Introduzione al calcolo parallelo per il metodo degli elementi finiti

Simone Parisotto

Università di Verona
Dipartimento di Informatica

24 Aprile 2013

Scienza Computazionale

Scienza Computazionale

Un qualsiasi ramo delle scienze matematiche, fisiche, chimiche e naturali che utilizza le potenze di calcolo dei più recenti calcolatori al fine di risolvere problemi inaccessibili per i tempi e le modalità di calcolo umani.

L'uso di calcolatori per studiare sistemi fisici permette di gestire fenomeni:

- molto grandi;
- molto piccoli;
- molto complessi;
- molto pericolosi o costosi.

Scienza Computazionale

Scienza Computazionale

Un qualsiasi ramo delle scienze matematiche, fisiche, chimiche e naturali che utilizza le potenze di calcolo dei più recenti calcolatori al fine di risolvere problemi inaccessibili per i tempi e le modalità di calcolo umani.

L'uso di calcolatori per studiare sistemi fisici permette di gestire fenomeni:

- molto grandi;
- molto piccoli;
- molto complessi;
- molto pericolosi o costosi.

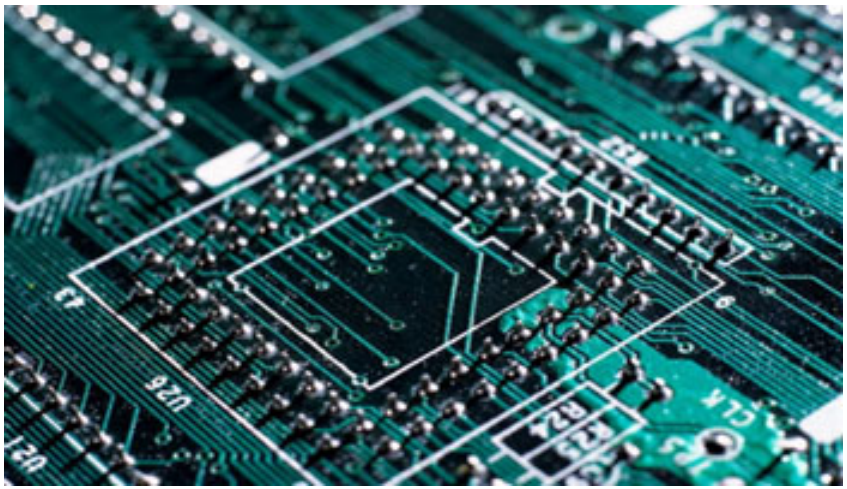
Scienza Computazionale

Scienza Computazionale

Un qualsiasi ramo delle scienze matematiche, fisiche, chimiche e naturali che utilizza le potenze di calcolo dei più recenti calcolatori al fine di risolvere problemi inaccessibili per i tempi e le modalità di calcolo umani.

L'uso di calcolatori per studiare sistemi fisici permette di gestire fenomeni:

- molto grandi;
- molto piccoli;
- molto complessi;
- molto pericolosi o costosi.



Scienza Computazionale

Scienza Computazionale

Un qualsiasi ramo delle scienze matematiche, fisiche, chimiche e naturali che utilizza le potenze di calcolo dei più recenti calcolatori al fine di risolvere problemi inaccessibili per i tempi e le modalità di calcolo umani.

L'uso di calcolatori per studiare sistemi fisici permette di gestire fenomeni:

- molto grandi;
- molto piccoli;
- molto complessi;
- molto pericolosi o costosi.

Scienza Computazionale

Scienza Computazionale

Un qualsiasi ramo delle scienze matematiche, fisiche, chimiche e naturali che utilizza le potenze di calcolo dei più recenti calcolatori al fine di risolvere problemi inaccessibili per i tempi e le modalità di calcolo umani.

L'uso di calcolatori per studiare sistemi fisici permette di gestire fenomeni:

- molto grandi;
- molto piccoli;
- molto complessi;
- molto pericolosi o costosi.

- Sono richiesti calcolatori sempre più potenti che consentano di considerare problemi in precedenza non affrontabili;
- la conoscenza scientifica avanza attraverso l'uso di tecnologie di calcolo e di comunicazione ad alte prestazioni.

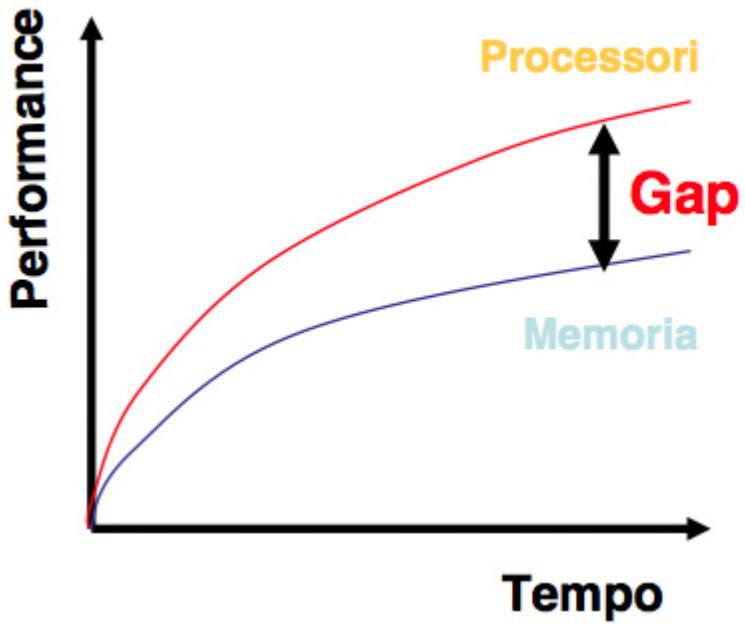
Supercomputer

Insieme dei calcolatori più potenti[1] disponibili in un determinato arco di tempo.

