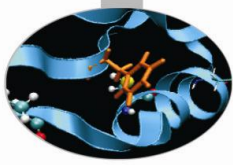
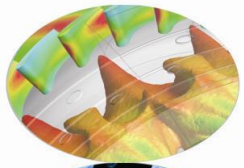
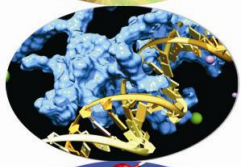
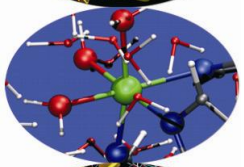
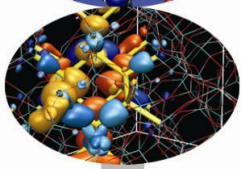
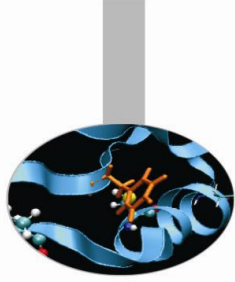


Introduzione

Introduzione al calcolo parallelo



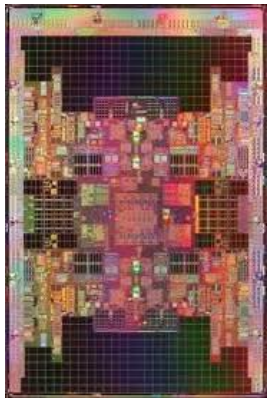
Aumento delle prestazioni dei chip

La capacità computazionale dei processori aumenta continuamente!

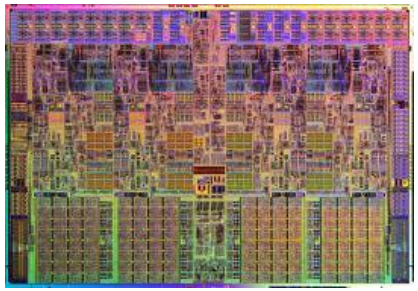
- Transistor più piccoli => maggiore densità dei circuiti nei processori
- Maggiore densità di transistor => aumento della velocità di calcolo
- Velocità di calcolo elevate => aumento di calore e dei consumi energetici

i486 (1989)
***1,2M** transistors*

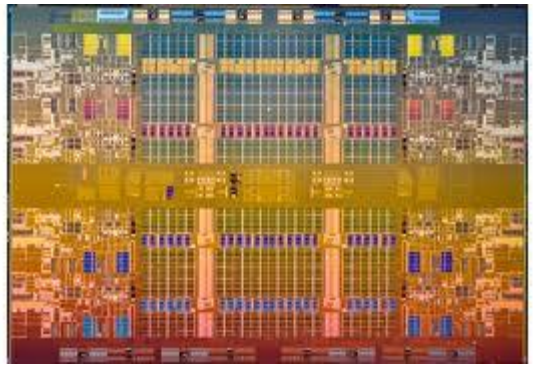
itanium2 (2003)
***220M** transistors*

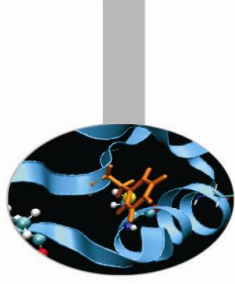


Xeon nehalem (2007)
***781M** transistors*



Xeon nehalem-ex (2011)
***2300M** transistors*

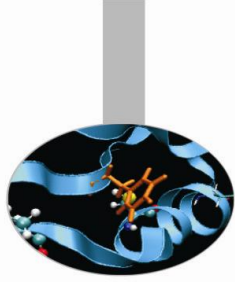




Come sfruttare la densità dei transistor?

Aumentando la velocità computazionale aumenta il calore.
Il surriscaldamento fa diminuire le prestazioni dei chip (e li può danneggiare!)

Non è possibile aumentare la capacità di dissipazione degli attuali sistemi di raffreddamento dei processori (air/water cooling) tanto quanto servirebbe
C'è quindi un limite fisico oltre il quale non è possibile andare, anche se tecnicamente è possibile ridurre le dimensioni dei circuiti integrati ed aumentare ancora la velocità dei processori

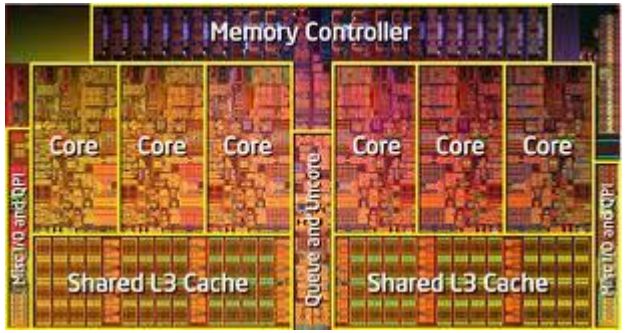


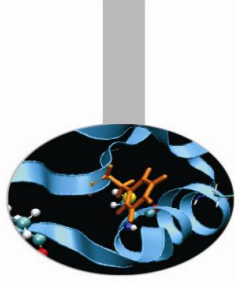
Usiamo sistemi paralleli

Possiamo aumentare la capacità computazionale attraverso il **parallelismo**!

Anzichè produrre processori single-core monolitici più veloci e più complicati, è meglio aumentare il numero di processori sul singolo chip.

I processori **multicore** hanno quindi più CPU su cui svolgere i calcoli.





Perché abbiamo bisogno di maggiore capacità di calcolo?

L'aumento delle prestazioni dei processori ha permesso di affrontare aspetti della ricerca scientifica finora poco praticabili.

Maggiori capacità di calcolo danno la possibilità di risolvere problemi estremamente complessi e/o di grandi dimensioni:

Fluidodinamica

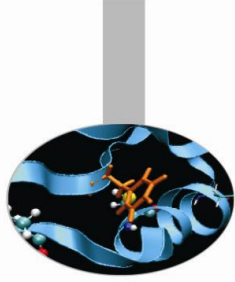
Modellazione climatica

Analisi delle proteine

Scoperta nuovi farmaci

Ricerca energetica

Analisi di grandi quantità di dati



Aree di ricerca complesse

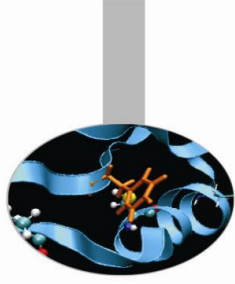
Modellazione Climatica

Per comprendere meglio gli attuali cambiamenti climatici è necessario elaborare dei modelli estremamente complessi.

I modelli devono interpretare le interazioni tra atmosfera, oceani, terre emerse, ghiacci polari, temperature, correnti, ecc...

