

Introduzione al Calcolo Parallelo

Parallel computing e sistemi paralleli

La capacità di elaborazione parallela è ormai caratteristica di moltissime architetture:

- multicore
- multiprocessors
- acceleratori: GPU, FPGA
- sistemi distribuiti
- cluster
- sistemi di calcolo ad elevate prestazioni (High Performance Computing systems)

La disponibilità di hardware parallelo rende possibile lo sviluppo di applicazioni che sfruttino contemporaneamente e in modo ottimale le risorse di calcolo a disposizione.

Per sfruttare al meglio il parallelismo disponibile a livello HW gli sviluppatori possono utilizzare tecnologie ad hoc volte ad esprimere, gestire e massimizzare la capacità di esecuzione parallela.

Parallelismo vs. Concorrenza

Concorrenza: l'esecuzione di un programma concorrente dà luogo ad un insieme di attività contemporaneamente «in progress» (iniziate e non ancora terminate); l'esecuzione viene portata a avanti a turno in base alle decisioni dello scheduler. Può usare HW parallelo.

In generale: **numero dei processi > numero CPU**

Parallelismo: l'esecuzione di un programma parallelo determina l'esecuzione **contemporanea** di attività multiple. Richiede la disponibilità di HW parallelo.

In generale: **numero dei processi = numero CPU**

Parallel Computing: motivazioni

La principale motivazione allo sviluppo delle tecnologie hardware e software per il parallel computing è l'aumento di performance.

Ovvero:

- risolvere problemi di **complessità** elevata in tempi contenuti
- risolvere gli **stessi problemi in tempi più bassi**

Oggi le esigenze applicative sono sempre più complesse, ponendo questioni relative a come dominare tale complessità in tempi accettabili.

La complessità dipende:

- dal **problema** e dal suo procedimento risolutivo
- dalla **quantità dei dati** da elaborare.

Esempio: Big Data

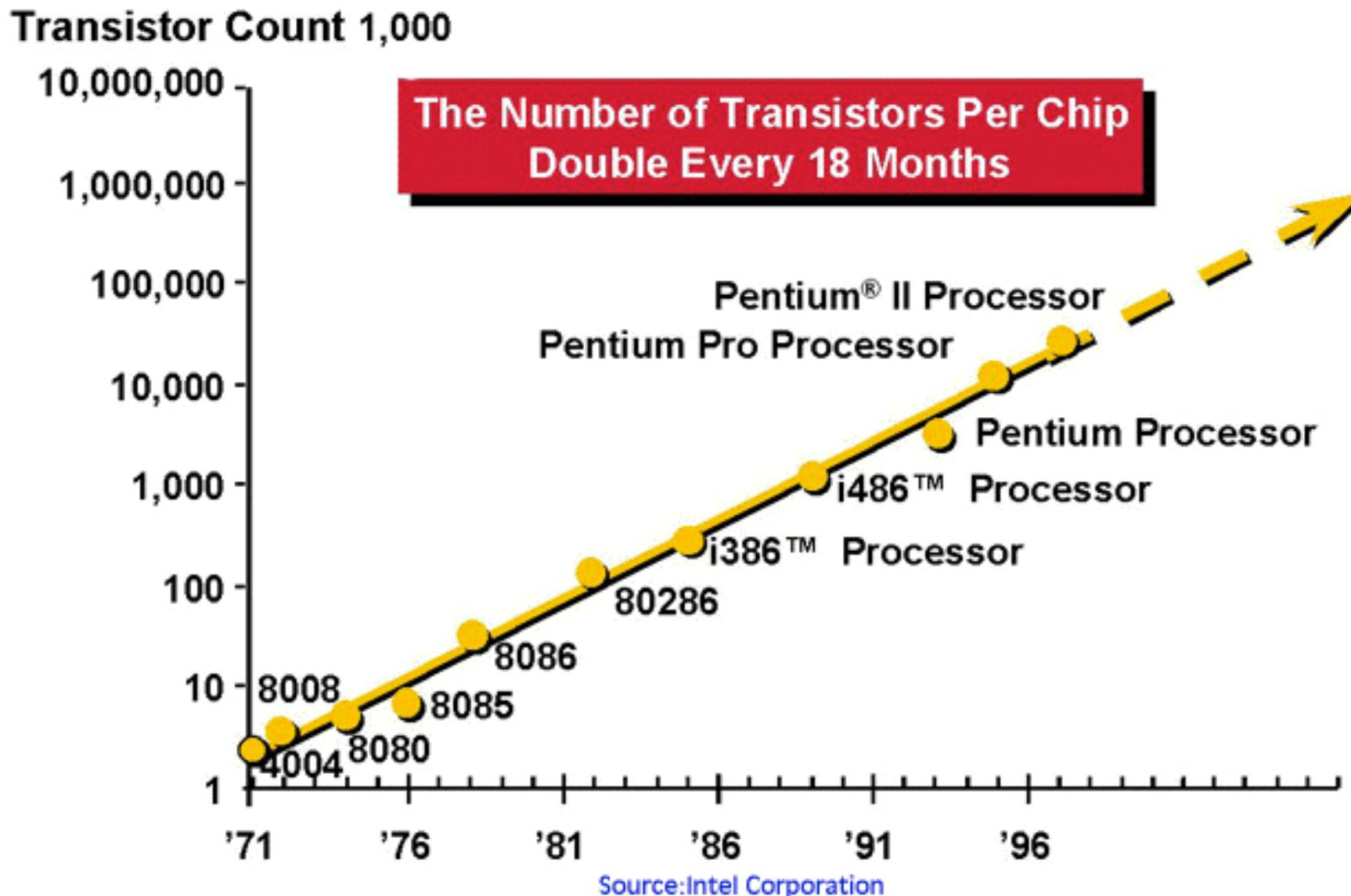
- disponibilità di enormi quantità di dati, inutili se non si ha la possibilità di analizzarli in tempi accettabili

Prestazioni dell'Hardware

Fino ai primi anni 2000 l'evoluzione dei sistemi di calcolo è stata «governata» dalla **Legge di Moore** (Gordon E. Moore, 1965, Electronics Magazine).

- Progresso tecnologico → complessità dei processori sempre maggiore, grazie alla crescente densità di transistor all'interno dei chip
- Per effetto della crescente densità dei transistori, a partire dagli anni 70 le performance dei processori crescono costantemente:
 - fino al 2000: raddoppio della densità ogni 18 mesi -> aumento della capacità di elaborazione del chip -> aumento della velocità di calcolo

Legge di Moore



Limiti fisici alle prestazioni di singoli chip di elaborazione

A partire dai primi anni 2000 ci si è trovati **sempre più prossimi ai limiti** fisici alla densità dei transistor sui chip.

A causa di effetti parassiti (effetto joule) le previsioni della legge di Moore sono state sempre di più disattese.

Non è stato più possibile aumentare la frequenza di clock → necessità di aumentare la capacità di calcolo a parità di frequenza.

Miglioramento delle prestazioni

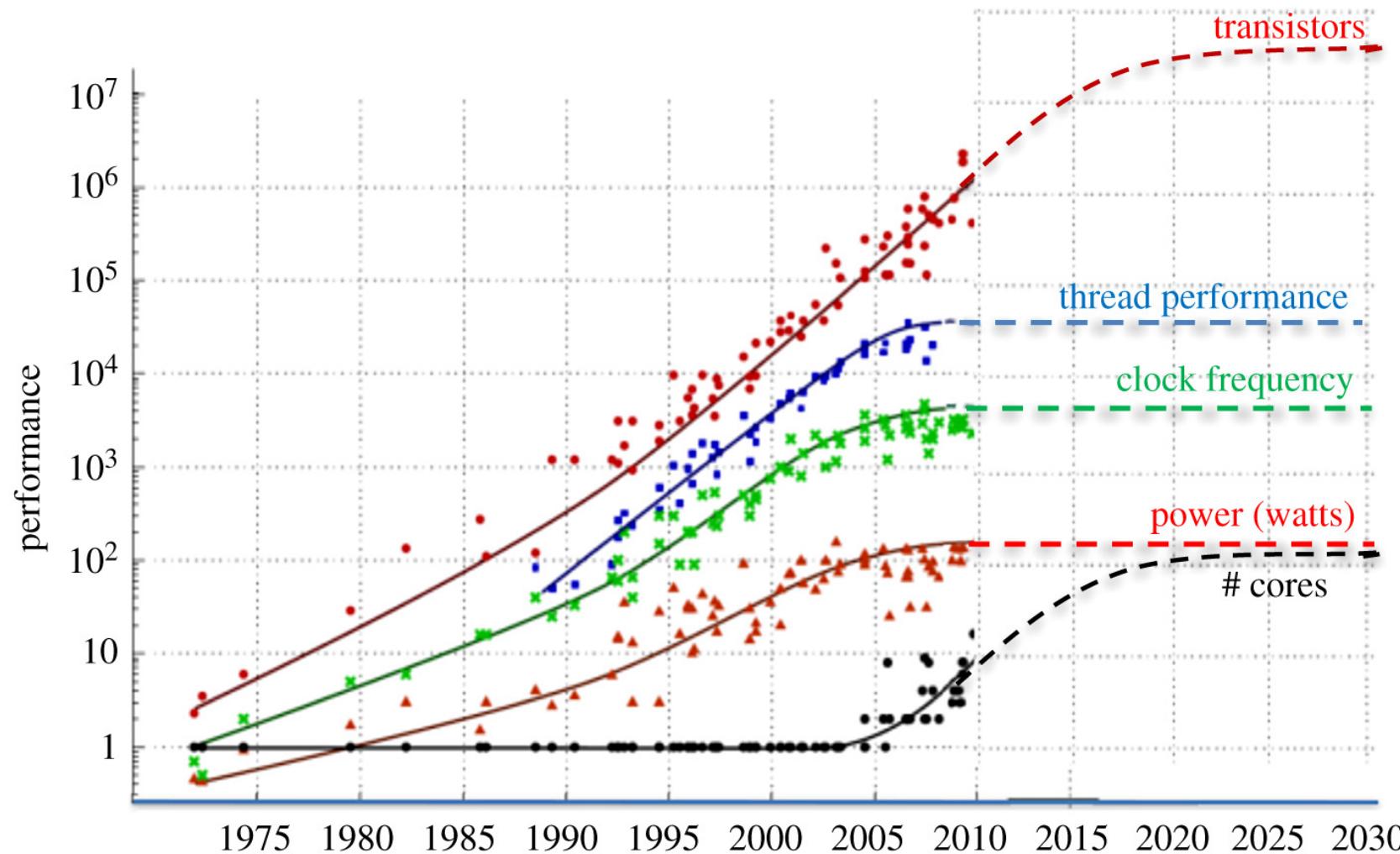
L'esigenza di prestazioni computazionali crescenti con frequenze di clock invariate ha trovato soluzione nell'introduzione di varie forme di parallelismo al livello hw.