[1] di esecuzione, capacità di memoria e precisione macchina

Possibili miglioramenti:

- aumentare la potenza dei calcolatori, ma ci sono dei limiti:
  - velocità della luce;
  - dissipazione del calore;
- aumentare il parallellismo.



L'idea di parallellismo non è nuova:

- Charles Babbage (1791- 1871)
- John von Neumann (1903-1957)

Nel 1842 **Menabrea**, descrivendo la Macchina analitica di Charles Babbage (25.000 parti), evidenziava la possibilità che, ad esempio, parti diverse della stessa macchina calcolassero simultaneamente gli elementi indipendenti di una tabella.
Negli anni '40, **Von Neumann** propone metodi per risolvere equazioni differenziali su una griglia discreta. Tutti i punti della griglia venivano aggiornati in parallelo determinardo come i vicini si influenzavano gli uni gli altri.

SUPERCOMPUTER: New statistical machines with the mental power of 100 skilled mathematicians in solving even highly complex algebraic problems

New York World, Marzo 1920

Terminologia

Dimensione Computazionale

Numero di operazioni globalmente necessarie per la soluzione del problema (funzione della dimensione delle strutture dati coinvolte: n, n^2, $n \log n$. . .).

- 1 flop: floating point operation, operazione aritmetica su decimali in virgola mobile;
- flop/s: unità di misura della velocità dei calcolatori.

NB: 1 anno $\approx 3 \times 10^7$ secondi, calcolatori più potenti: 10^{12} flop/s.

TOP500: Lista dei 500 più potenti supercomputer non distribuiti.

GREEN500: Lista dei supercomputer nella TOP500 in termini di efficienza energetica.

Legge (empirica) di Moore

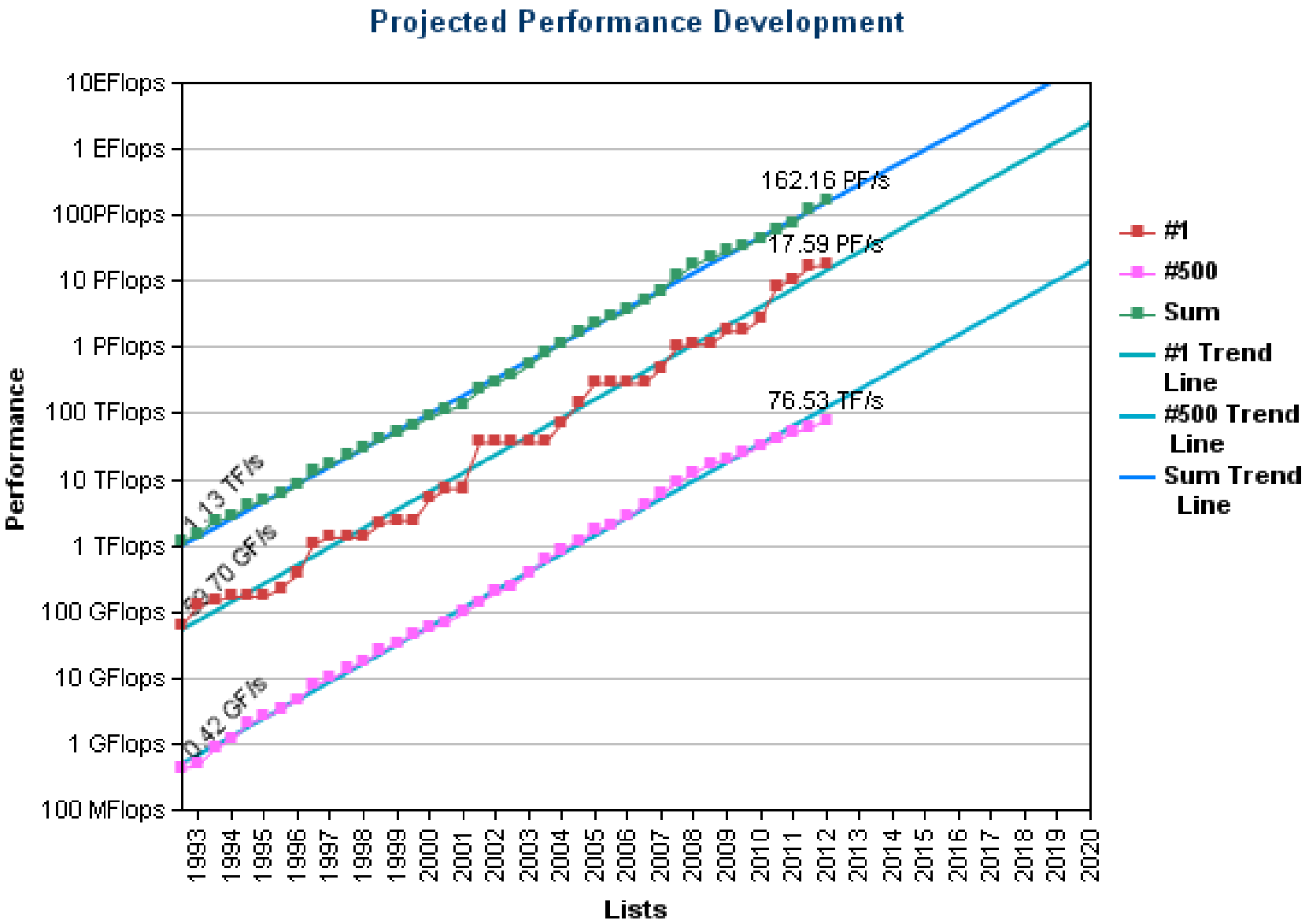
La complessità dei dispositivi (n. di transistor per square inch nei microprocessori) raddoppia ogni 18 mesi (nel futuro vale applicato al numero di core per processore).