Analisi delle proteine

Le proteine sono molecole estremamente complesse, che coinvolgono numerosi aspetti della nostra vita e delle malattie. Studiare le configurazioni possibili delle proteine è estremamente costoso in termini computazionali e da questi studi dipendono le ricerche sulle terapie di malattie come Huntington, Parkinson, Alzheimer



Aree di ricerca complesse

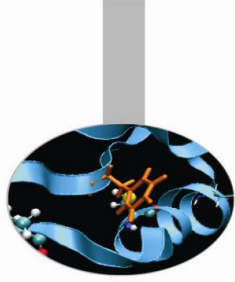
Scoperta di nuovi farmaci

Studiare le interazioni delle molecole farmaceutiche con i sistemi del corpo umano permette la creazione di farmaci efficaci. Ad esempio, attraverso l'analisi del genoma dei pazienti è possibile capire se i farmaci per uno specifico trattamento possono essere realmente utili nella terapia, diminuendo così il numero di trattamenti inefficaci e gli effetti collaterali.

Ricerca energetica

La disponibilità di potenze di calcolo crescenti permette la progettazione e la modellazione sempre più dettagliata di tecnologie per la produzione e l'uso di energia, come turbine eoliche, celle solari e batterie. Questi studi sono necessari per aumentare l'efficienza nella produzione e nell'uso di energia

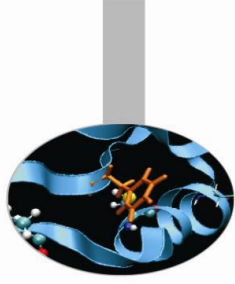
Aree di ricerca complesse



Data analysis

La capacità mondiale di salvataggio dei dati raddoppia ogni 2 anni, ma questa enormità di informazioni non viene quasi mai elaborata. Un esempio può essere rappresentato dal basso utilizzo delle sequenze di nucleotidi del genoma umano. La comprensione di come queste sequenze incidono sull'evoluzione della vita e delle malattie umane richiede una estensiva fase di analisi dei dati.

Un secondo esempio è fornito dal recente LHC del CERN, un vastissimo progetto di sperimentazione che genererà enormi quantità di dati potenzialmente utili nei campi di ricerca astronomica, fisica, medica. Tutti questi dati, prima di poter essere interpretati, necessitano di una fase di elaborazione computazionale.



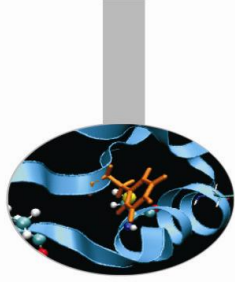
Come si scrivono i programmi paralleli?

In generale si parte dal concetto di **partizionamento** del problema.

Infatti un programma parallelo implica sempre la suddivisione del problema, dandoci modo di distribuire il calcolo tra i processori a nostra disposizione.

Si distinguono due approcci principali:

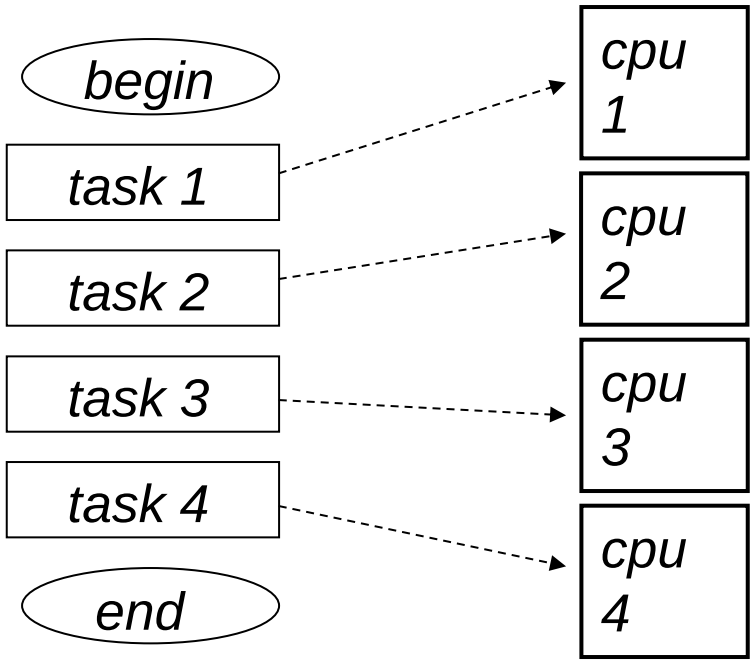
- parallelismo a livello di **task**
- parallelismo a livello di **dati**

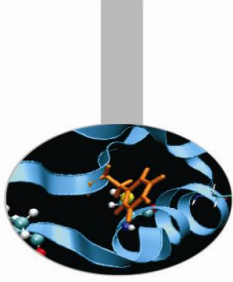


Task parallelism

Il parallelismo a livello di task si basa sul principio della ripartizione delle operazioni di un algoritmo tra i processori disponibili.

Se un algoritmo di un programma prevede una serie di operazioni differenti, possiamo parallelizzarlo assegnando ad ogni processore una serie di differenti operazioni