Se l'hw è in grado di svolgere più operazioni per ciclo, la velocità di elaborazione dell'intero sistema aumenta.

Parallelismo come forma di accelerazione dell'hardware:

- più processori su singolo chip (multicore hardware, gpu, fpga)
- più processori su più chip -> multiprocessori, cluster, grid, ecc.

Il futuro



Architetture parallele

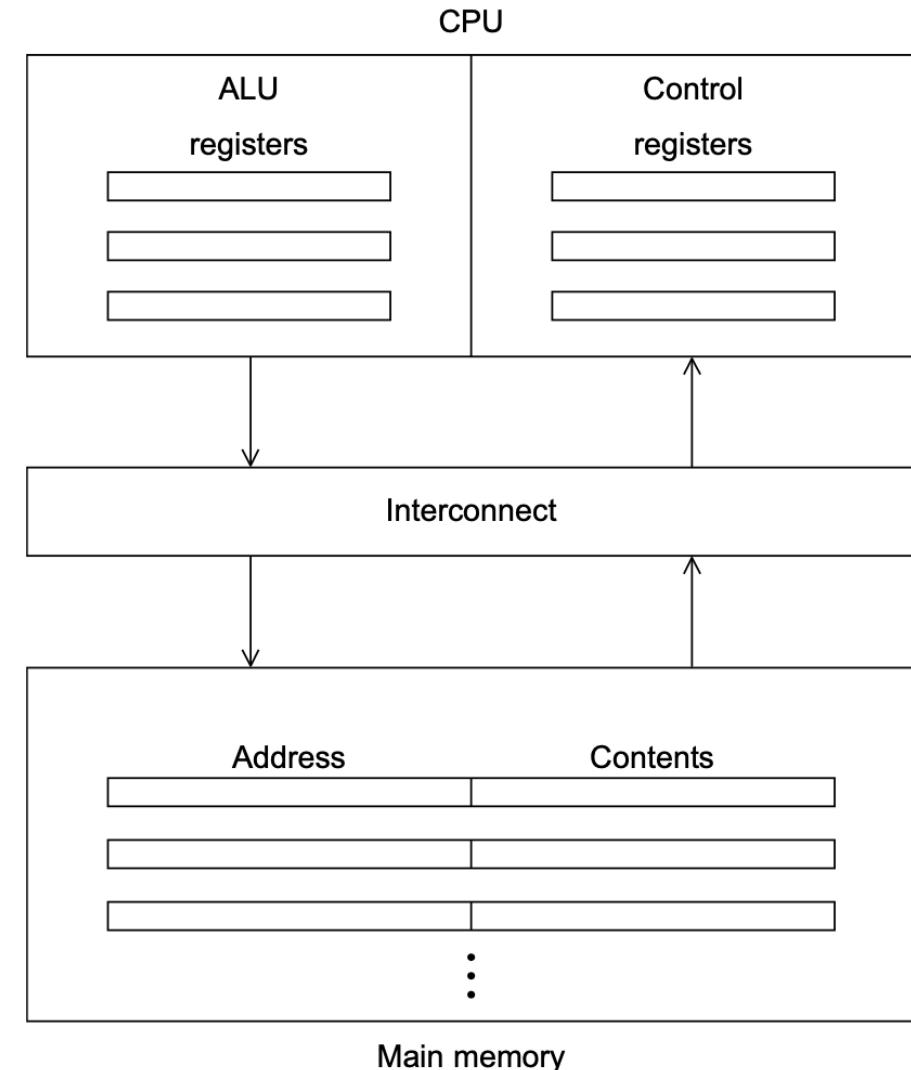
Il modello di Von Neumann

Il modello di Von Neumann descrive lo schema funzionale di un tradizionale sistema sequenziale.

L'unica CPU è collegata alla memoria centrale (che contiene dati e istruzioni) attraverso un mezzo di interconnessione (es: bus).

La separazione tra memoria e CPU costituisce una limitazione nella velocità di accesso a dati e istruzioni → «**Von Neumann bottleneck**»

Questa limitazione influisce sulla velocità di elaborazione del sistema.



Architetture parallele

Von Neumann Bottleneck: la velocità di fetching di istruzioni e dati dipende dalla velocità di trasmissione del Bus → limitazioni alla velocità di esecuzione.

Per mitigare questo problema, modello di Von Neumann è stato esteso con l'introduzione di:

- Memorie Cache
- Parallelismo di basso livello:
 - ILP
 - HW multithreading

Cache

E' una memoria **associativa**:

- **ad accesso veloce**: risiede sul chip del processore e si colloca a un livello intermedio, tra i registri e la memoria centrale
- **di capacità limitata**: la cache non può contenere tutte le istruzioni e i dati necessari al programma in esecuzione

Viene gestita con criteri basati sul principio di località (spaziale e/o temporale).

- **cache hit**: l'informazione richiesta è presente in cache
- **cache miss**: l'informazione richiesta non è presente in cache → necessità di load dalla memoria centrale

Se la gestione è tale da mantenere **hit-rate** (% hit su totale degli accessi) soddisfacenti, gli effetti del Von Neumann bottleneck possono essere contenuti.

Parallelismo Low Level: ILP

Instruction-level parallelism (ILP): L'esecuzione di ogni istruzione viene attuata attraverso una sequenza di fasi.

Ad esempio: somma di due float $C=A+B$

1. fetch operandi A e B
2. confronto esponenti ed eventuale shift
3. somma
4. normalizzazione del risultato
5. memorizzazione del risultato C

Ognuna delle fasi può essere affidata a un'**unità funzionale** indipendente che opera in parallelo alle altre:

- **pipelining:** tutte le unità funzionali sono collegate tra loro in una pipeline; fasi diverse di istruzioni diverse possono essere eseguite in parallelo;
- **multiple issue:** più istanze di ogni unità funzionale

Es: ILP pipeline

unità funzionali
collegate in pipeline

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
:	:	:	:	:	:	:	:
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Parallelismo Low Level

ILP non è sempre vantaggioso: se in un programma è presente una lunga serie di istruzioni tra loro dipendenti , il parallelismo a livello di istruzione non è utile.

Esempio: calcolo della serie di Fibonacci:

```
f[0] = f[1] = 1;  
for (i = 2; i <= n; i++)  
    f[i] = f[i-1] + f[i-2];
```

l'istruzione da calcolare ad ogni passo dipende dai risultati delle due precedenti.

In alternativa/aggiunta a ILP, i processori moderni offrono **parallelismo a livello di thread** (thread level parallelism) mediante **HW multithreading** .

Parallelismo Low Level: Hardware multi-threading

Hardware multithreading (TLP): permette a più thread (ad esempio, 2 thread) di condividere la stessa CPU (core), utilizzando una tecnica di sovrapposizione.

Ciò è reso possibile dalla duplicazione dei registri che mantengono lo stato di ogni thread (PC, IR, ecc) e da un meccanismo HW che implementa il context switch tra un thread ed un altro in modo molto efficiente.

Due approcci:

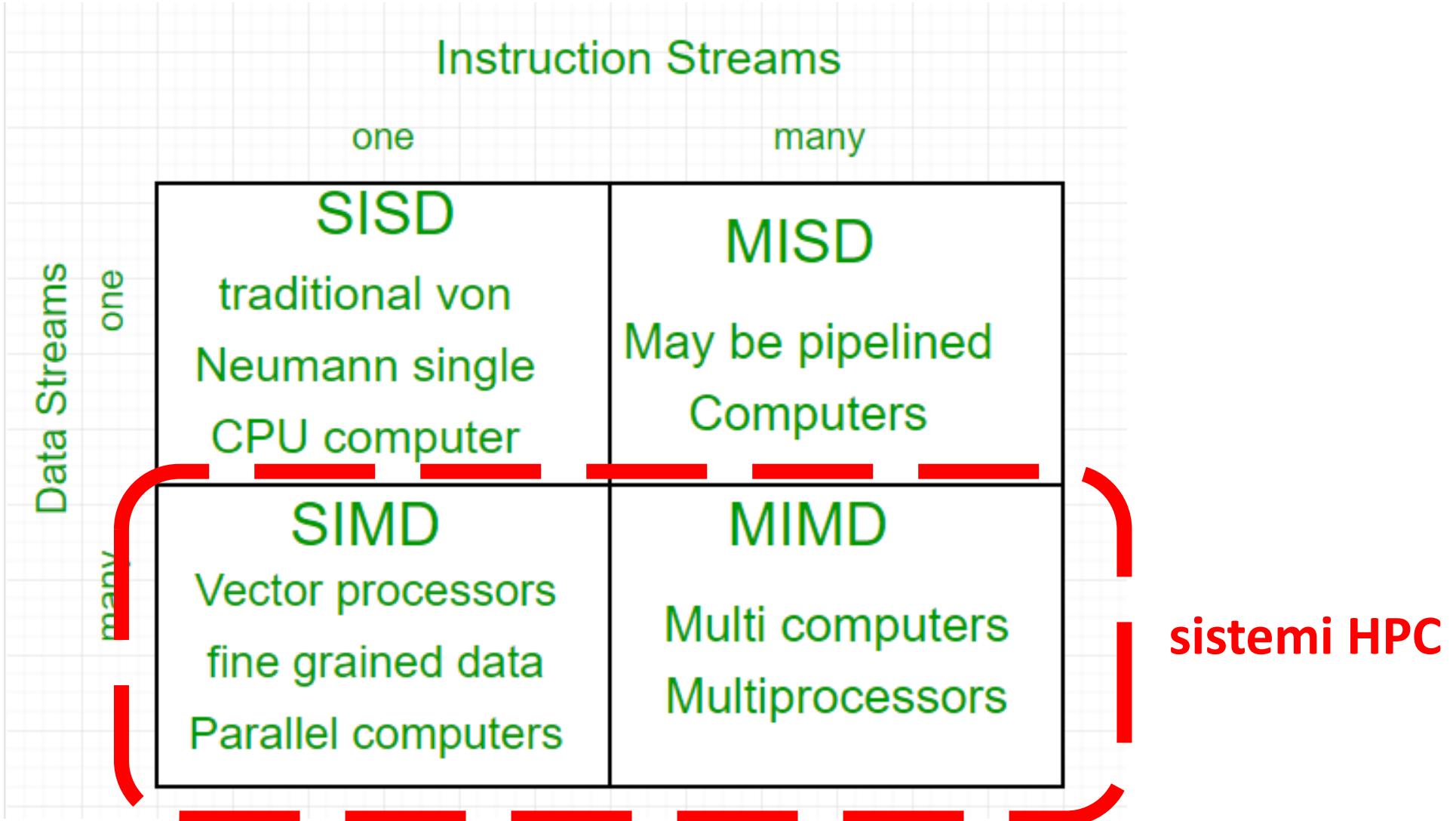
- **Multithreading a grana fine (*fine-grained*)**: viene eseguito un context switch dopo ogni istruzione.
 - **Vantaggio**: Attese (lunghe o brevi) di thread vengono «nascoste» allocando la CPU ad altri thread.
 - **Svantaggio**: velocità di esecuzione dei thread ridotta.
- **Multithreading a grana grossa (*coarse-grained*)**: il context switch avviene quando il thread corrente è in una situazione di attesa (ad esempio: in caso di cache «miss», deve attendere il caricamento dell'informazione dalla memoria centrale).
 - **Vantaggio**: meno context switch, quindi velocità media di esecuzione dei thread più alta.
 - **Svantaggio**: il context switch è più costoso (necessità di vuotare la pipeline). Throughput più basso.