Il LINPACK Benchmark

Come confrontare i supercomputer? Si usa il LINPAck Benchmark:

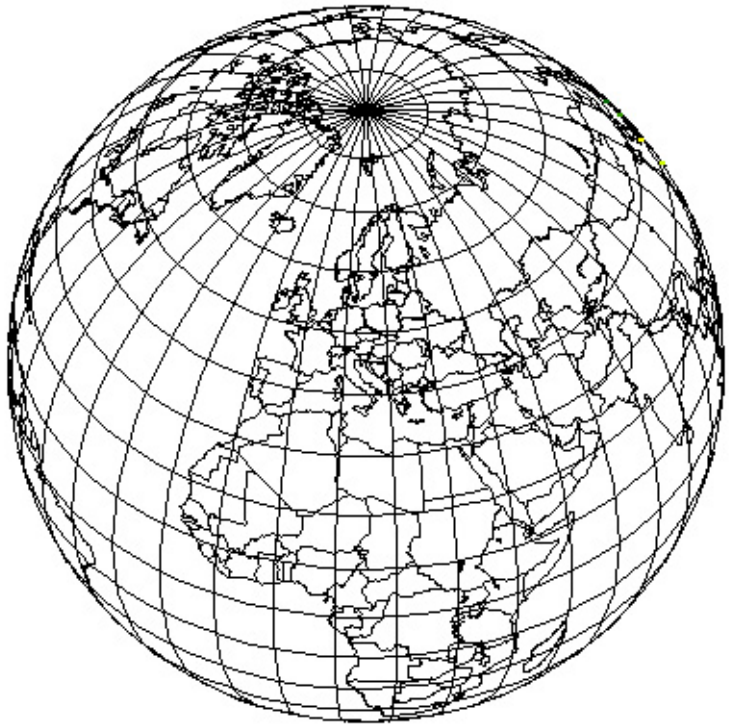
- introdotto da Jack Dongarra;
- si basa sulla risoluzione di un sistema denso di equazioni lineari;
- per la TOP500 si usa la versione del benchmark che permette all'utente di scalare il problema e ottimizzare il software per ottenere il miglior risultato;
- riflette la performance di un sistema dedicato per risolvere un sistema denso di equazioni lineari (il problema è molto regolare);
- misurare la performance per problemi di grandezza differente n permette non solo di misurare la performance migliore $\mathbf{R_{max}}$ per il problema di dimensione $\mathbf{N_{max}}$ ma anche per il problema $\mathbf{N_{1/2}}$ dove è raggiunta la metà della performance $\mathbf{R_{max}}$;
- con $\mathbf{R_{peak}}$ (picco teorico di performance) si può stilare la classifica della TOP500.
- per poter confrontare uniformemente i vari computer, l'algorimo usato nel risolvere il sistema di equazioni nella procedura del benchmark deve conformarsi alla operazione standard della LU con pivoting parziale: il conto delle operazioni per l'algoritmo deve essere $2/3 \cdot n^3 + \mathcal{O}(n^2)$ di operazioni floating point. Questo esclude la possibilità di usare algoritmo veloci come il metodo di *Strassen*.

Evoluzione TOP500



Esempio: previsioni del tempo

- Griglia 3D per la superficie terrestre:
  - circonferenza $\approx 4 \cdot 10^4$ Km;
  - raggio ≈ 6370 Km;
  - superficie $\approx 4\pi r^2 \approx 5 \cdot 10^8$ Km^2
- 6 variabili:
  - temperatura;
  - pressione;
  - umidità;
  - velocità del vento (x,y,z).



- Celle di 1 Km di lato, 100 slices in altezza (evoluzione ai vari livelli dell'atmosfera).
- Time step ogni 30 secondi di simulazione;
- 1000 operazioni per time-step ogni cella (Navier-Stokes, turbolenza, . . .);
- impensabile su scala globale (su scala locale celle di 10-15 Km di lato).