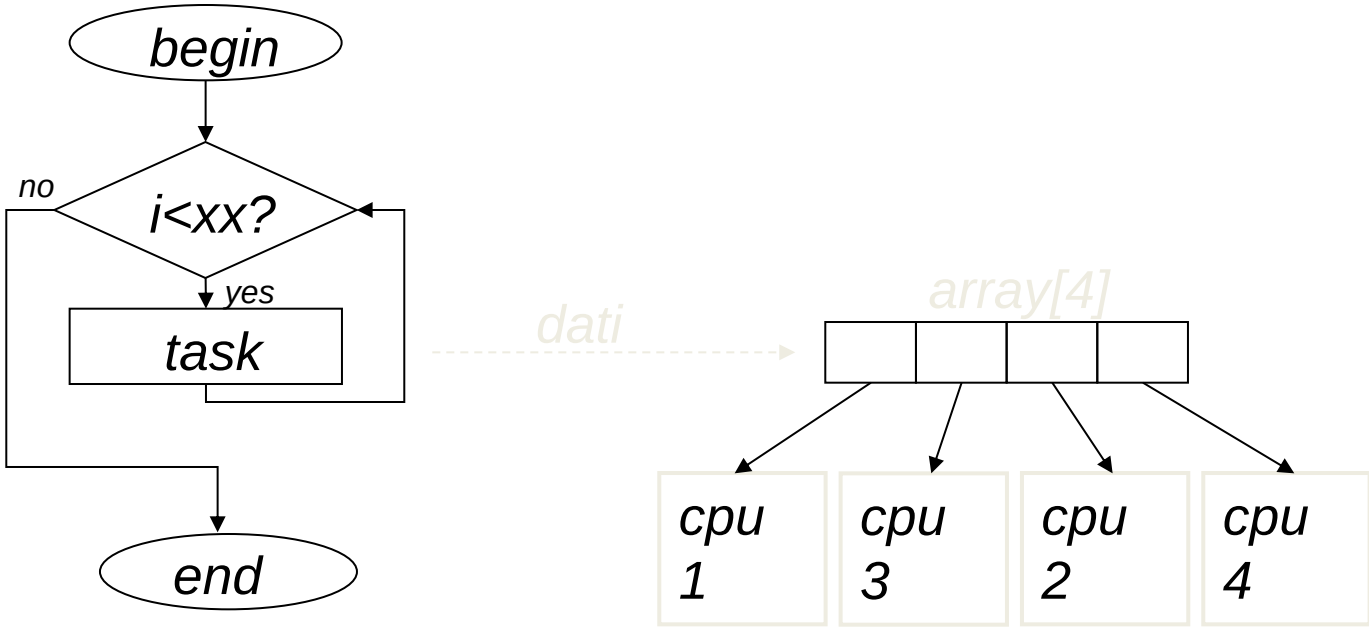




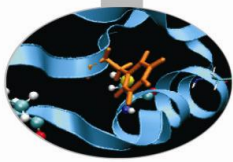
Data parallelism

Il parallelismo a livello di dati (o data parallelism) consiste nel ripartire tra le varie cpu il carico dei dati da elaborare.

Il calcolo eseguito dalle cpu è (quasi) sempre lo stesso, cambiano i dati che vengono letti e scritti dalle unità computazionali.



Parallelo, concorrente e distribuito



Qual'è la differenza tra programmazione parallela, concorrente e distribuita?

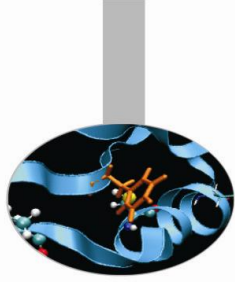
Un programma è **concorrente** quando possiamo ritrovare threads multipli generati da esso in fase di svolgimento di calcolo.

Un programma si dice **parallelo** quando threads multipli cooperano strettamente nella computazione di un problema.

Un programma è **distribuito** quando altri programmi, o sezioni indipendenti di esso, concorrono in maniera cooperativa alla risoluzione del problema.

Tali definizioni non sono univoche, dipendono dall'interpretazione dell'autore. In questo corso facciamo riferimento alla definizione descritta in "An introduction to parallel programming", P. Pacheco

Parallelo, concorrente e distribuito

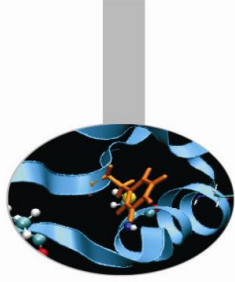


Stando a quanto detto in precedenza, i programmi paralleli e distribuiti sono programmi concorrenti, in quanto i loro threads possono essere in fase di computazione in ogni istante.

Generalmente un programma parallelo viene eseguito su un'insieme di processori che condividono tra loro la memoria (oppure sono connessi con una rete ad altissima velocità) e sono fisicamente vicini tra loro.

I programmi distribuiti invece sono eseguiti su sistemi ad elevata distribuzione (distanza) geografica e le componenti del programma sono considerate indipendenti tra loro.

Processi, threads e multitasking

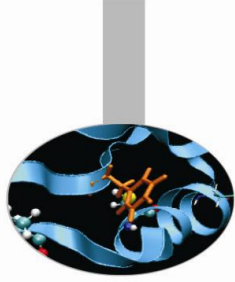


I sistemi operativi sono programmi che si occupano di gestire le risorse hardware e software di un computer. Controllano l'allocazione della memoria e determinano la sequenza e le modalità di esecuzione dei programmi, determinando l'accesso concorrente alle periferiche (hard disks, rete, altre periferiche...)

Quando un programma viene eseguito, il sistema operativo crea un **processo**, un'istanza del programma costituito da:

- codice macchina eseguibile
- una partizione della memoria, costituita da stack, heap e altre aree di memoria
- una descrizione delle risorse allocate per l'esecuzione del programma
- informazioni di sicurezza e di accesso all'hardware e al software
- informazioni sullo stato del processo (se è in stato di esecuzione, di attesa della disponibilità di una risorsa necessaria, sullo stato della memoria, ecc.)

Processi, threads e multitasking

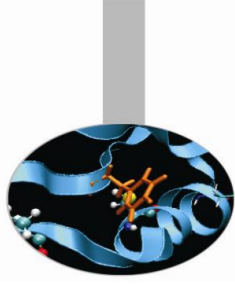


I moderni sistemi operativi sono detti **multitasking** in quanto capaci di gestire l'esecuzione contemporanea di più programmi.

Questo si traduce nella capacità del sistema operativo di gestire le richieste dei programmi, a partire dalle risorse da essi richieste. Se una risorsa non è immediatamente disponibile il programma viene bloccato finchè la risorsa si libera.

In ogni caso il processo può continuare ad eseguire altre parti di codice indipendenti dalla richiesta effettuata: si dice quindi che viene suddiviso in **thread**, cioè dei task (compiti) eseguibili in maniera indipendente.

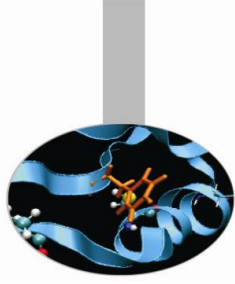
I thread di un processo condividono tutte le informazioni del processo, incluse risorse allocate, memoria e periferiche assegnate.



Interazione fra processi

Le interazioni tra processi sono riconducibili a diversi tipi:

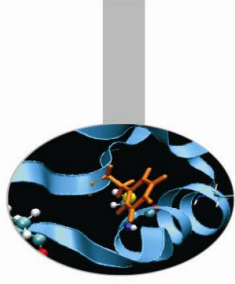
- Cooperazione
- Competizione
- Interferenza
- Mutua esclusione
- Deadlock



Cooperazione

E' una forma di interazione *prevedibile e desiderata* durante la quale avviene uno scambio di informazioni fra i processi; si può scambiare un semplice segnale oppure può avvenire un trasferimento di dati vero e proprio.

L'interazione fra i processi consiste in una loro *sincronizzazione* e, se c'è trasferimento di dati, in una *comunicazione.*



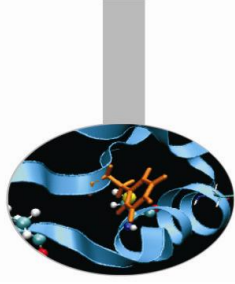
Competizione

E' una forma di interazione *prevedibile e non desiderata ma necessaria*; si presenta quando azioni eseguite da processi differenti hanno la necessità di operare su risorse comuni che non possono essere usate contemporaneamente da più processi (es. aggiornamento di una variabile condivisa da più processi). Occorre gestire in *mutua esclusione* le operazioni sulle risorse comuni.

