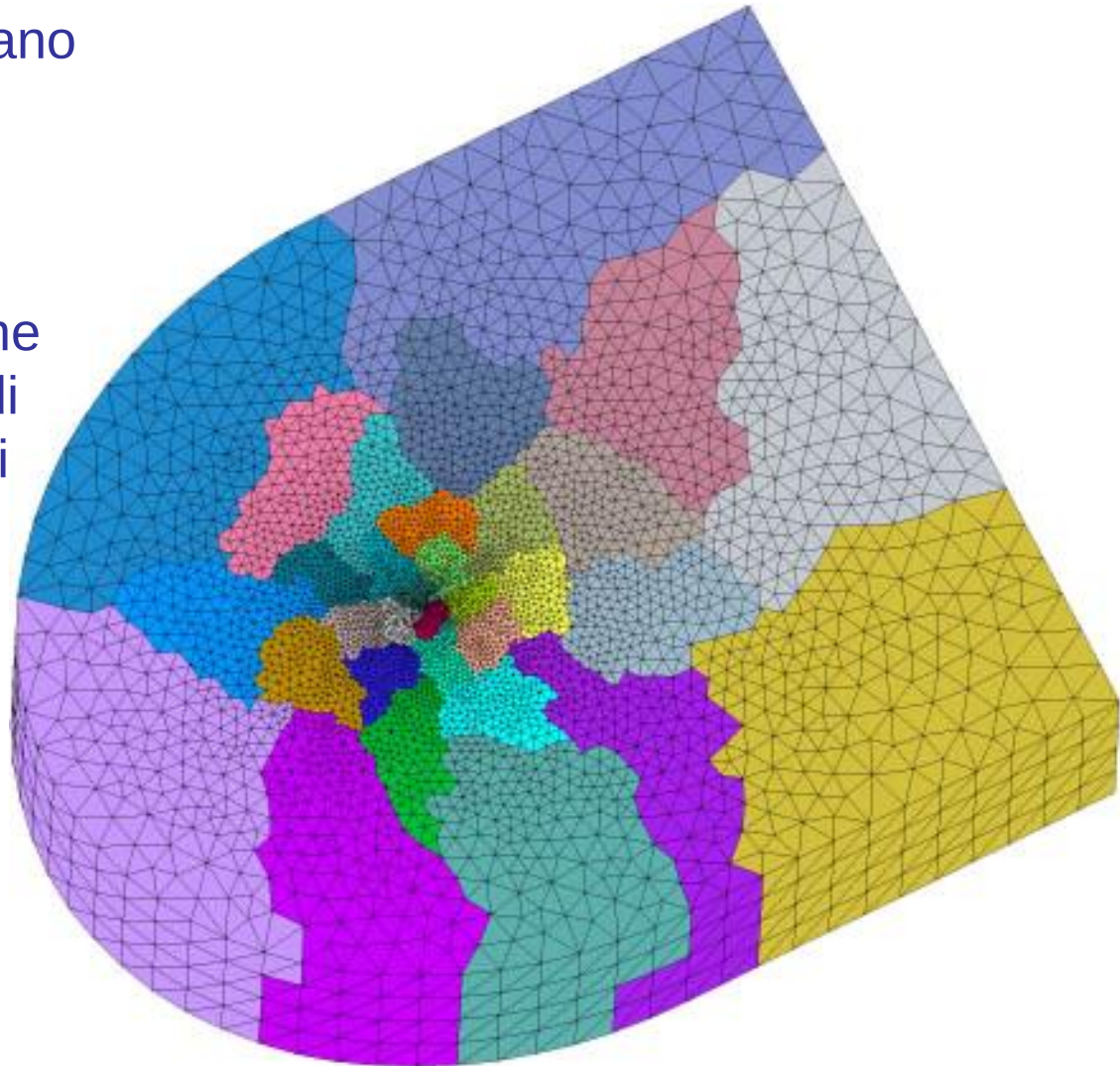
- Vari modi possibili per partizionare i dati
- Ogni processo lavora su una porzione di dati



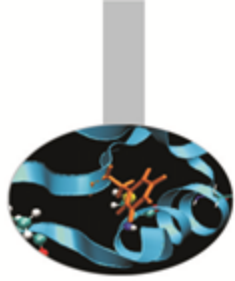
Partizionamento dei dati: esempio con griglie non strutturate