Esempio: previsioni del tempo

- Griglia di $5 \cdot 10^8 \cdot 100 = 5 \cdot 10^{10}$ celle (8 Byte per cella):
  - $(6 \text{ var}) \cdot (8 \text{ Byte}) \cdot (5 \cdot 10^{10} \text{ celle}) \approx 2 \cdot 10^{12}$ Byte = 2 TB;
- Per una previsione a 24 ore occorrono:
  - $24 \cdot 60 \cdot 2 \approx 3 \cdot 10^3$ timesteps;
  - $(5 \cdot 10^{10} \text{ celle}) \cdot (10^3 \text{ operazioni}) \cdot (3 \cdot 10^3 \text{ timesteps}) = 1.5 \cdot 10^{17}$ operazioni;

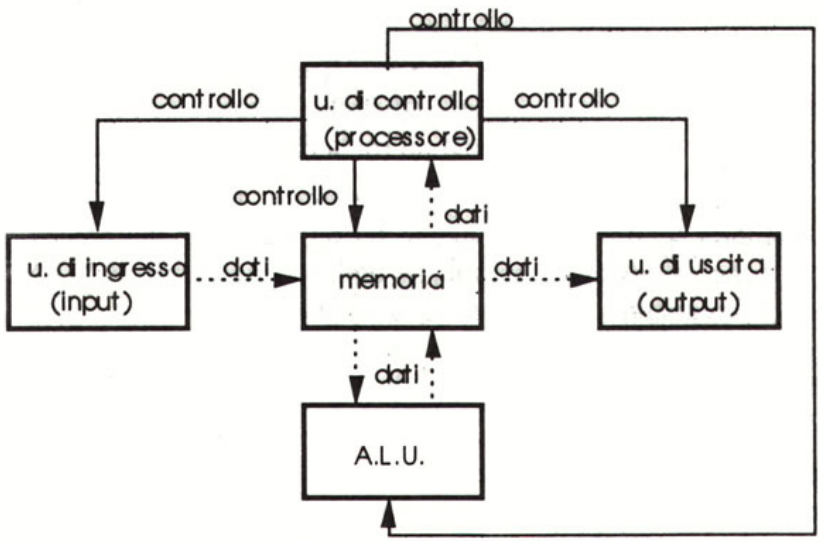
Un calcolatore con una potenza di 1Tflop/s impiegherà $1.5 \cdot 10^5$ secondi.

Previsioni a 24h si avranno in 2 giorni.

Molto accurate ma inutili.

Emilio Bellavite, storico metereologo veronese.

Modello di Von Neumann: singola CPU



Istruzioni processate sequenzialmente:

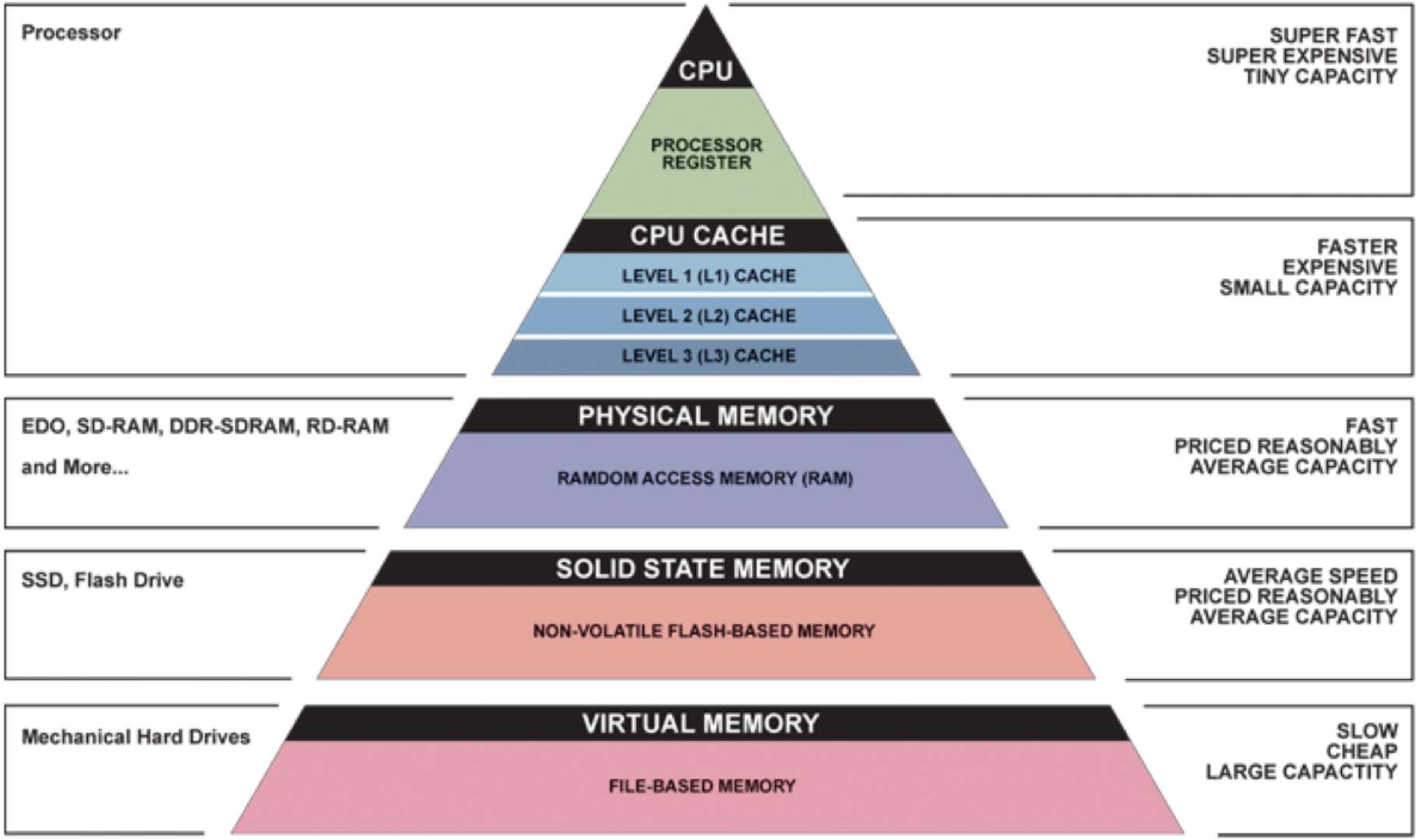
- una istruzione viene caricata dalla memoria (fetch) e decodificata ;
- vengono calcolati gli indirizzi degli operandi;
- vengono prelevati gli operandi dalla memoria;
- viene eseguita l'istruzione (una per ogni istante di tempo);
- il risultato viene scritto in memoria (store)