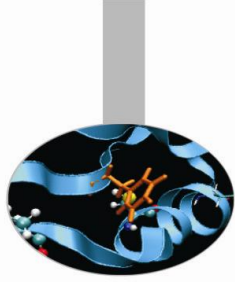
Anche nel caso della competizione l'interazione fra i processi si estrinseca in un vincolo di precedenza tra eventi di processi diversi ma la natura di questo vincolo è però diversa da quella dei vincoli imposti dalla cooperazione.

Si parla di *sincronizzazione diretta o esplicita* per indicare i vincoli imposti dalla cooperazione, di *sincronizzazione indiretta o implicita* per indicare i vincoli imposti dalla competizione.

Interferenza



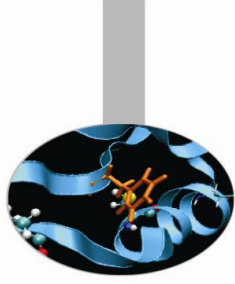
E' una forma di interazione non prevista e non desiderata ed è in genere provocata da errori di programmazione nella preparazione del programma parallelo. Gli errori possono essere dovuti all'inserimento di interazioni non richieste dalla natura del problema oppure ad interazioni necessarie ma non gestite correttamente.
Si tratta di errori dipendenti dal tempo in quanto eventuali effetti indesiderati possono manifestarsi oppure no, dipendendo dai rapporti di velocità tra i processi.



Mutua esclusione

Il problema si presenta quando più processi per volta possono accedere ad un insieme di variabili comuni ovvero a risorse che i processi possono richiedere contemporaneamente, tipico del modello a memoria comune.
In questi casi la sequenza di istruzioni con la quale un processo *accede* e *modifica* un insieme di variabili comuni prende il nome di *sezione critica*.

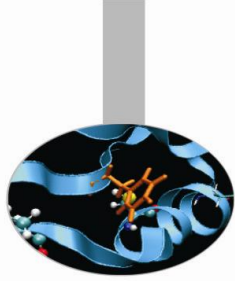
Le sezioni critiche assicurano la mutua esclusione nel tempo dei processi, ovvero un solo processo alla volta può eseguire il codice racchiuso all'interno della sezione critica.



Deadlock

Questa è la situazione indesiderata in cui uno o più processi rimangono indefinitamente bloccati perché non possono verificarsi le condizioni necessarie per il loro proseguimento.

Un gruppo di processi entra in **deadlock** quando tutti i processi del gruppo attendono un evento (acquisizione o rilascio di risorse) che può essere causato solo da un altro dei processi in attesa.



Prestazioni dei programmi paralleli

L'obiettivo dei programmi paralleli è quindi quello di aumentare le prestazioni.

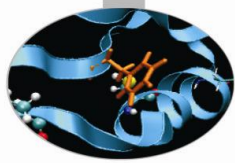
Idealmente possiamo suddividere equamente i compiti svolti dal programma tra i core a disposizione. Indichiamo con *Tseriale* il tempo di esecuzione seriale del progamma e con *Tparallelo* il tempo di esecuzione parallela.

Se abbiamo a disposizione *p* cores su cui eseguire il programma, allora possiamo aspettarci un tempo di esecuzione parallelo così calcolato:

$$T_{parallelo} = \frac{T_{seriale}}{p}$$

Se questo rapporto è vero, cioè che il tempo di esecuzione si riduce proporzionalmente rispetto al numero di cores impiegati, si dice che il programma ha uno **speedup lineare**.

Speed-up ed efficienza

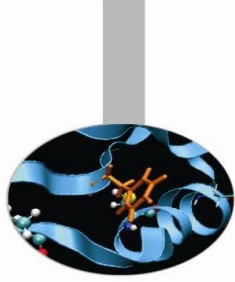


In realtà uno speedup lineare è difficile, se non impossibile da ottenere: dobbiamo considerare alcuni tempi accessori (overhead) causati dalla suddivisione in thread e processi multipli del programma. Inoltre esistono problemi legati all'accesso di risorse condivise (memoria e periferiche) e alla trasmissione dei dati (bus, network) che emergono solo in caso di un'esecuzione parallela, mentre nel caso seriale queste criticità non esistono o sono estremamente ridotte (1 solo processo!).

La definizione di speedup è la seguente:

$$S = \frac{T_{seriale}}{T_{parallelo}}$$

Nel caso di speedup lineare abbiamo S=p dove p è il numero di cores o processori.

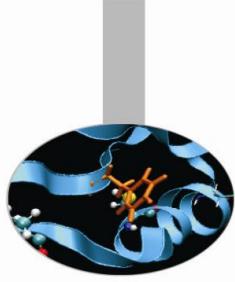


Speed-up ed efficienza

A causa delle limitazioni dette in precedenza, lo speedup lineare è difficilmente raggiungibile ed inoltre il rapporto tra speedup e numero di cores decresce con l'aumento di p. Ciò significa che aumentando il numero di processori diminuisce lo speedup garantito dal singolo processore, proprio perché aumentano i tempi di overhead del programma.

L'efficienza è il rapporto tra speedup e numero di cores utilizzati:

$$E = \frac{S}{p} = \left(\frac{\frac{T_{seriale}}{T_{parallelo}}}{p} \right) = \frac{T_{seriale}}{p \cdot T_{parallelo}}$$

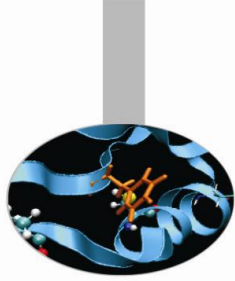


Overhead

I tempi di overhead sono quindi un problema rilevante nei programmi paralleli e questo influenza notevolmente le prestazioni del calcolo.

Se consideriamo i tempi di overhead possiamo calcolare il tempo di esecuzione parallela *Tparallelo* in questo modo:

$$T_{parallelo} = \frac{T_{seriale}}{p} + T_{overhead}$$



Dimensioni del problema

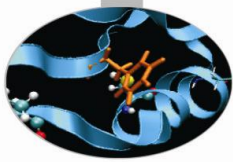
Un altro aspetto importante è la dimensione del problema. Se il numero di cores ed i tempi di overhead influenzano lo speedup del programma, allo stesso modo le dimensioni dei dati utilizzati per il calcolo o la grandezza stessa del calcolo possono influenzare le prestazioni.