Architetture ad elevate prestazioni per High performance Computing

ILP e Hardware multithreading hanno permesso un miglioramento delle prestazioni dei processori, tuttavia tali meccanismi sono trasparenti per gli sviluppatori dei programmi. → modello **Von Neumann Esteso**

Nei sistemi HPC, invece, il parallelismo disponibile per l'esecuzione dei programmi è visibile al programmatore, che deve progettare il software in modo da sfruttare al meglio tutte le risorse computazionali a disposizione. → architetture **non Von Neumann**

La classificazione di Flynn



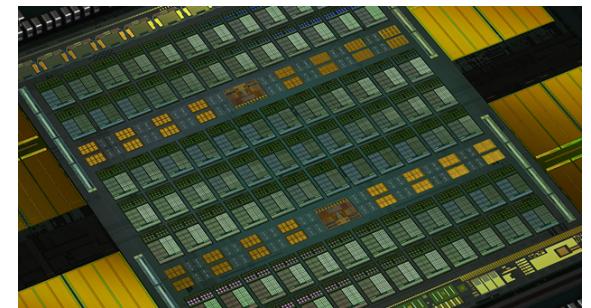
Sistemi SIMD

SINCRONICITA' nel FUNZIONAMENTO: Ad ogni istante le unità di calcolo presenti a livello HW eseguono la stessa istruzione su dati diversi.

- Più processori ed 1 sola control unit.

Esempio: GPU

Nelle prime GPU i processori grafici sono costituiti da molti core vincolati all'esecuzione della stessa istruzione; in quelle più recenti si ha un'architettura ibrida, in cui gruppi di core diversi possono eseguire flussi di istruzioni diverse.



Nvidia Tesla V100 Volta
architecture (5,120 CUDA cores)

Sistemi MIMD

Asincronicità delle attività nei diversi nodi: più processori, ognuno con la sua propria control unit.

Ogni CPU esegue una sequenza di istruzioni diversa dagli altri.

Necessità di interazione tra nodi

2 categorie di sistemi:

shared memory (UMA, NUMA multiprocessor/multicore)

distributed memory (cluster, grid, supercomputers)

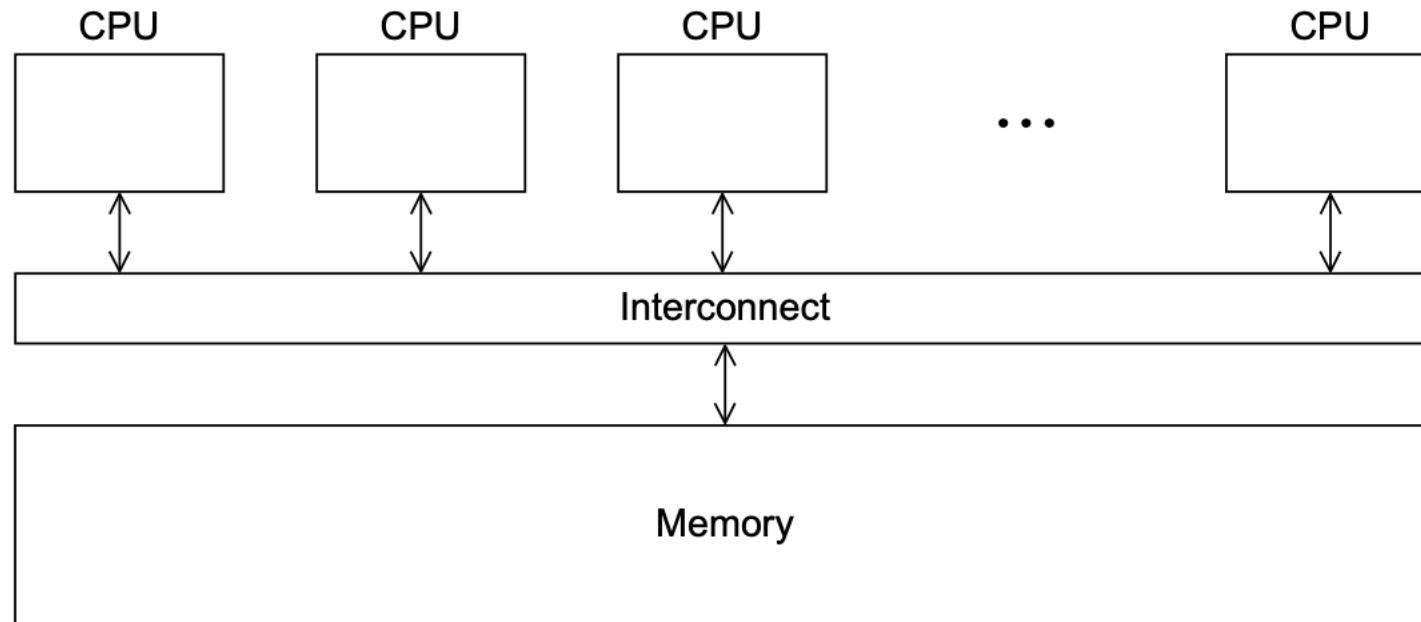
Necessario supporto a livello architetturale per consentire l'interazione:

- memoria & cache -> problema cache coherence
- reti di interconnessione ad alte prestazioni: bassa latenza e banda elevata



D'ora in avanti faremo riferimento implicitamente a sistemi **MIMD**.

Shared Memory system



Shared memory systems

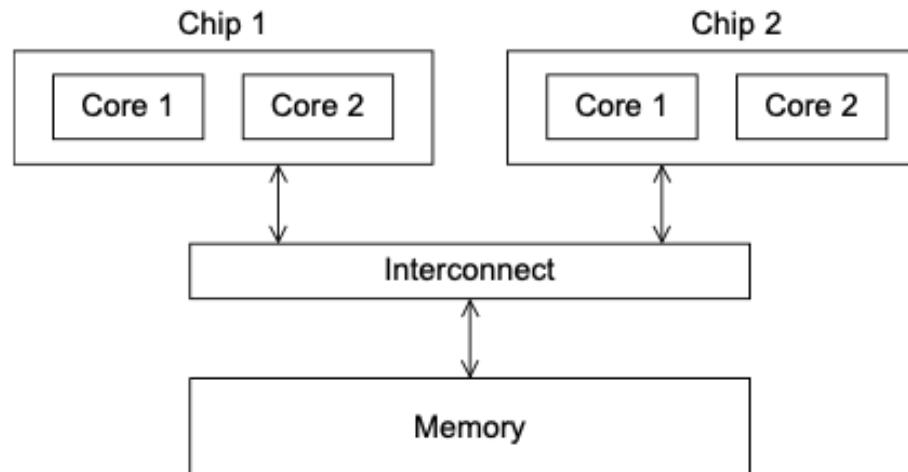
Fanno parte di questa categoria i sistemi **multicore** e **multiprocessor**.

Esempio: Sistema Multicore

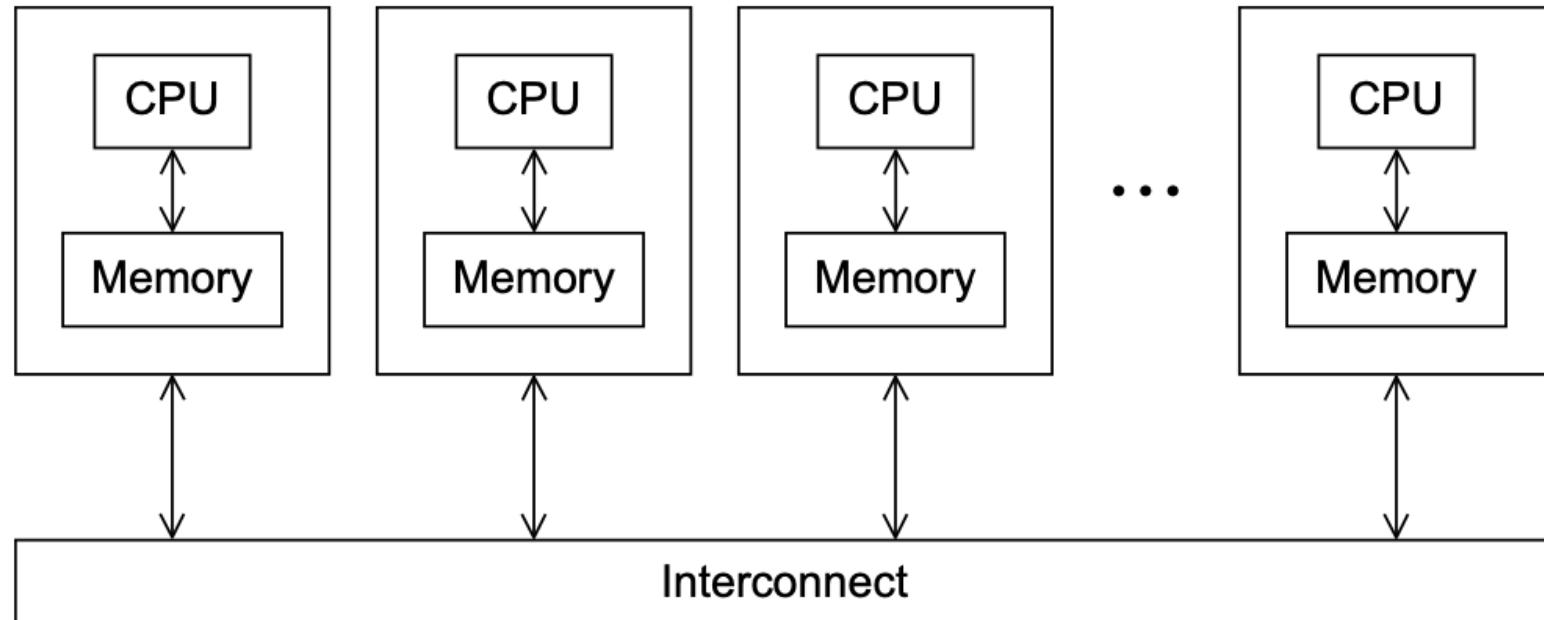


Intel Xeon Phi 7290
72 cores
x4 hyperthreading

Normalmente: più processori, ognuno con più core:



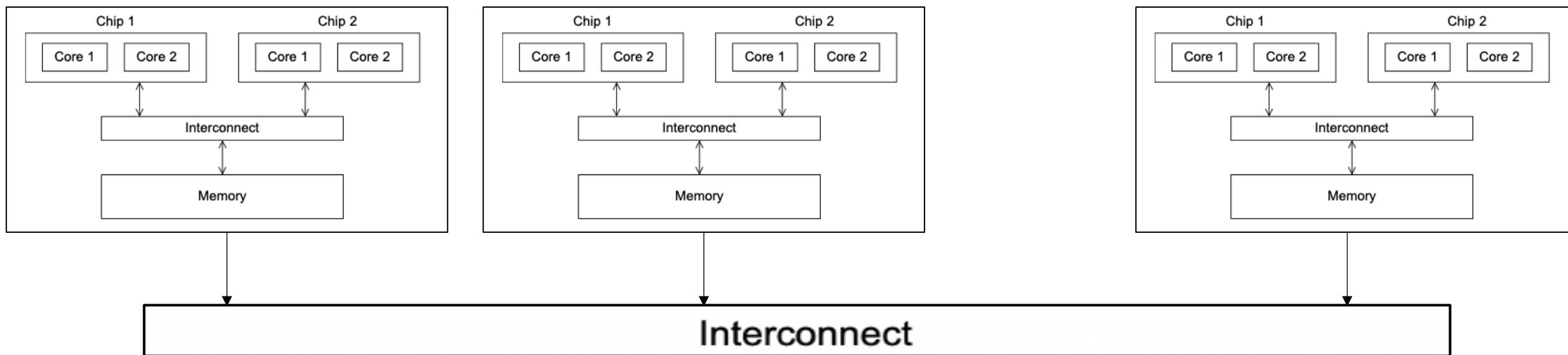
Distributed Memory system



- E' fondamentale che l'interconnessione tra nodi venga realizzata in modo tale da garantire:
 - latenza bassa
 - alta bandwith
- 👉 No Bus, ma reti ad alte prestazioni (topologie mesh, toroidali, ipercubi, ecc).

HPC systems: modelli ibridi

- La maggior parte dei sistemi HPC oggi combina il modello a memoria distribuita con il modello shared memory. Ad esempio:
 - cluster di nodi a memoria distribuita collegati da reti ad alte prestazioni
 - ogni nodo è un multiprocessore: insieme di processori multicore.



Struttura tipica di un cluster HPC

Esempio: **Marconi 100**, **Cineca**: cluster di 980 nodi, ognuno equipaggiato con :

Processors: 2x16 cores IBM POWER9 AC922 at 3.1 GHz → 32 cores/node

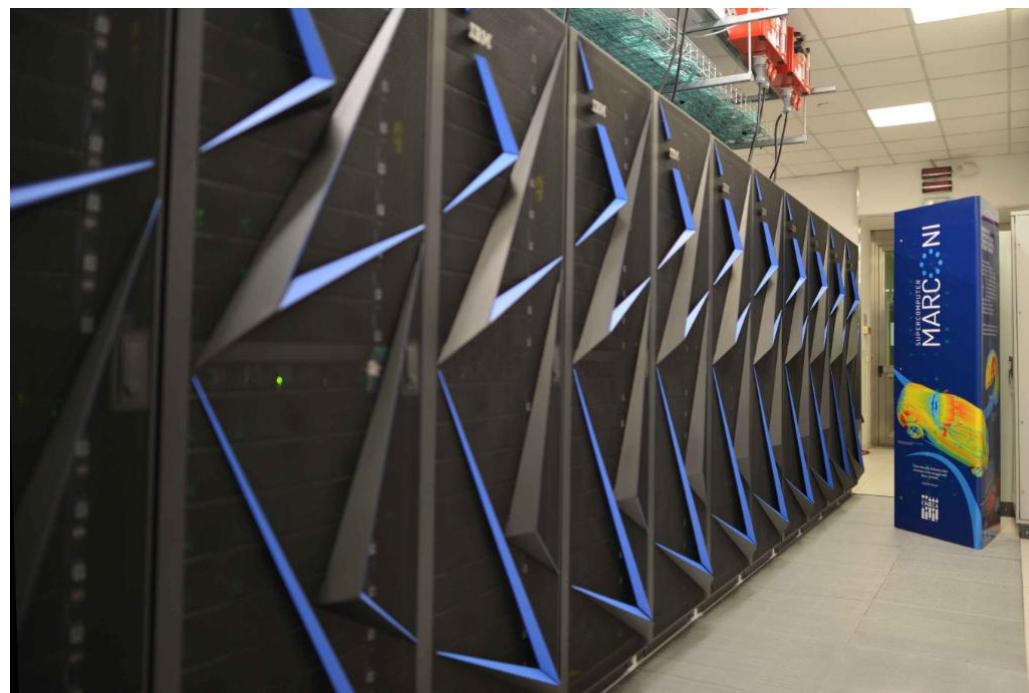
Accelerators: 4 x NVIDIA Volta V100 GPUs, NVlink 2.0, 16GB

RAM: 256 GB/node

Rete di interconnessione: Mellanox IB EDR DragonFly++

Totale: 980×32 cores → **31.360 unità di elaborazione**

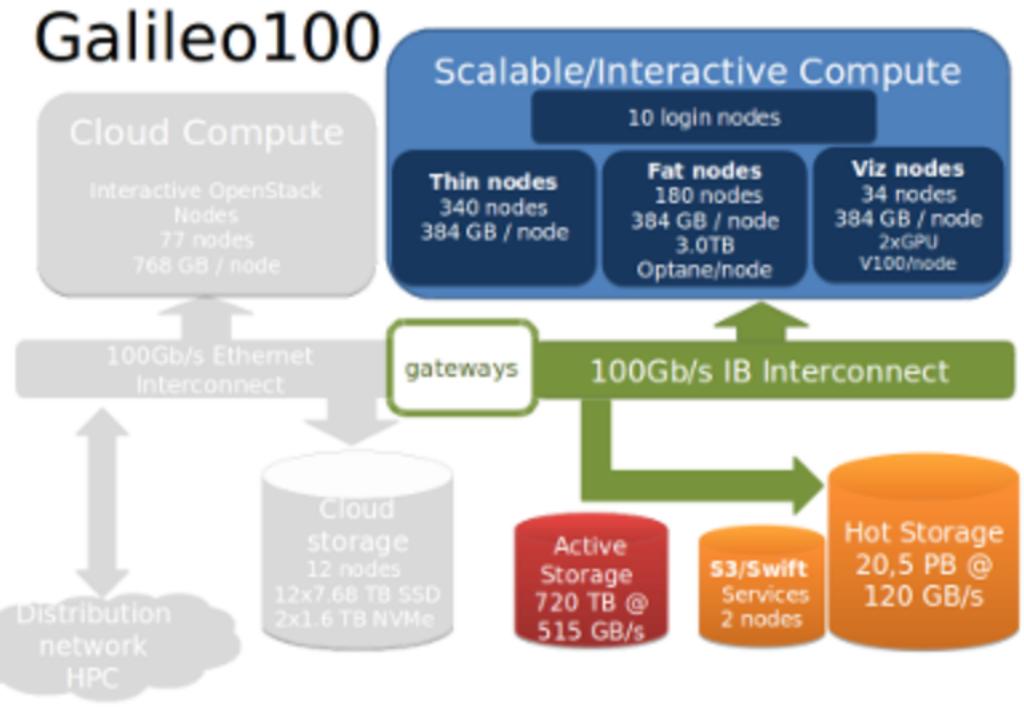
Ogni GPU ha 5.120 CUDA core → più di 20 Milioni di GPU cores



Struttura tipica di un cluster HPC

Esempio: **Galileo 100, Cineca:**

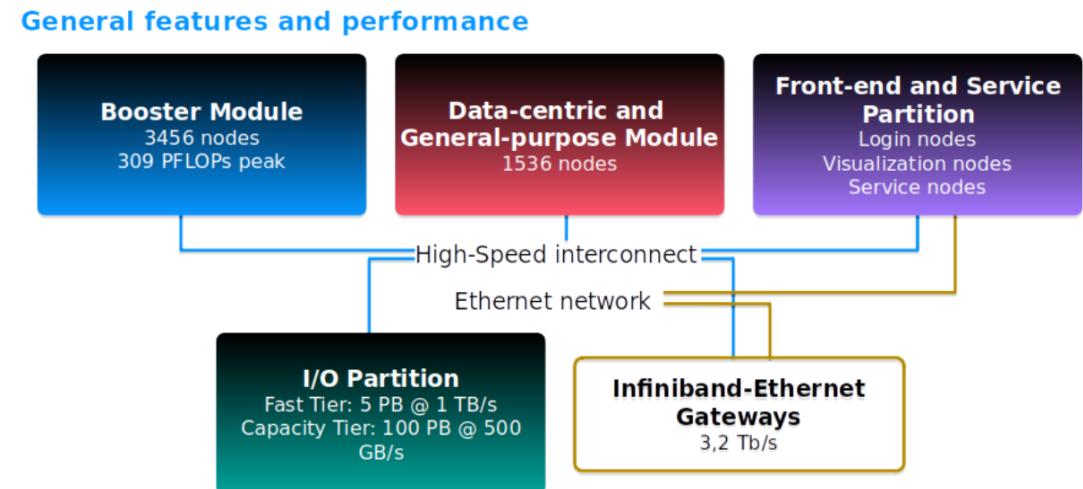
- 528 computing nodes each 2 x CPU Intel CascadeLake 8260, with 24 cores each, 2.4 GHz, 384GB RAM, subdivided in:
 - 348 standard nodes ("thin nodes") 480 GB SSD
 - 180 data processing nodes ("fat nodes") 2TB SSD, 3TB Intel Optane
 - 36 GPU nodes with 2x NVIDIA GPU V100 with 100Gbs Infiniband interconnection and 2TB SSD.
- 77 computing server OpenStack for cloud computing, 2x CPU 8260 Intel CascadeLake, 24 cores, 2.4 GHz, 768 GB RAM, with 100Gbs Ethernet interconnection.
- 20 PB of active storage accessible from both cloud and HPC nodes.
- 5 PB of fast storage for HPC system.
- 1 PB Ceph storage for Cloud (full NVMe/SSD)