Ciclo macchina (clock period) τ

Intervallo temporale che regola la sincronizzazione delle operazioni in un calcolatore (unità di misura $1\text{ns} = 10^{-9}$): tutte le operazioni all'interno del processore durano un multiplo di τ.

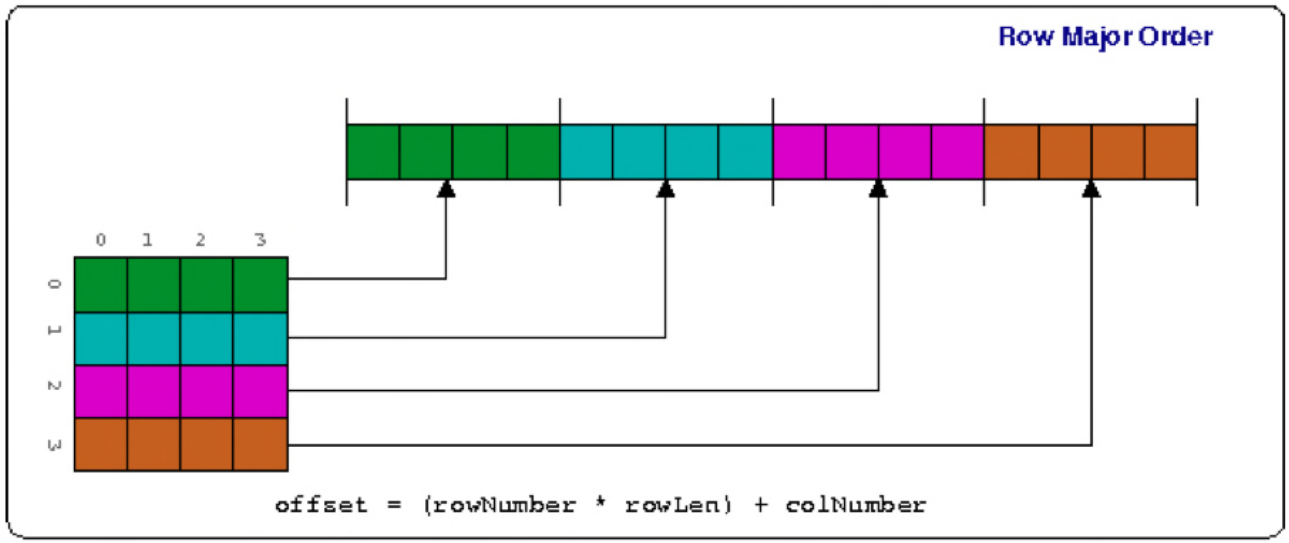
Per velocizzare i calcoli, qualcosa si può già fare:

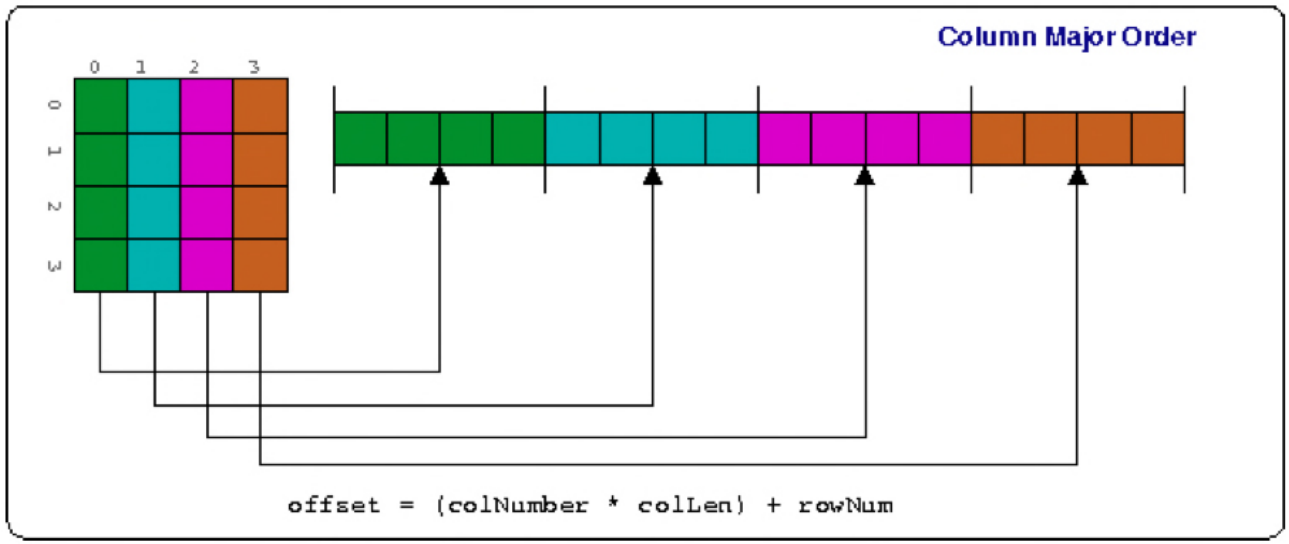
- minimizzare gli accessi alla memoria (performance 2x ogni 6 anni circa).