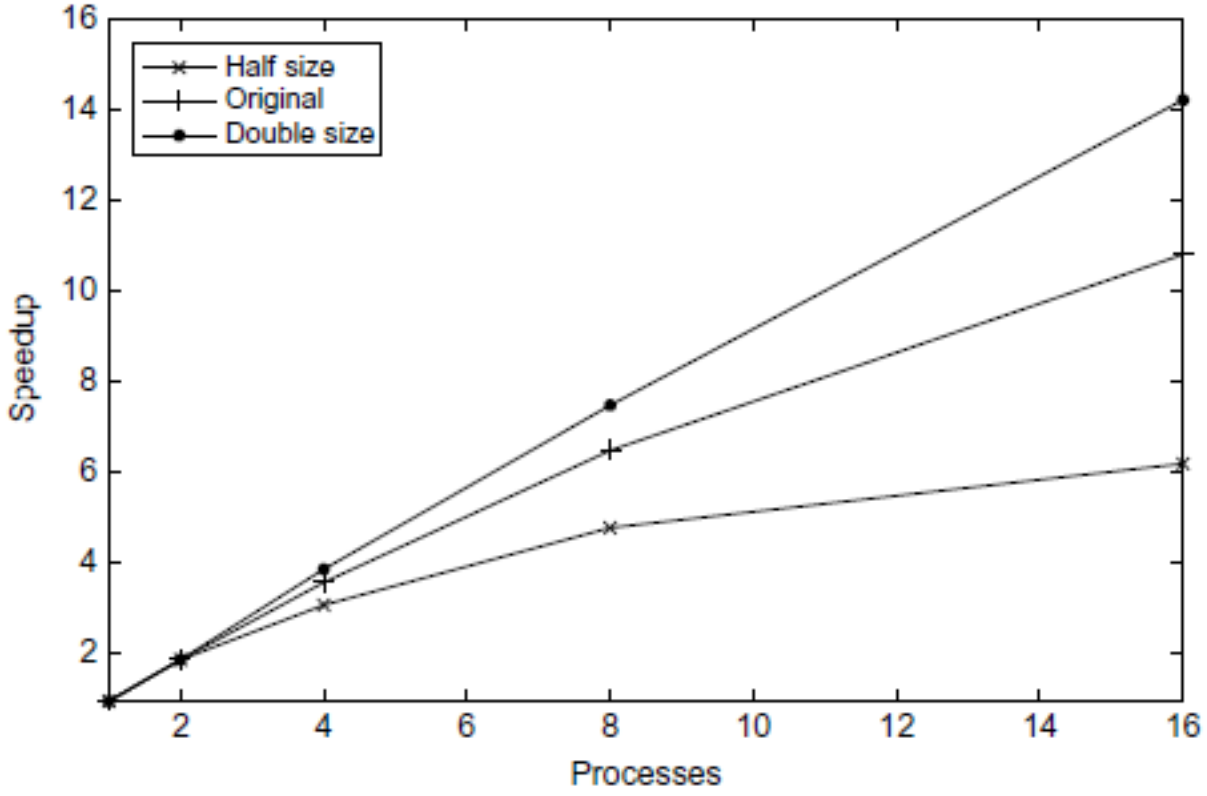
Generalmente, più le dimensioni del problema aumentano, maggiore sarà lo speedup al crescere del numero di cores.

Questo concetto è evidente se consideriamo che la suddivisione del carico di lavoro su più processori introduce sempre dei tempi di overhead. Se aumentiamo la dimensione del problema, ogni processore aumenterà i tempi di calcolo o la quantità di dati calcolati, ma i tempi di overhead rimarranno pressappoco gli stessi!

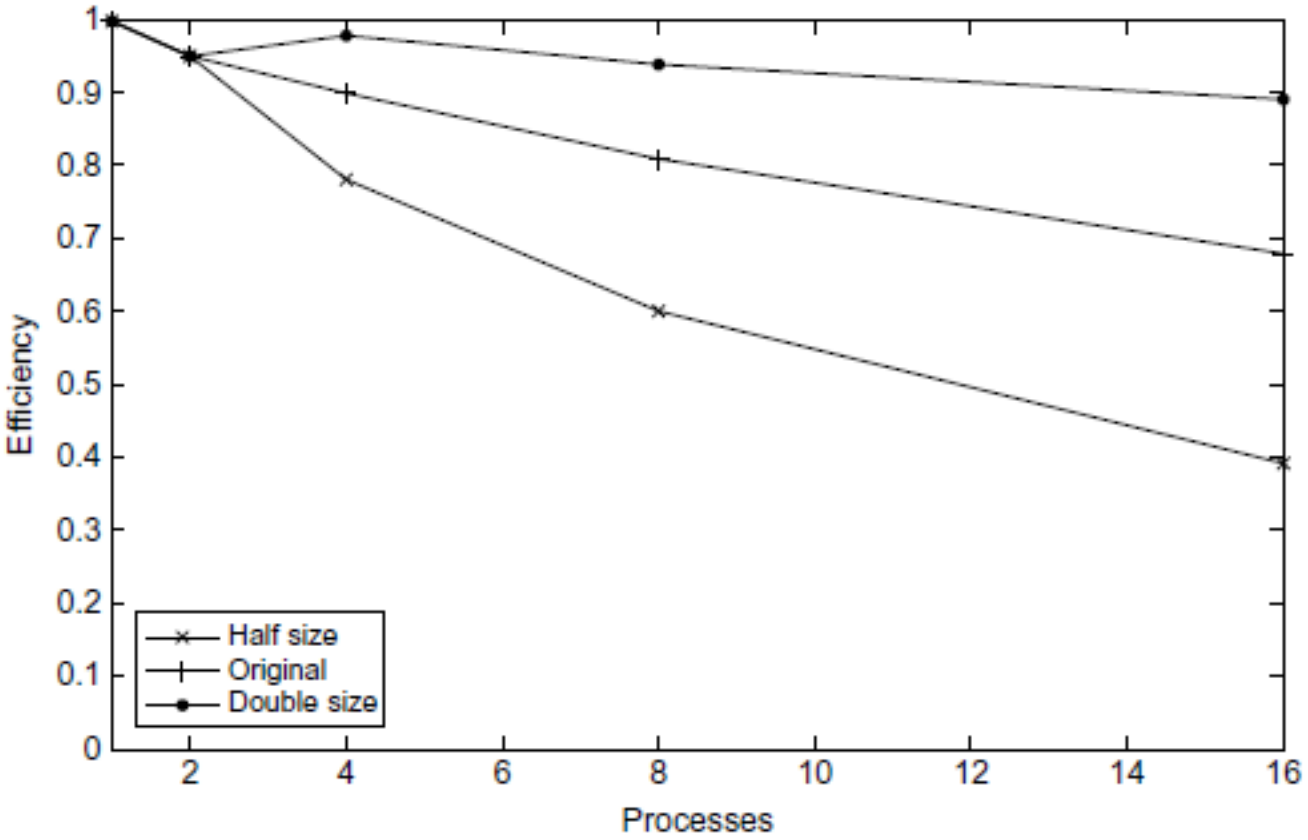
Sfruttando questo aspetto possiamo quindi aumentare lo speedup.