Struttura tipica di un cluster HPC

Esempio: **Leonardo, Cineca (24 novembre 2022)**:

- 5092 computing nodes subdivided in:
 - 3456 booster nodes Intel Xeon 8358 each with :
 - 32 cores, 2.6 GHz → Cores: 110592 (32 cores/node),
 - 4XNvidia custom Ampere GPU 64GB HBM2,
 - RAM: (8x64) GB DDR4 3200 MHz
 - 1536 data-centric nodes Intel Saphire Rapids, 4.8 GHz, each with;
 - 2x56 cores → Cores: 172032 (112 cores/node)
 - RAM: (48x32) GB DDR5 4800 MHz
- 16 visualization nodes 2 x Icelake ICP06 32cores 2.4GHz, 3 NVIDIA Tesla V100, RAM: (16 x 32) GB DDR5 4800 MHz
- 106 PB (raw) Large capacity storage, 620 GB/s
- High Performance Storage 5.4 PB, 1.4 TB/s Based on 31 x DDN Exascaler ES400NVX2
- Login and Service nodes: 16 Login nodes are available. 16 service nodes for I/O and cluster management.



E' il 4^a sistema HPC più potente al mondo

Sviluppo di software parallelo

La necessità di adattare il codice alle risorse HPC per sfruttare efficacemente il parallelismo disponibile a livello HW, impone al programmatore di trasformare i propri programmi seriali in codice parallelo.

Come parallelizzare?

- **parallelizzazione automatica:** esistono compilatori «parallelizzanti» che traducono codice sequenziale in codice parallelo (es: traduzione parallela di clci). Normalmente le prestazioni ottenibili con questo approccio sono poco soddisfacenti (es. global sem pacheco pag.5)
- **parallelizzazione esplicita:** il programmatore esprime il parallelismo nel codice dei programmi che sviluppa, con la finalità di sfruttare nel modo migliore possibile le risorse computazionali HW: linguaggi e librerie per il calcolo parallelo. Un programma in esecuzione è un insieme di processi, ognuno in esecuzione su un nodo fisico distinto, che all'occorrenza possono interagire.

2 modelli di interazione:

- scambio di messaggi (es. MPI)
- memoria condivisa (es. OpenMP)

Software Parallelo

In sistemi HPC chi esegue un programma parallelo può contare sulla disponibilità esclusiva di un insieme di nodi di elaborazione da usare per l'esecuzione (differenza con OS multithreading/multitasking).

Task parallelism: In sistemi di questa categoria il parallelismo si ottiene distribuendo task diversi a processi diversi; ogni processo è assegnato a una CPU a sua completa disposizione.

Sviluppo di Software Parallelo: SPMD

Nello sviluppo di sw parallelo si utilizza quasi sempre il paradigma **SPMD** («Single Program Multiple Data»): ogni core esegue lo stesso programma; per differenziare il codice eseguito dai diversi nodi, si sfrutta **il branching condizionale**.

Ad esempio:

```
void run(int nodo)
{
    if (nodo==0) // nodo è il rank associato al processo
        master(); //codice eseguito dal task sul nodo 0
    else
        slave(nodo); // eseguito da ogni task su un nodo diverso da 0
}
```

👉 Il codice viene personalizzato in base all'indice (**rank**) del nodo sul quale il singolo processo esegue.

Sviluppo di Software Parallello

Parallelizzazione: In molti casi si parte da una versione seriale del programma che si desidera rendere più veloce nell'esecuzione.

👉 La versione seriale viene parallelizzata, ovvero trasformata in un insieme di processi paralleli, per essere eseguita in un'architettura HPC.

Serial Program → **Parallel Program**

In generale, la parallelizzazione può essere un compito complesso, ma in alcuni casi, il tipo di algoritmo eseguito rende molto semplice la transizione seriale -> parallelo (algoritmi «**embarassingly parallel**»)

Programmi embarrassingly parallel

Programmi embarrassingly parallel: la parallelizzazione è ottenuta dividendo il lavoro tra i processi in modo che ognuno esegua le stesse istruzioni su dati diversi in modo indipendente dagli altri. I processi ottenuti sono tra loro **indipendenti**.

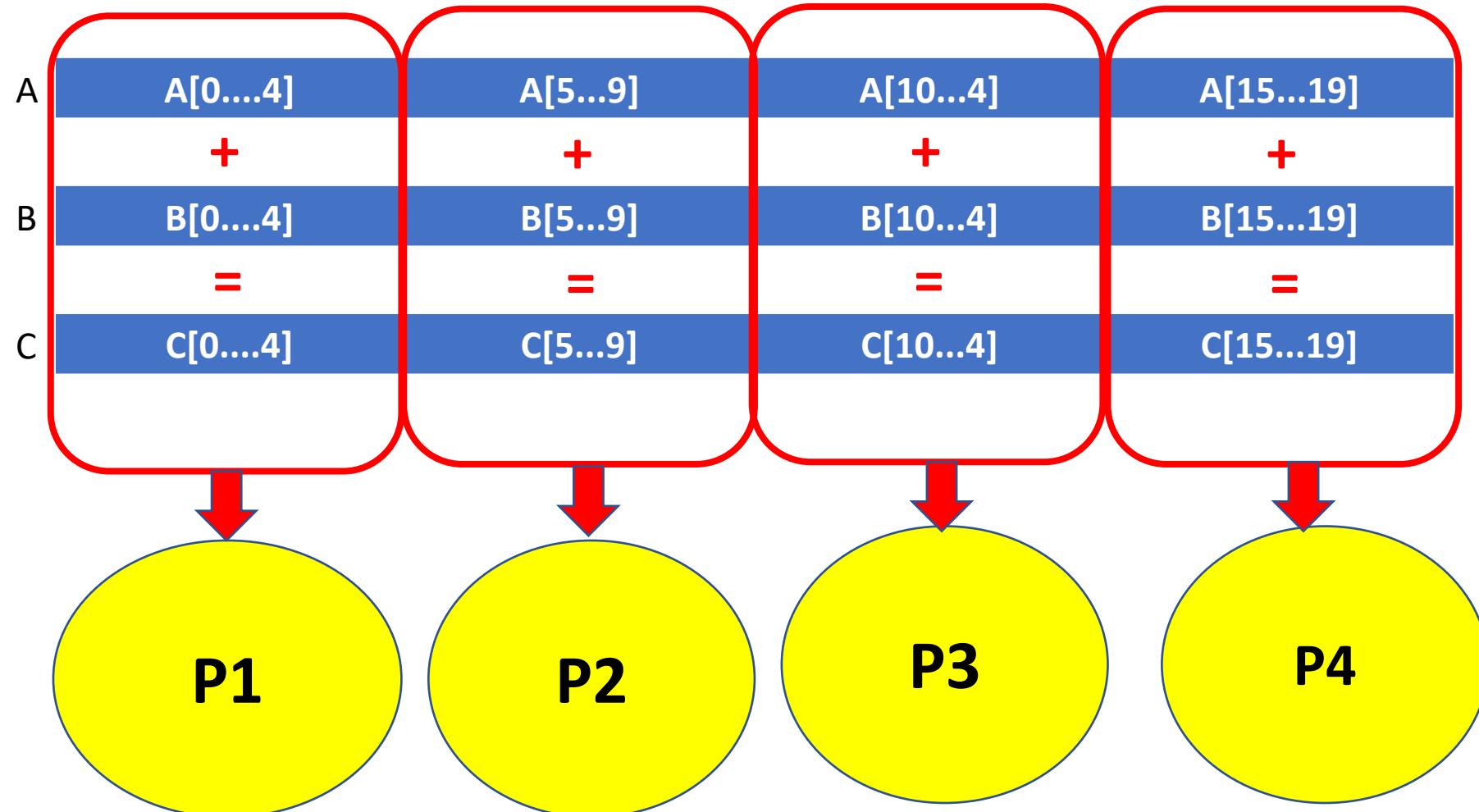
Esempio: parallelizzazione della somma di due vettori A e B

```
int A[20], B[20], C[20];  
...  
for(i=0; i<20; i++){  
    C[i] = A[i]+ B[i];
```

👉 ogni iterazione può essere eseguita in modo **indipendente** dalle altre

Programmi embarrassingly parallel

Quindi, se disponiamo di 4 nodi, allochiamo su ogni nodo un processo:



Partizionamento degli elementi dei vettori tra i 4 processi paralleli:

- P1 esegue le prime 5 iterazioni
- P2 esegue le iterazioni [5..9]
- P3 esegue le iterazioni [10..14]
- P4 esegue le iterazioni [15..19]

Idealmente: il tempo di esecuzione dovrebbe essere $\frac{1}{4}$ del tempo sequenziale.

Parallelizzazione

La maggior parte degli algoritmi non è embarrassingly parallel.

Ad esempio:

```
for (i = 0; i < n; i++)
    C[i] = A[i]*C[i-1];
```

le iterazioni non sono indipendenti → la parallelizzazione impone punti di sincronizzazione/comunicazione tra processi paralleli.

Sviluppo di un programma parallelo

In generale:

1. **dividere il lavoro** tra i processi, in modo da:
 - bilanciare il carico tra i nodi
 - minimizzare le interazioni
2. progettare la **sincronizzazione** tra processi paralleli
3. progettare la **comunicazione** tra processi

I problemi 2 e 3 sono correlati:

- **Nei sistemi shared memory:** i processi comunicano grazie alla sincronizzazione
- **Nei sistemi distributed memory:** i processi si sincronizzano per mezzo della comunicazione

Prestazioni del software parallelo

Come misurare le prestazioni di un sistema HPC?