Per velocizzare i calcoli, qualcosa si può già fare:

- migliorare lo storage dei dati in memoria (evitando salti di lettura e scrittura).





Verso il parallellismo: più CPU

- L'esecuzione di un programma può essere vista come un'alternarsi di sequenze di calcoli e operazioni I/O.
- Durante le operazioni di I/O è preferibile iniziare o riprendere l'esecuzione di un altro programma piuttosto che tenere *idle* la CPU.

Un primo passo verso il parallelismo consiste nel suddividere le funzioni dell'ALU e progettare unità indipendenti capaci di operare in parallelo (unità funzionali indipendenti).

Con più core il concetto viene estremizzato.

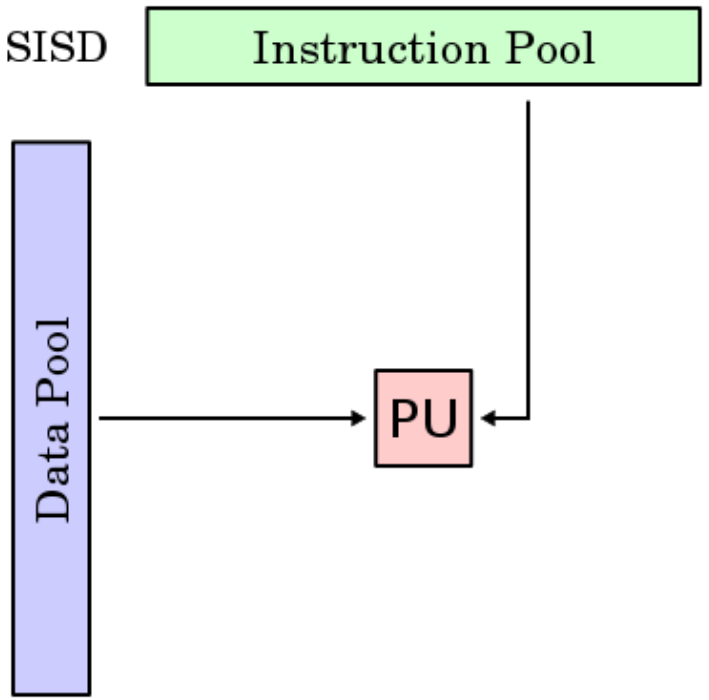
- il problema viene decomposto in parti che posso essere risolte in maniera concorrente (in parallelo);
- ogni parte è spezzata in sequenze di istruzioni da eseguire su una singola CPU;
- le istruzioni di ognuna delle parti sono eseguite simultaneamente su CPU differenti.

Classificazione di Flynn

1966: Michael J. Flynn propone una classificazione delle architetture dei computer.

Single Instruction Single Data:

- una singola unità esegue un singolo flusso di dati;
- corrisponde alla classica architettura di Von Neumann;
- singola connessione tra il processore e la memoria: collo di bottiglia (una istruzione per volta);
- ormai obsoleta.

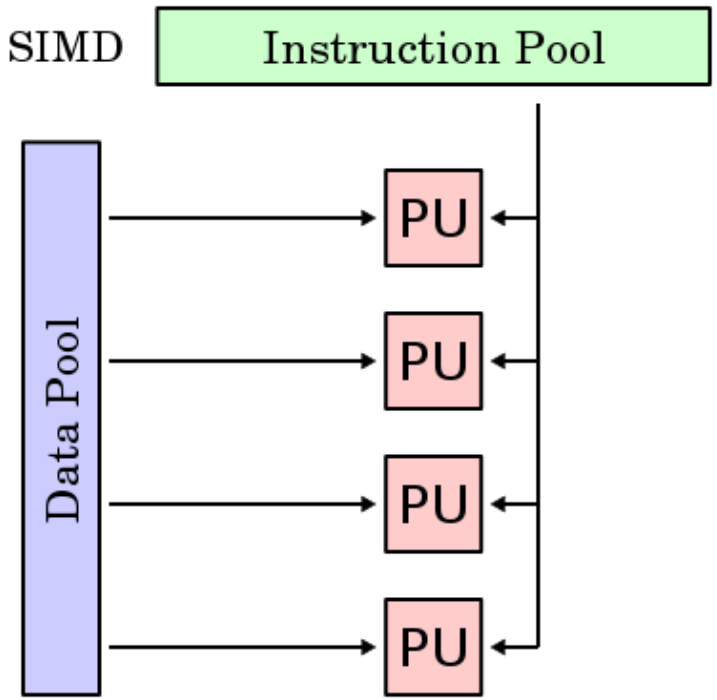


Classificazione di Flynn

1966: Michael J. Flynn propone una classificazione delle architetture dei computer.