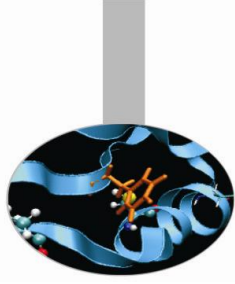
Speedup e dimensioni del problema





Efficienza e dimensioni del problema





Legge di Amdahl

Ci dà un'idea dello speedup massimo teorico di un programma parallelo.

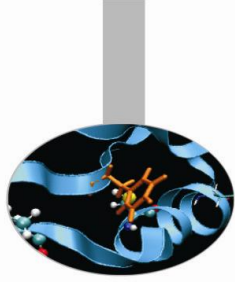
Ipotizziamo di parallelizzare il 90% di un'applicazione in maniera perfetta, ed il restante 10% rimane sequenziale; poniamo che il tempo di esecuzione seriale sia Tseriale = 20 sec. Il tempo di esecuzione parallelo sarà

Tparallelo = (0.9xTseriale)/p + 0.1xTseriale = 18/p + 2

Lo speedup sarà *S=20/(18/p + 2).*

Da questo possiamo dedurre che più p (il numero di cores) cresce, più la parte parallela *(0.9xTseriale)/p* tenderà a 0. Il tempo parallelo ideale non potrà quindi essere minore di *0.1 x Tseriale= 2 sec*.

Svolti i calcoli abbiamo che lo **speedup massimo è S≤10**.

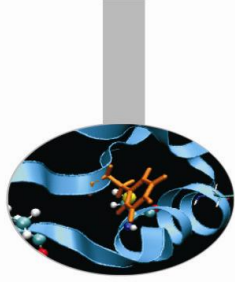


Legge di Amdahl

La legge di Amdahl dice quindi che la frazione non parallelizzabile di un programma, che chiamiamo *r*, influenza lo speedup massimo possibile, che sarà sempre minore di *1/r*.

Non preoccupiamoci troppo però!

Questa legge non tiene in considerazione molti aspetti dei programmi paralleli, ed il più importante tra questi è la **dimensione del problema**.



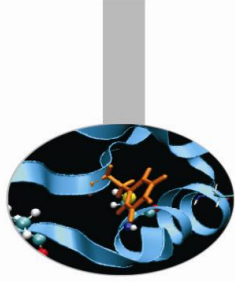
Scalabilità

Per valutare la scalabilità di un programma dobbiamo tentare di aumentare il numero di cores/thread e la dimensione del problema in modo che l'efficienza del programma E non diminuisca.

In altre parole, un programma si definisce **scalabile** quando all'aumentare del numero di cores/threads e della dimensione del problema, l'efficienza del programma rimane pressoché costante.

Se un programma mantiene fissa l'efficienza E, aumentando il numero di processori/threads senza aumentare la dimensione del problema, esso si definisce **fortemente scalabile**. Al contrario, se per mantenere fissa l'efficienza del programma dobbiamo aumentare il numero di cores/threads con lo stesso rapporto con cui aumentiamo la dimensione del problema, esso viene detto **debolmente scalabile**.

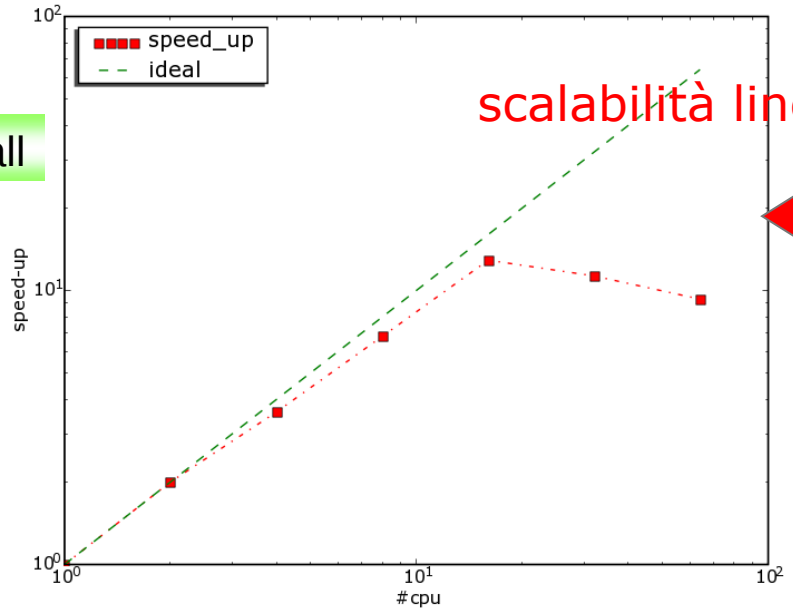
Test Fluent

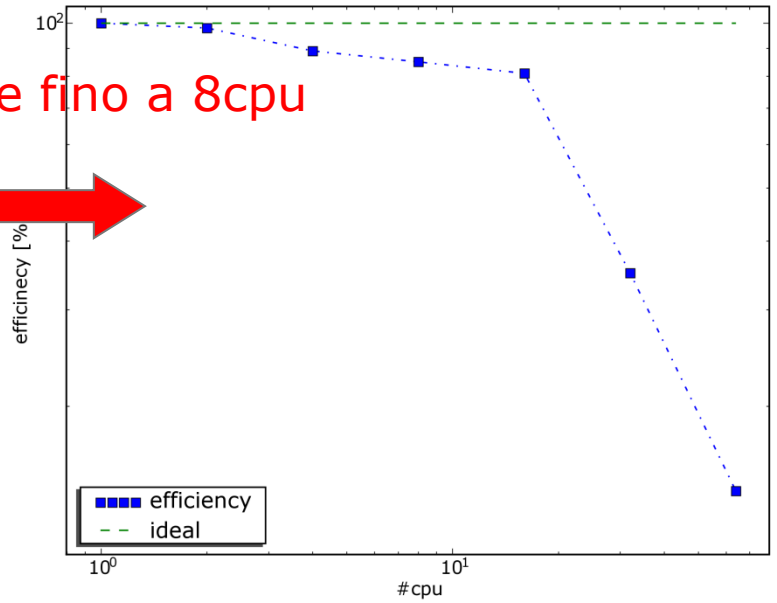


- Benchmark suite ufficiale comprende:
  - case small, (32000, 32000, 89856 celle + modelli fisici)
  - case medium, (155188, 242782 ,352800 celle + modelli fisici)
  - case large, (847764, 3618080, 9792512 celle + modelli fisici)