Unità di misura delle prestazioni è il flops (floating point operations per second):

- yottaFLOPS: 10^{24} flops
- zettaFLOPS: 10^{21} flops
- exaFLOPS 10^{18} flops
- **petaFLOPS 10^{15}** (Marconi100 ~ 30 PFlops)
- teraFLOPS 10^{12}
- gigaFLOPS 10^9 (Intel I7)
- megaFLOPS 10^6
- kiloFLOPS 10^3

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
6	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
7	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371

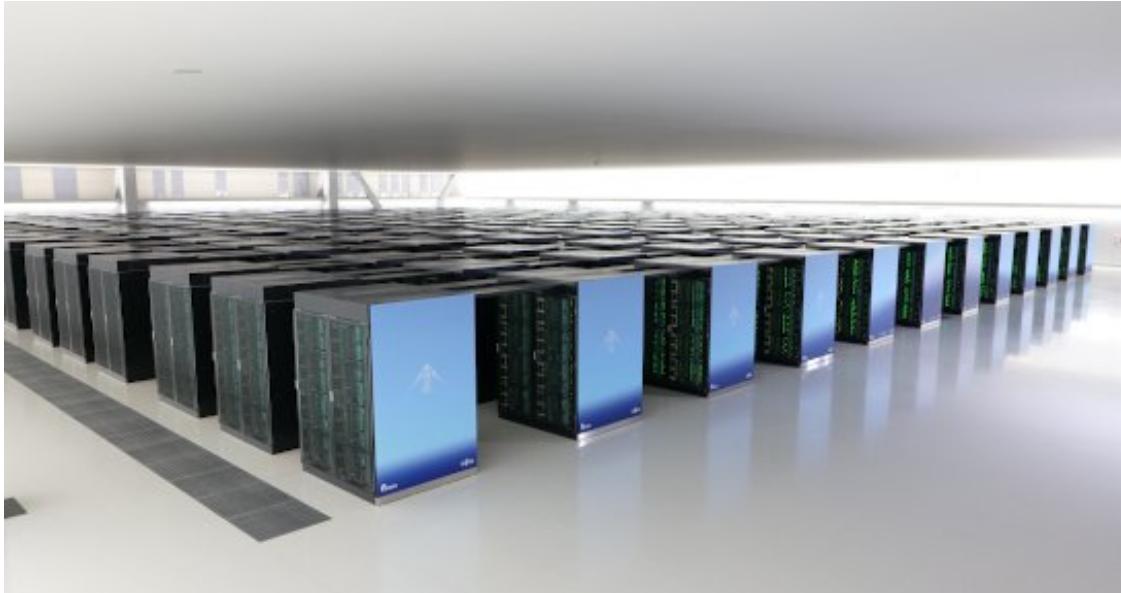
Top500.org

La lista Top500 riporta la classifica dei 500 sistemi di elaborazione più potenti del mondo.

[dati novembre 2022]

HPC systems: il trend

I sistemi HPC tendono verso “the exascale”: 10^{18} flops



Frontier, il sistema più potente al mondo (novembre 2022)
8.730.112 cores.
1,102 exaflops

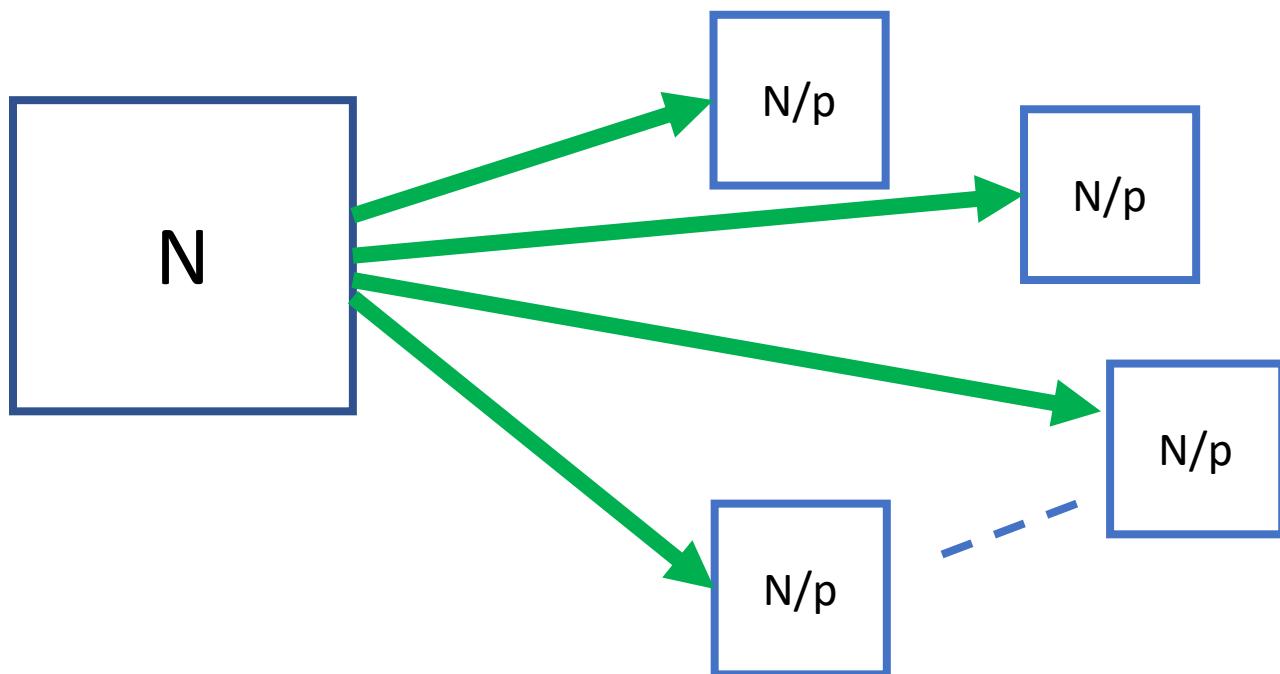
Obiettivi del calcolo parallelo

- Risolvere lo stesso problema in minor tempo
- Risolvere un problema più grande nello stesso tempo

Risoluzione di un problema in minor tempo

Sia N la dimensione del problema; sia p il numero di processori a disposizione:

→ decomposizione del problema in tanti sottoproblemi (possibilmente indipendenti) di dimensione N/p



Prestazioni del software parallelo

Per valutare il vantaggio derivante dall'esecuzione di programmi paralleli in sistemi HPC si utilizzano alcune metriche:

- **Speedup**
- **Efficienza**

Speedup

Def: $S = \frac{Tseq}{Tpar}$

- dove $Tseq$ è il tempo di esecuzione del programma nella sua versione sequenziale (su un solo nodo) e $Tpar$ il tempo di esecuzione del programma nella sua versione parallela.

👉 Lo speedup misura quanto è più veloce la versione parallela rispetto alla versione sequenziale, ovvero il guadagno derivante dalla parallelizzazione.

Speedup

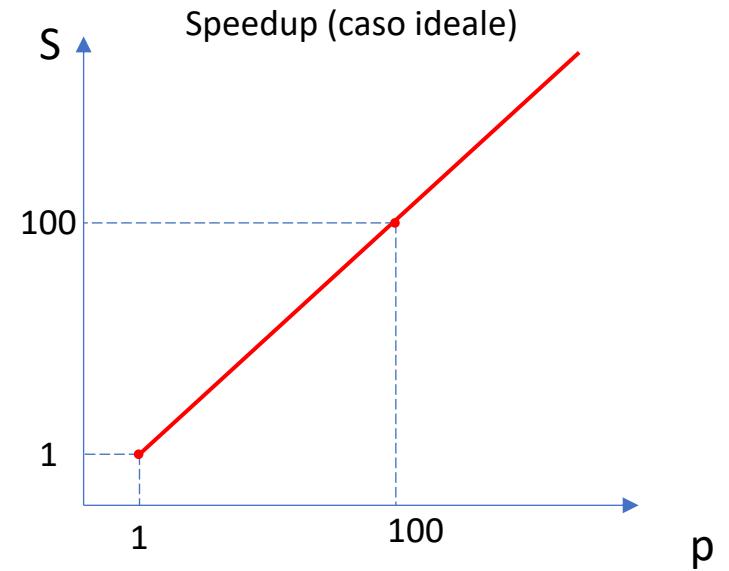
CASO IDEALE: Assumiamo di usare p processori per l'esecuzione parallela; nel caso ideale:

$$Tseq = p \cdot Tpar \rightarrow S = p$$

ovvero: nel caso ideale il tempo di esecuzione della versione parallela è $1/p$ del tempo nella versione sequenziale: $Tpar = \frac{Tseq}{p}$

In generale (casi non ideali):

$$Tpar = \frac{Tseq}{p} + T_{overhead} \rightarrow S < p$$



dovuto a creazione/allocazione processi, comunicazione/sincronizzazione, distribuzione non bilanciata del lavoro ecc.

Efficienza

Def: $E = \frac{S}{p}$

dove S è lo speedup e p è il numero di processori utilizzati.

Caso ideale: $S=p \rightarrow E=1$

Casi reali: $S < p \rightarrow E < 1$

Scalabilità: se un programma mantiene la **stessa efficienza** al variare del numero di processori utilizzati e/o al variare della quantità di dati da elaborare, si dice che è scalabile

Legge di Amdahl

Dato un programma, osserviamo che, in generale, non tutto il programma può essere parallelizzabile; il tempo ottenuto dall'esecuzione su p processori si può quindi esprimere come segue:

$$T_{par} = r \cdot T_{seq} + (1 - r) \cdot \frac{T_{seq}}{p} \quad \text{dove } r \in [0,1]$$

r esprime la frazione del tempo totale di esecuzione spesa nella parte non parallelizzabile del programma dato; $(r-1)$ è invece, la frazione di tempo impiegata nella parte parallelizzabile.