Single Instruction Multiple Data:

- più unità elaborano dati diversi in parallelo;
- unica unità di controllo che controlla le ALU;
- più ALU che operano in maniera sincrona;
- stessa istruzione su dati differenti.

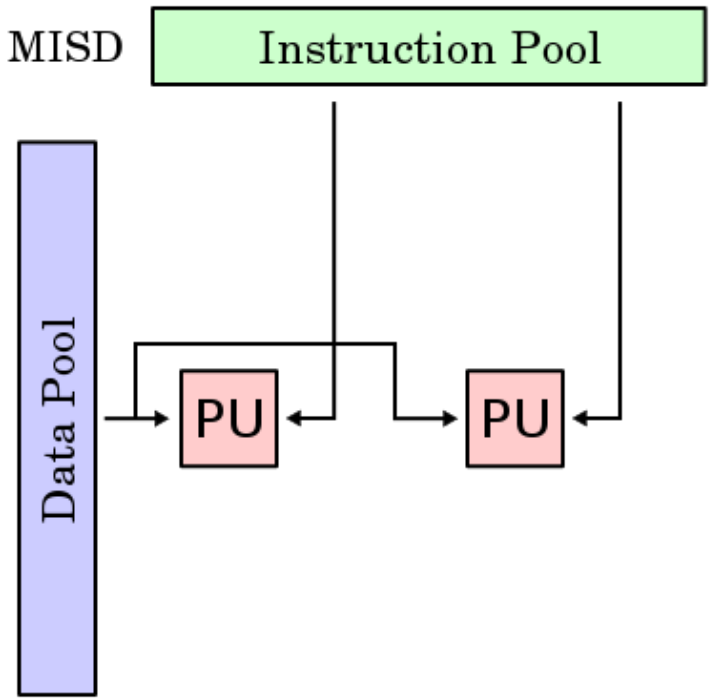


Classificazione di Flynn

1966: Michael J. Flynn propone una classificazione delle architetture dei computer.

Multiple Instruction Single Data:

- diverse unità effettuano diverse elaborazioni sugli stessi dati;
- prevista fin dal 1972;
- mai realizzata.

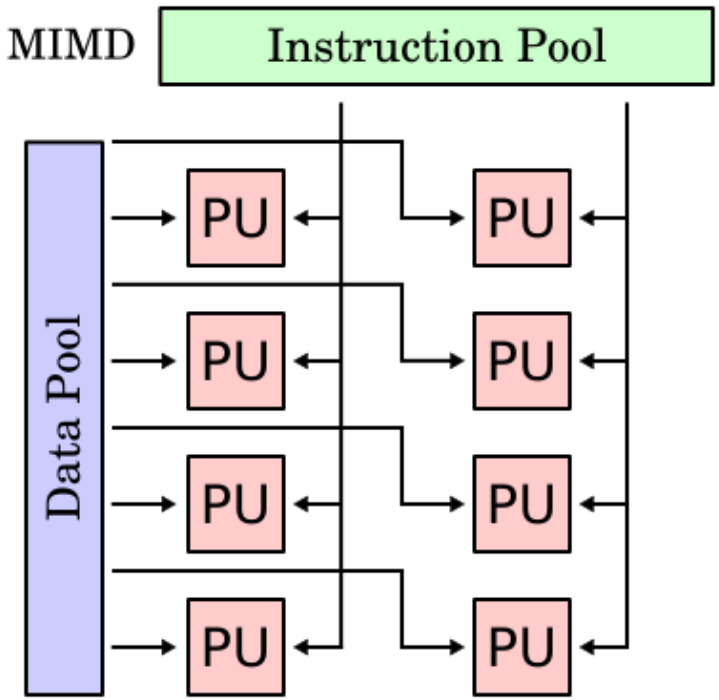


Classificazione di Flynn

1966: Michael J. Flynn propone una classificazione delle architetture dei computer.

Multiple Instruction Multiple Data:

- diverse unità effettuano diverse elaborazioni su dati diversi;
- processori operano in parallelo in modo asincrono;
- ogni processore risolve un sotto-problema;
- comunicazione tra i vari processori mediante memoria condivisa o rete.



Valutazione delle prestazioni

Come scala il sistema parallelo?

- tempo esecuzione seriale: tempo che intercorre tra l'inizio e la fine dell'esecuzione del programma su un solo processore;
- tempo esecuzione parallelo: tempo che intercorre tra l'inizio dell'esecuzione parallela e il momento in cui l'ultimo processore termina l'esecuzione.
- speed-up: $S_p^{(\tilde{p})} = \frac{T_{\tilde{p}}\tilde{p}}{T_p} = \frac{\text{tempo esecuzione sequenziale del miglior algoritmo sequenziale conosciuto}}{\text{tempo di esecuzione su p processori}}$;
- efficienza parallela relativa: $E_p^{(\tilde{p})} = \frac{S_p^{(\tilde{p})}}{p}$.

Un programma scala per un numero di processori P, se passando da $p-1$ a p processori si osserva un miglioramento in termini di speed-up.

Legge di Amdahl: *Make the common case fast*

L'aumento di velocità complessivo prodotto dal miglioramento sarà $\frac{1}{(1-P)+P/S}$.

Esempio: se un miglioramento può velocizzare il 30% del calcolo, P sarà 0.3, se il miglioramento raddoppia la velocità della porzione modificata, S sarà 2.