- ☛ Frequente quando si usano algoritmi con
 - ☛ elementi finiti
 - ☛ volumi finiti
- ☛ Si possono usare tool che si basano sulla tecnica di partizionamento dei grafi
 - ☛ Metis
 - ☛ Parmetis
 - ☛ Scotch

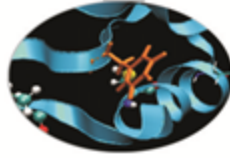


Decomposizione funzionale

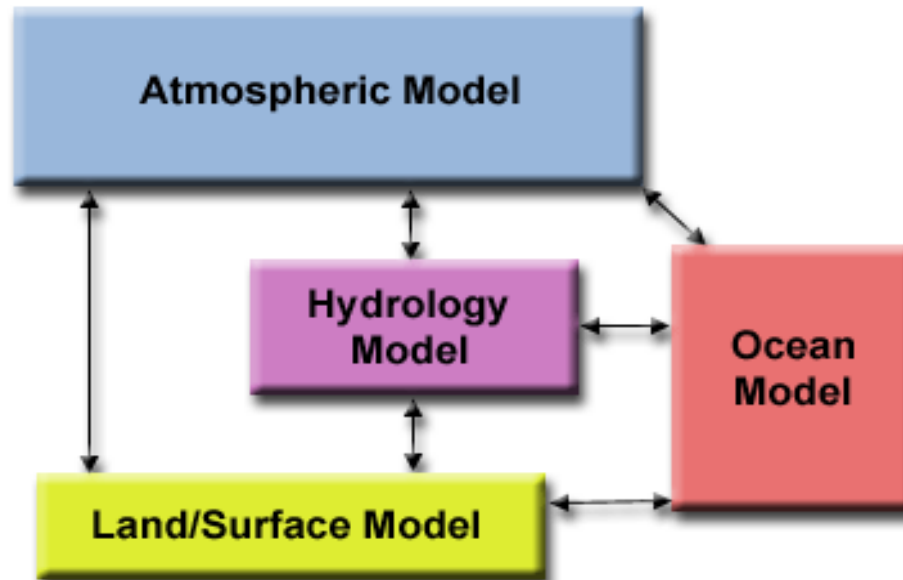


- † Insieme di elaborazioni differenti ed indipendenti
- † Decompongo il problema in base al lavoro che deve essere svolto
- † Ogni processo prende in carico una particolare elaborazione (→MPMD)
- † **PRO**: scalabile con il numero di elaborazioni indipendenti
- † **CONTRO**: vantaggiosa solo per elaborazioni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

Decomposizione funzionale: esempio

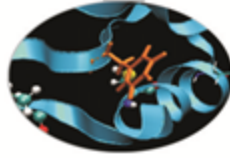


Modello climatico



- 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

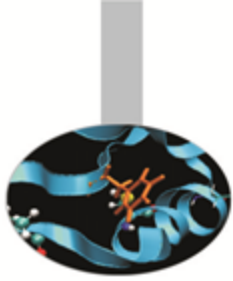
- ✦ Esempio con 4 processi:

- ✦ tempo seriale = 40
- ✦ tempo parallelo (teorico) = 10
- ✦ tempo parallelo (reale) = 12
→ 20% più lento



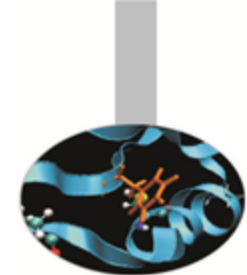
- ✦ La velocità dell'esecuzione dipenderà dal processo più lento

Load balancing: assegnazione statica



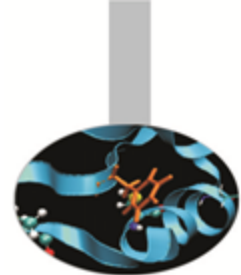
- Partizionare ugualmente il lavoro di ogni processo
- Per operazioni su array/matrici dove ogni processo svolge operazioni simili, distribuisco uniformemente i dati sui processi
- Per cicli dove il lavoro fatto in ogni iterazione è simile, distribuisco le iterazioni uniformemente sui processi