Legge di Amdahl

Pertanto possiamo esprimere lo speedup come segue:

$$S = \frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{r \cdot T_{seq} + (1-r) \cdot \frac{T_{seq}}{p}} = \frac{1}{r + \frac{(1-r)}{p}} \quad [\text{Legge di Amdahl}]$$

Osserviamo che la funzione $S(p)$ ha un comportamento asintotico:

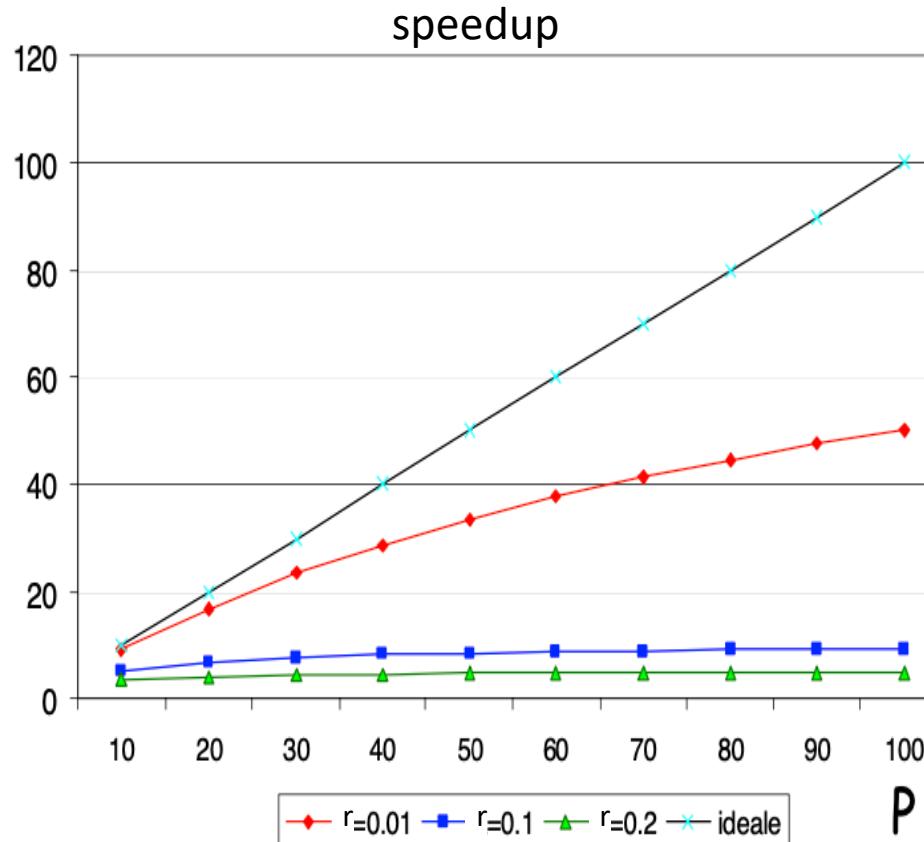
$$\lim_{p \rightarrow \infty} S = \frac{1}{r}$$

ovvero: se r è diverso da 0, lo Speedup non può crescere all'infinito ma può raggiungere, al massimo, il valore $1/r$.

Legge di Amdahl

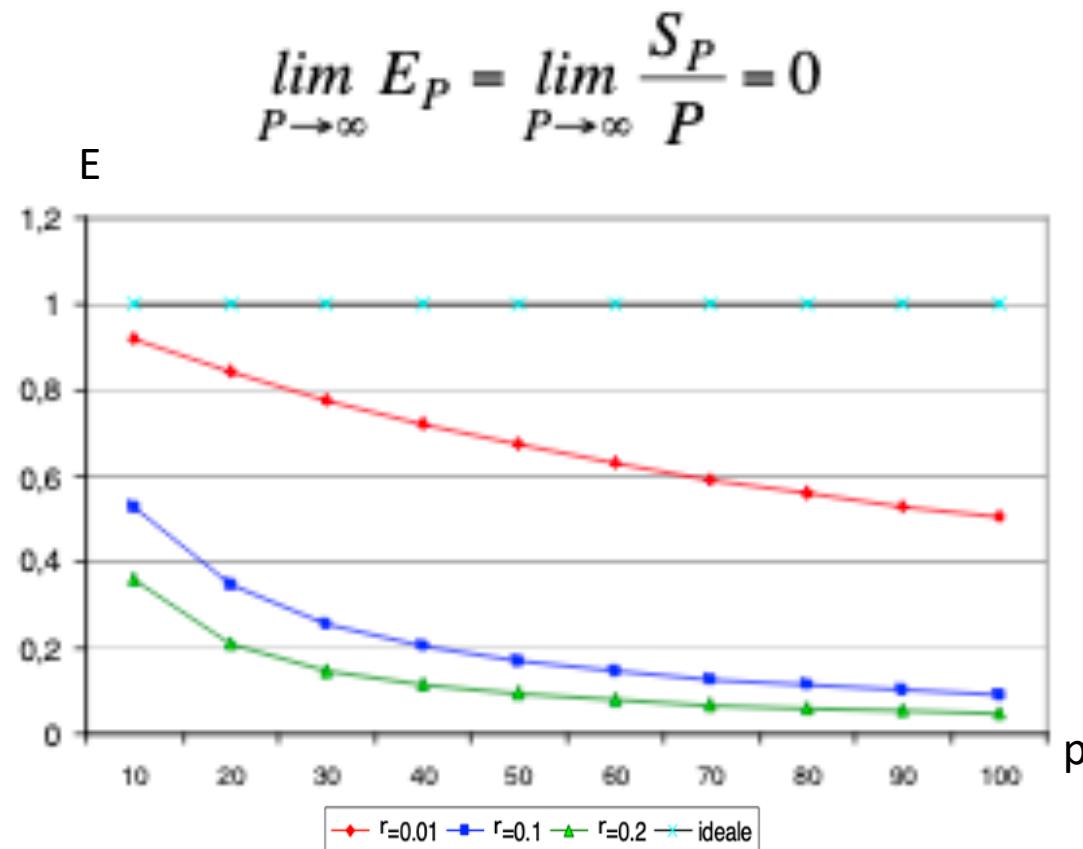
La legge di Amdahl descrive l'andamento dello Speedup al variare del numero di processori impiegati per la soluzione dello stesso problema.

$$\lim_{p \rightarrow \infty} S = \frac{1}{r}$$



Legge di Amdahl e efficienza

Con riferimento all'efficienza, la legge di Amdahl mostra che all'aumentare del numero dei nodi, l'efficienza cala:

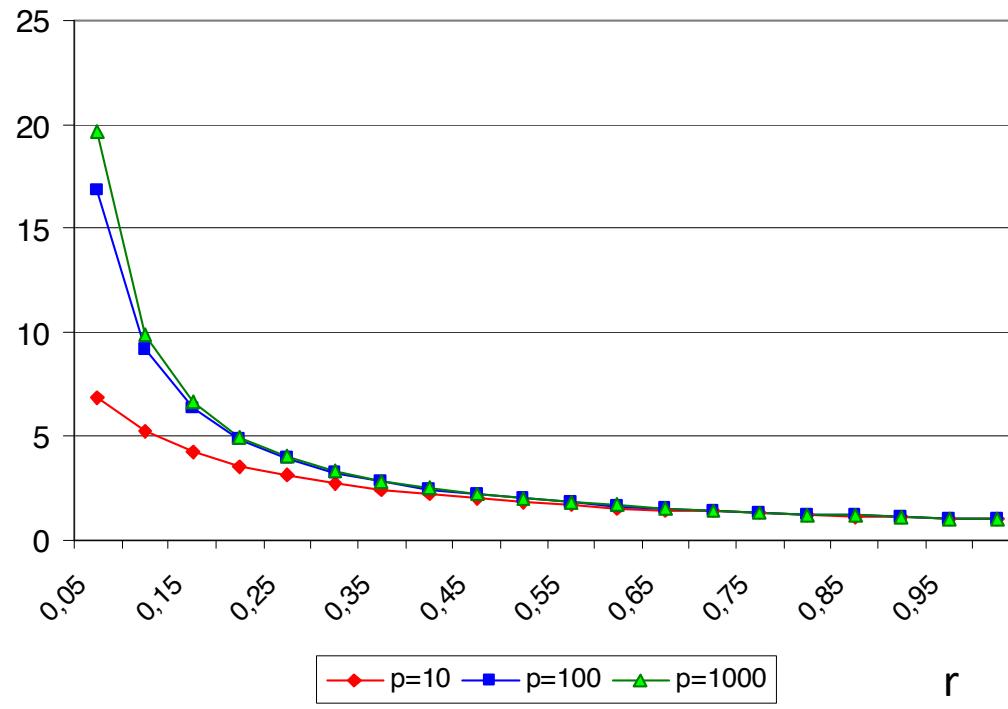


Scalabilità strong: misura la variazione di efficienza/speedup al variare del numero di processori.

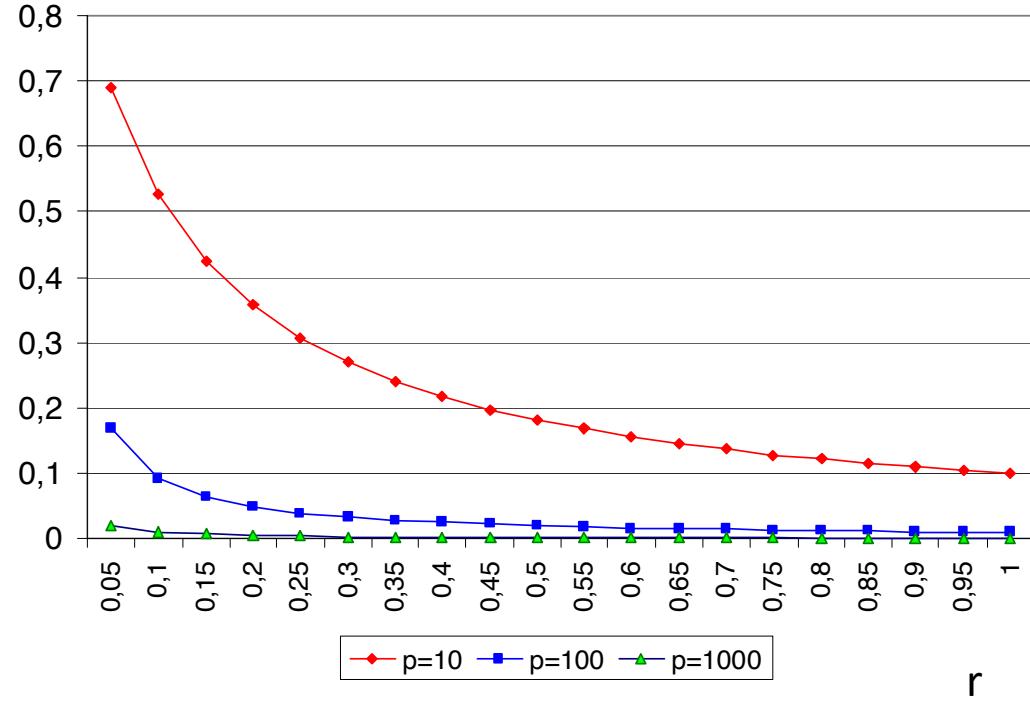
Nel caso **ideale**, al crescere di p , l'efficienza si mantiene costante.