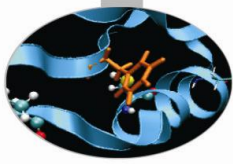
Small

scalabilità lineare fino a 8cpu

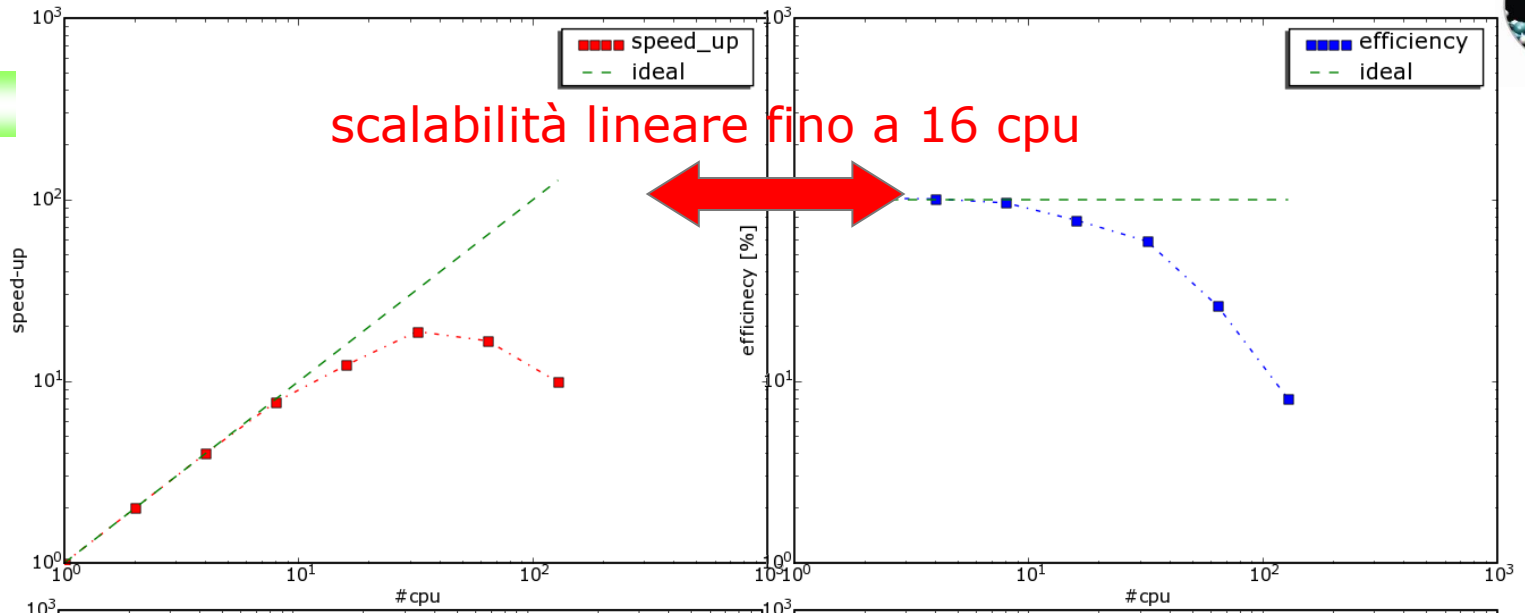




Test Fluent



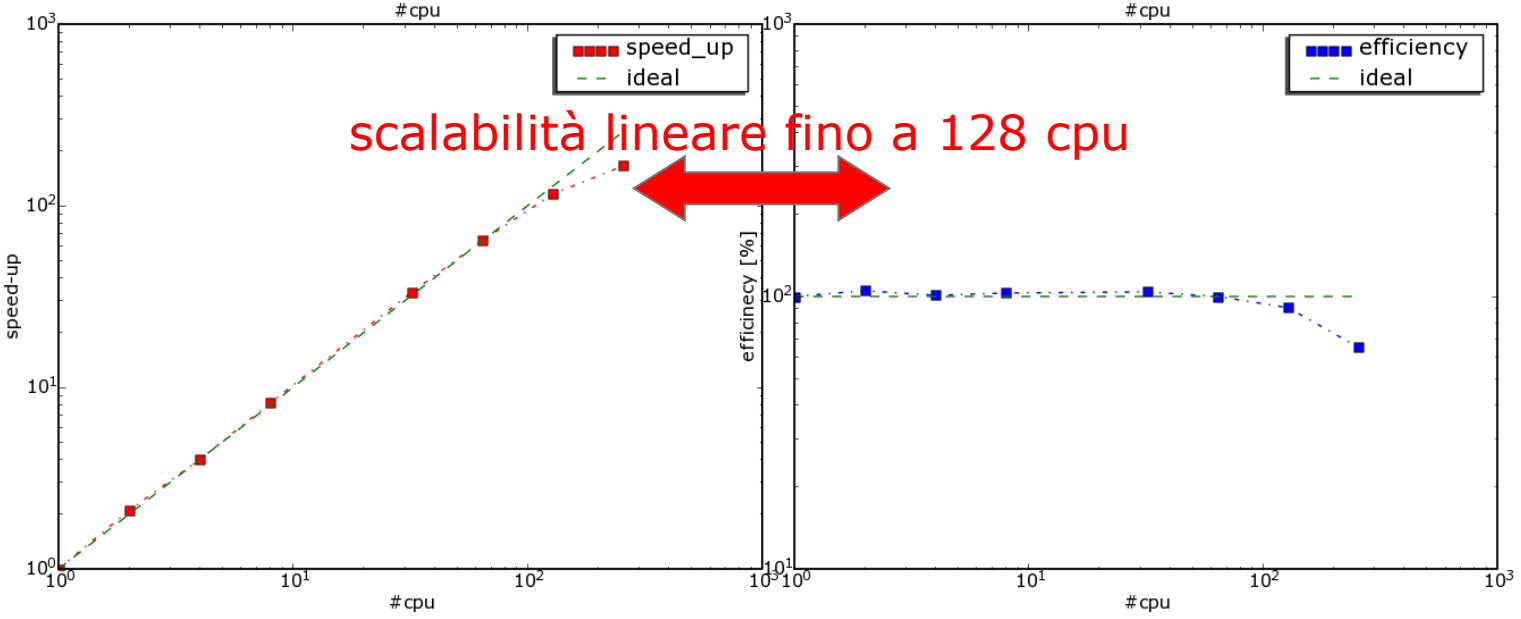
Medium



scalabilità lineare fino a 16 cpu

Large

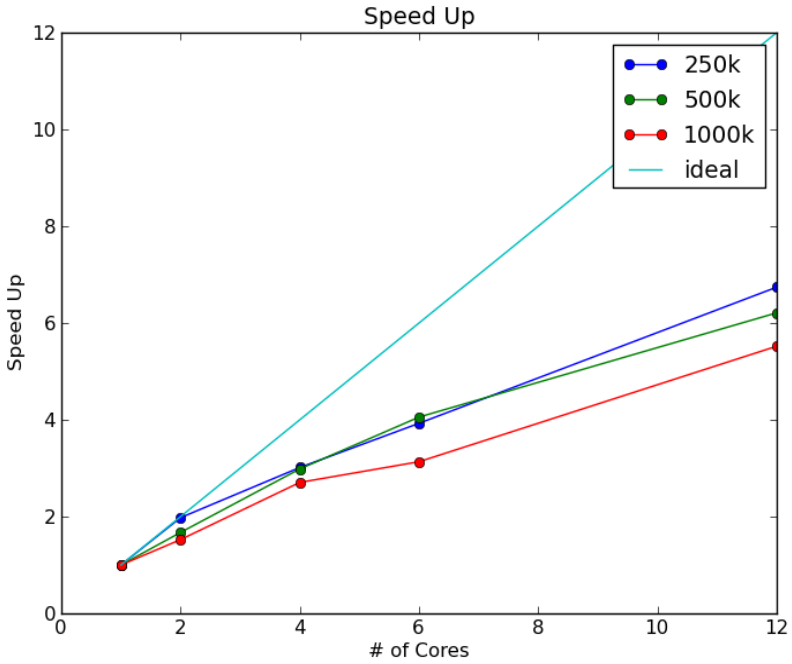
L3

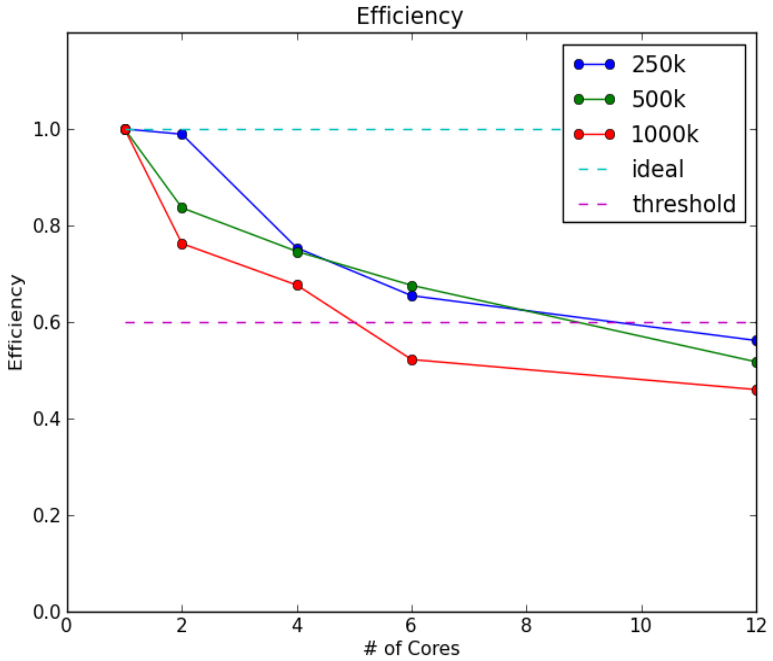


scalabilità lineare fino a 128 cpu