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

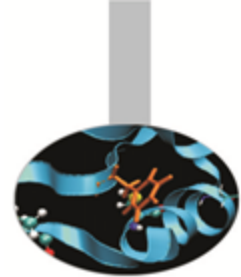
Load balancing: confronti



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

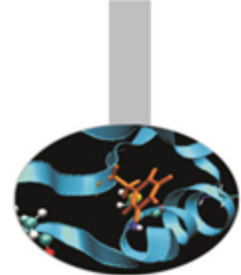
- 🔧 Assegnazione dinamica:
 - 📌 ☺ può curare problemi di sbilanciamento
 - 📌 ☹ introduce un overhead dovuto alla gestione del bilanciamento

Comunicazioni



- 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

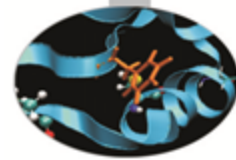
...e la rete?



- ✦ 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:

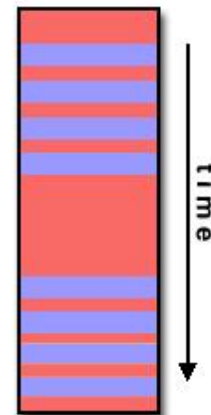
$$T = L + N/B$$

Granularità



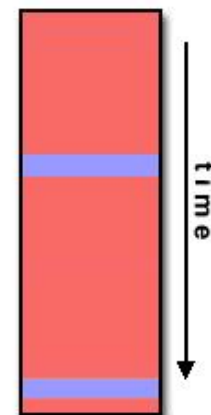
Parallelismo a grana fine

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



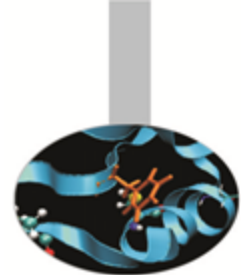
Parallelismo a grana grossa

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



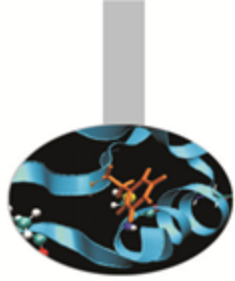
communication
computation

Input e Output



- ⌚ Le operazioni di I/O sono generalmente seriali
- ⌚ Possono creare “colli di bottiglia”
- ⌚ La gestione dell’I/O porta all’utilizzo di costrutti specifici del linguaggio e del modello di programmazione usato:
 - ⌚ Gather/Scatter per rimettere insieme/distribuire i dati su modelli message-passing (MPI)
 - ⌚ Utilizzo di un solo thread nel modello di programmazione multi-threaded (OpenMP)
- ⌚ Utilizzo di specifiche librerie: MPI-IO, HDF5, netCDF, etc.

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

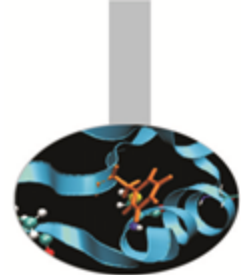
📌 $T(n)$ = tempo totale usando n processi

📌 $S(n) = n$ → 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

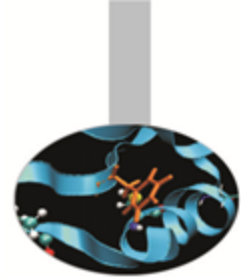
🔧 $E(n) < 1$ → caso reale

🔧 $E(n) \ll 1$ → 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:

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

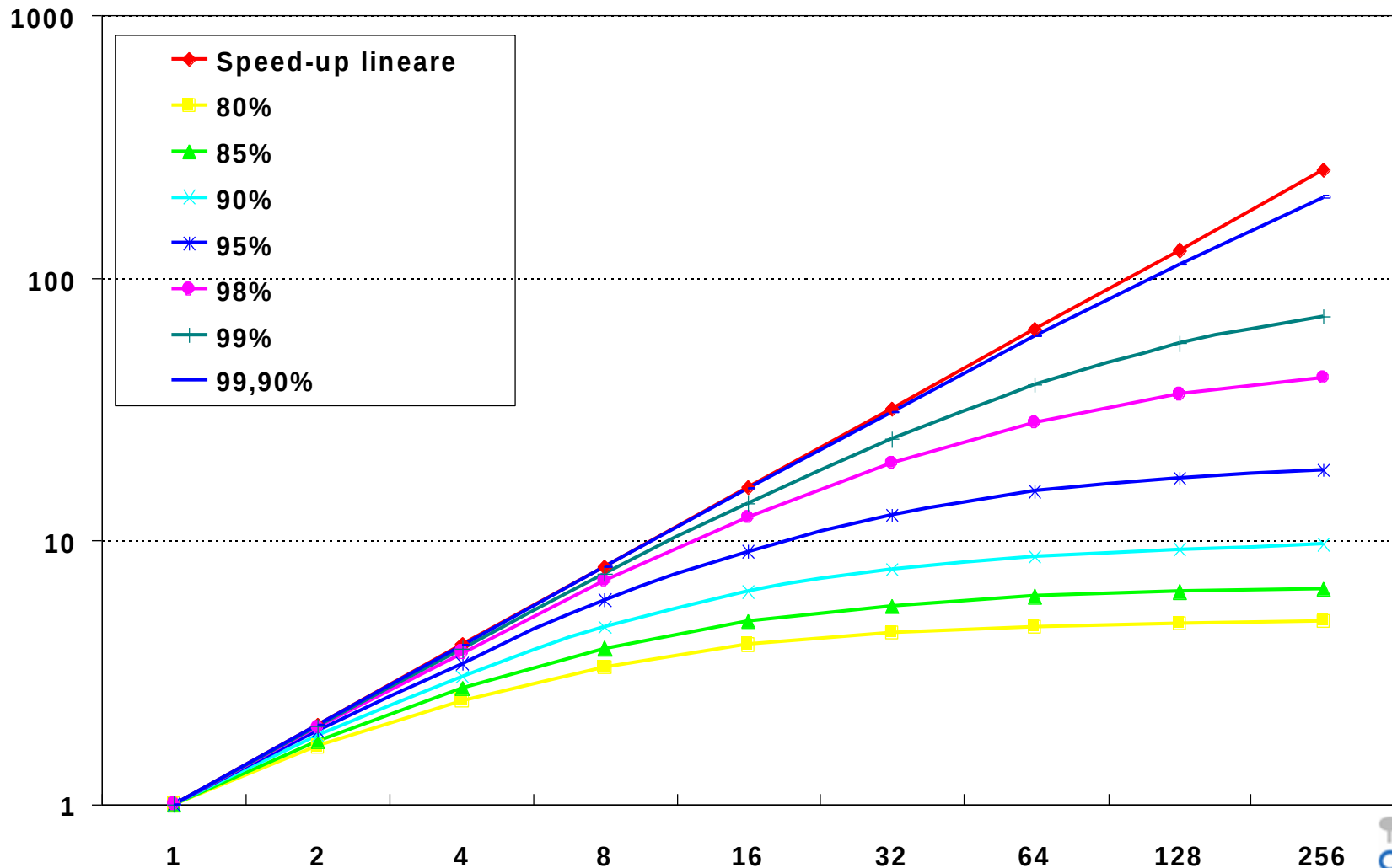
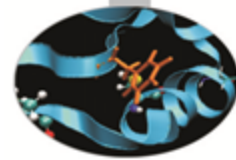
- ☛ Speed-up su n processi:

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

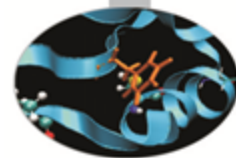
- ☛ Per n grande $S(n) = (F + P)/F$

- ☛ esiste un limite asintotico allo speed-up!