Impatto della componente non parallelizzabile r

$$\lim_{r \rightarrow \infty} S = 0$$



Speed Up



Efficienza

Impatto della dimensione del problema sulle prestazioni

Le prestazioni di un programma parallelo dipendono, oltre che dal numero di risorse (CPU) a disposizione, anche dalla **dimensione** del problema da risolvere (ovvero, dalla quantità di dati da elaborare).

$$\text{TEMPO DI ESECUZIONE} = T(\text{dim, proc})$$

Es. sistema di equazioni lineari -> dimensione della matrice dei coefficienti e termini noti

In alcuni casi, la dimensione del problema è fissata, in altri è variabile.

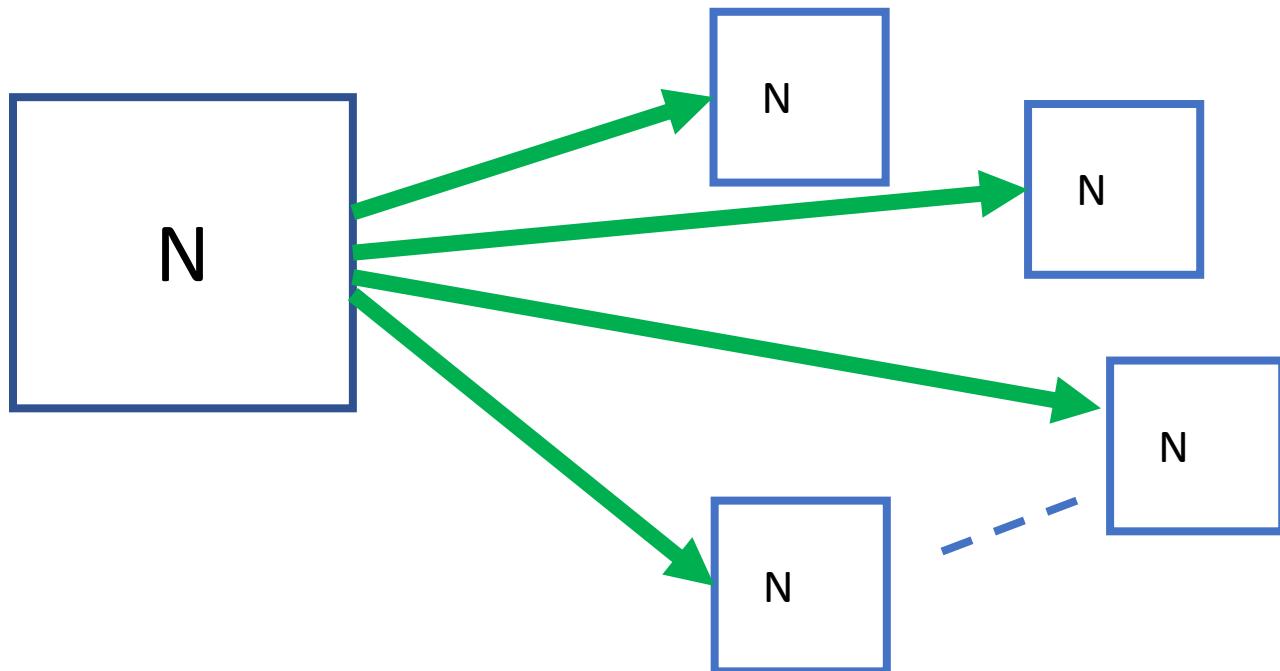
Impatto della dimensione del problema sulle prestazioni

Talvolta:

- invece di valutare quanto si può guadagnare nella soluzione di uno stesso problema di dimensione fissata aumentando il numero di processori (**scalabilità strong**),
- è più rilevante valutare se possiamo risolvere lo stesso problema con dimensioni maggiori nello stesso tempo (**scalabilità weak**).

Risoluzione di un problema più grande nello stesso tempo

Dato un problema di dimensione N risolto su un nodo, aumento linearmente la dimensione del problema al crescere del numero dei processori, mantenendo pari a N il carico di ogni processore con p processori \rightarrow dimensione $N.p$



Scalabilità strong & weak

Il concetto di scalabilità, in generale, esprime per un programma parallelo la capacità di mantenere invariata l'efficienza al variare del numero dei nodi e/o della dimensione del problema.

Scalabilità strong: la dimensione del problema è costante, si valuta l'efficienza al crescere del numero dei nodi. (V. legge Amdahl). In questo caso:

- il lavoro totale da eseguire è costante
- il lavoro del singolo nodo diminuisce al crescere del numero dei nodi

Se la dimensione del problema aumenta, per mantenere tempi paragonabili aumentiamo il numero dei nodi.

Scalabilità weak: mantenendo costante il carico di lavoro per singolo nodo, si valuta l'efficienza al crescere del numero dei nodi con lo stesso fattore.

👉 la dimensione del problema aumenta in proporzione al numero p di nodi utilizzati.

Scalabilità weak: speedup scalato

Per valutare la scalabilità weak, si considerano 2 metriche: efficienza scalata e speedup scalato.

Efficienza scalata

Def: $E_s(p, N) = \frac{T(N, 1)}{T(p.N, p)}$ **efficienza scalata** del programma

Dove:

- $T(N, 1)$ è il tempo necessario a risolvere un problema di dimensione N con 1 processore
- $T(p.N, p)$ è il tempo necessario a risolvere un problema di dimensione N.p con p processori

→ nel caso ideale il tempo è lo stesso: $E_s(p, N) = 1$

Speedup scalato

Def: $S_s(p, N) = E_s p$ **speedup scalato** del programma → nel caso ideale vale p

Scalabilità weak

Dato un problema di dimensione N , risolto in parallelo con p processori.



$$\text{Tempo totale } T_p = T_{\text{ser}} + T_{\text{par}}$$

r: frazione del tempo di esecuzione non parallelizzabile

(1-r): frazione parallelizzabile

$$\text{assumo } T_p = 1 \rightarrow T_{\text{ser}} = r$$

$T_{\text{par}} = (1-r)$ [workload assegnato ad ogni processore]

Lo stesso problema risolto con 1 solo processore:



In aggiunta alla parte seriale, l'unico processore ha un carico di lavoro che è p volte il carico di lavoro del singolo processore nel caso parallelo:

(assunzione: r non dipende da p)

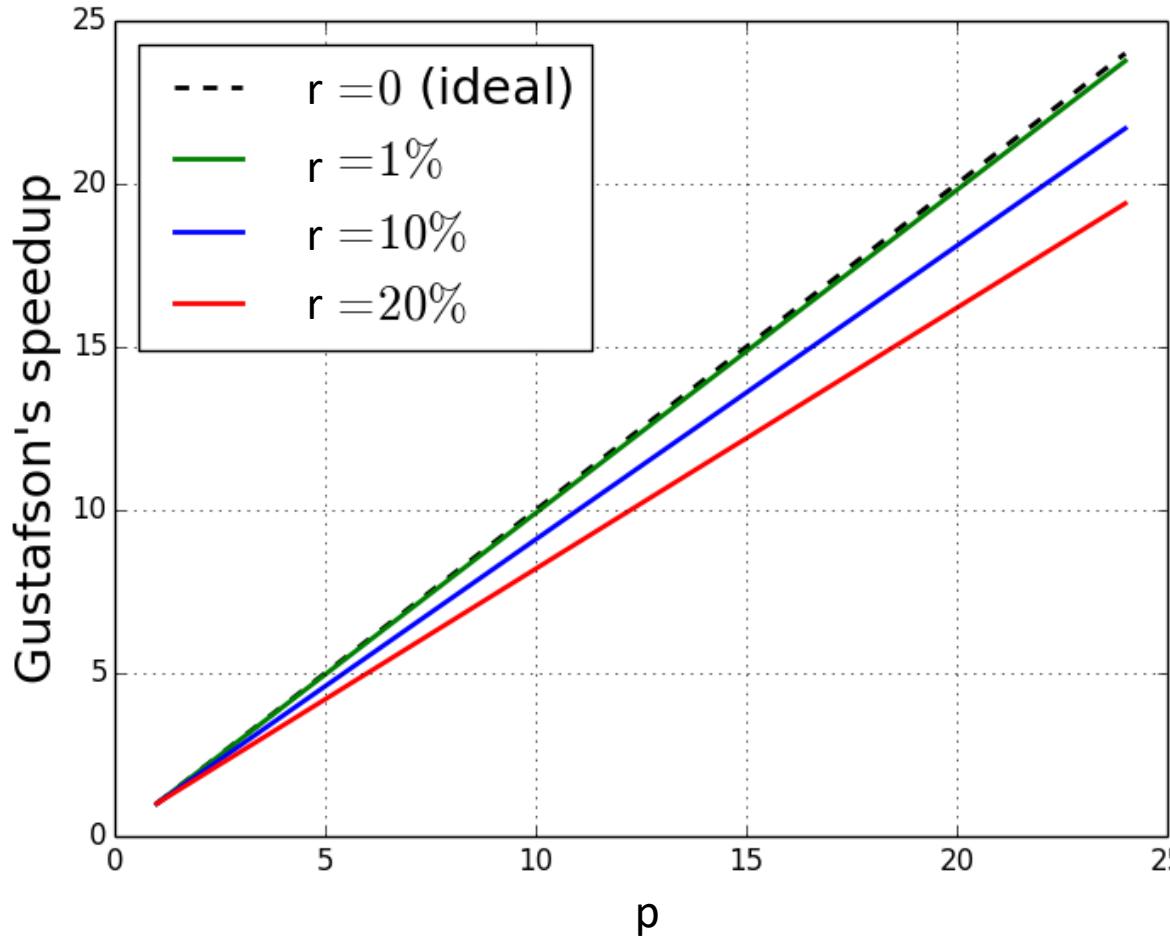
$$T_1 = r + (1-r) \cdot p$$

Legge di Gustafson

$$S = \frac{T_1}{T_p} \rightarrow$$

$$S = r + (1 - r) \cdot p \quad [\text{Legge di Gustafson}]$$

Implicazioni della legge di Gustafson



$$S = r + (1 - r) \cdot p$$

Assegnando ad ogni processore un workload costante ($1-r$), lo speedup cresce linearmente con il numero dei processori.