Test Abaqus

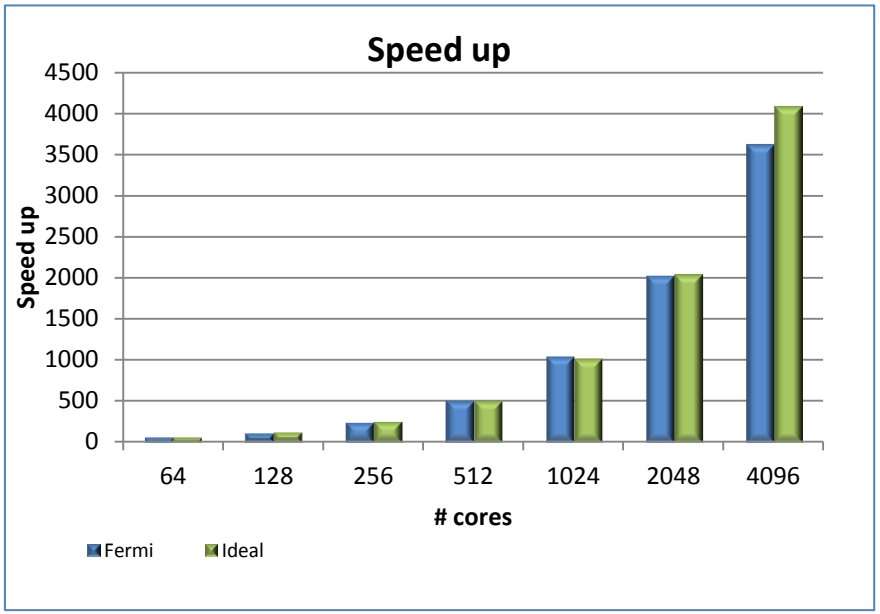
- Solutore fluido di abaqus su una griglia di dimensione crescente 250000 elementi, 500000 elementi, 1000000 elementi.

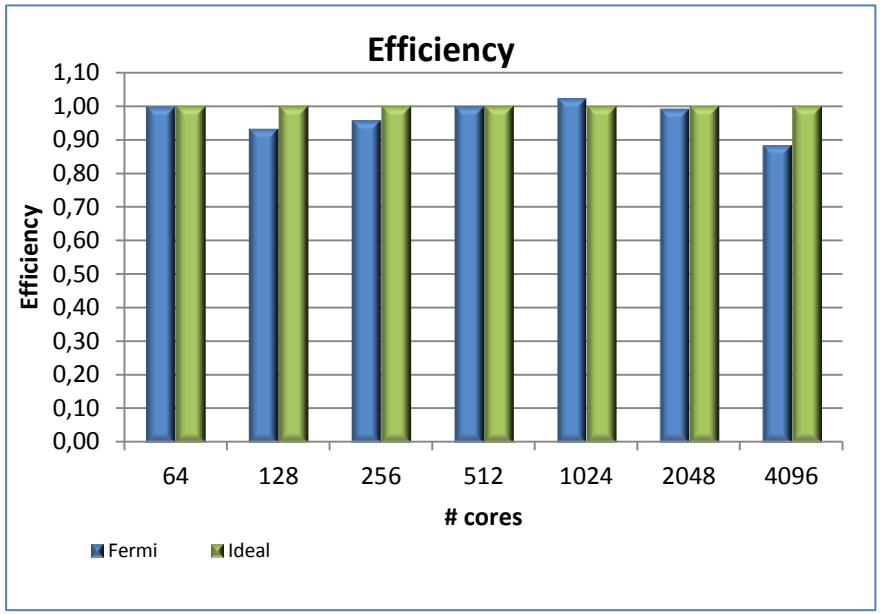




Test OpeFOAM

- Test lid-driven Cavity 3D, mesh 20.000.000 di elementi.





Test Gromacs

- Test effettuato con i benchmark ufficiali