Legge di Amdhal: Speedup



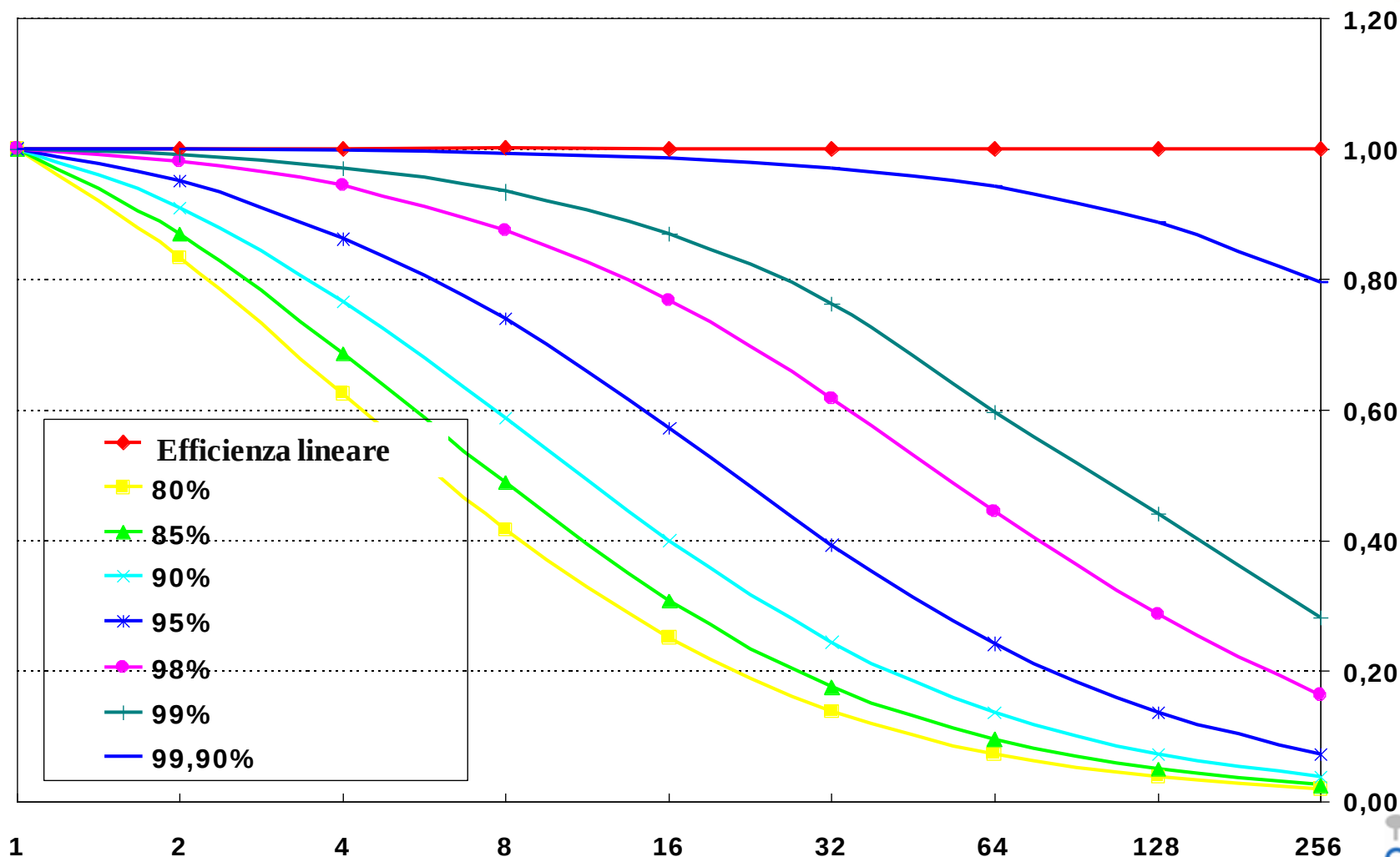
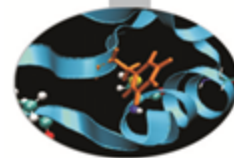
Legge di Amdhal: Speedup



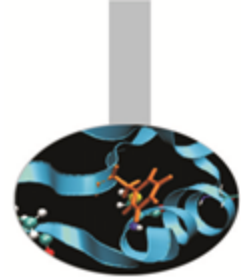
☛ Massimo speed-up raggiungibile in funzione della percentuale della parte parallela

	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

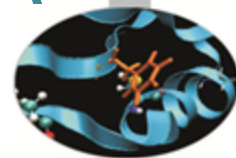
Legge di Amdhal: Efficienza



Legge di 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



- 🔧 tool per l'analisi delle performance di applicazioni parallele: MPI, OpenMP, MPI-OpenMP
- 🔧 open-source
- 🔧 può essere installato su tutte le architetture usate per l'HPC
- 🔧 analizza codici scritti in Fortran, C, C++
- 🔧 Il processo di analisi è diviso in tre passi:
 - 🔧 Compilazione:
skin [opzioni] compilatore [opzioni] <codice_sorgente>
 - 🔧 Esecuzione:
scan [opzioni] mpirun [opzioni] <codice_eseguibile>
 - 🔧 Analisi:
square [opzioni] scorep_[caratteristiche]
- 🔧 www.scalasca.org