Case study

Prodotto matrice sparsa-vettore

Matrici in formato sparso

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

Compressed Sparse Row Format:

$$\text{A.data} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];$$

$$\text{A.JA} = [1, 4, 1, 2, 4, 1, 3, 4, 5, 3, 4, 5];\ \text{(posizione nella colonna)}$$

$$\text{A.IA} = [1, 3, 6, 10, 12, 13];\ \text{(posizione in A.data di inizio nuova riga)}$$

Analogamente, il Compressed Sparse Column Format leggerà A in colonna:

$$\text{A.data} = [1, 3, 6, 4, 7, 10, 2, 5, 8, 11, 9, 12];$$

$$\text{A.IA} = [1, 2, 3, 2, 3, 4, 1, 2, 3, 4, 3, 5];\ \text{(posizione nella riga)}$$

$$\text{A.JA} = [1, 4, 5, 7, 11, 13];\ \text{(posizione in A.data di inizio nuova colonna)}$$

Prodotto Matrice-Vettore in CSR (seriale)

$$A.data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];$$
$$A.JA = [1, 4, 1, 2, 4, 1, 3, 4, 5, 3, 4, 5];$$
$$A.IA = [1, 3, 6, 10, 12, 13];$$
$$x = [1; 1; 1; 1; 1];$$

```
function CSR_MATRIX-VECTOR_PRODUCT(A(csr_format),x,b,number_rows)
t = 1;
    for (i=1;i≤number_rows;i++) do
        b(i)(1) = 0;
        k1 = A.IA(t);
        k2 = A.IA(t+1)-1;
        for (k=k1;k≤k2;k++) do
            b(i)(1) += (A.data(1)(k))×(x(A.JA(k))(1));
        end for
    end for
++t;
end function
```

Prodotto Matrice-Vettore in CSR (parallelo, numprocs=2)

$$Ax = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} A1 \\ A2 \end{bmatrix} [x] = \begin{bmatrix} 3 \\ 12 \\ 30 \\ 31 \\ 12 \end{bmatrix} = b.$$

$$A1.data = [1, 2, 3, 4, 5];$$

$$A1.JA = [1, 4, 1, 2, 4, 1];$$

$$A1.IA = [1, 3, 6]$$

$$A2.data = [6, 7, 8, 9, 10, 11, 12];$$

$$A2.JA = [3, 4, 5, 3, 4, 5];$$

$$A2.IA = [1, 5, 7, 8];$$

$$x = [1; 1; 1; 1; 1]; \text{(conosciuto da tutti i processi)}$$

SPMD ed MPI

Single Program Multiple Data:

- Tutti i processi eseguono lo stesso programma, ognuno su dati diversi;
- il flusso del programma (con il rank locale), differenzia le operazioni (**if**. . . **else**. . .);
- sottocategoria di MIMD (SIMD è sincrono e richiede processori vettoriali);
- memoria condivisa (area di memoria visibile dai processi contenente i dati);
- memoria distribuita (ogni processo dispone localmente solo dei dati necessari);

Message Passing Interface

- protocollo di comunicazione per computer;
- standard per la comunicazione tra nodi appartenenti a un cluster di computer
- portabile (disponibile per molte architetture) e veloce.
- OpenMPI è una (forse la migliore) implementazione di MPI (standard MPI-2.2).

Gestione dei rank in MPI

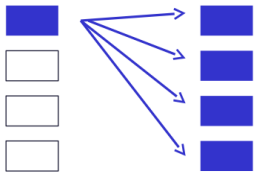
- 1 rank = 1 CPU;
- Header: #include<mpi.h> (in C) oppure include `mpif.h' (in Fortran)
- Formato: int error = MPI_Xxx(param,. . .); MPI_Xxx(param,. . .);
- Inizializzazione: int MPI_Init(int*argc, char***argv)
- In MPI è possibile dividere il numero totale di processi in gruppi detti *comunicatori*: MPI_Comm_World (default) racchiude tutti i processi;
- Si possono definire topologie cartesiane e periodiche con MPI_Cart_Create(comm_old, ndims, dims, periods, reorder, comm_cart)
- Per contare il numero di processi in un comunicatore: MPI_Comm_size(MPI_Comm comm, int *size);
- ID del processo nel comunicatore: MPI_Comm_rank(MPI_Comm comm, int *rank) (va da 0 a numprocs-1);
- Termine di MPI: int MPI_Finalize();

Comunicazioni Punto a punto

A manda l'informazione a B, B riceve l'informazione da A:
If rank == A **then** send information to B
elseif rank == B **then** receive information from A.
endif endif

- Comunicazioni bloccanti (non prosegue se non è completato):
  - Send: int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);
  - Receive: int MPI_Recv (void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Status *status);
- Comunicazioni non bloccanti (spesso per evitare *deadlock*):
  - Send: int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);
  - Receive: int MPI_Irecv (void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req);

Comunicazioni collettive

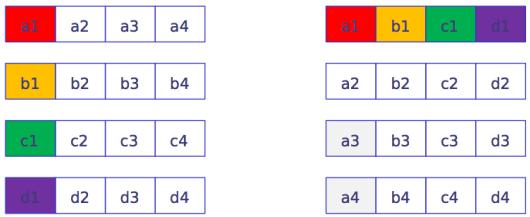


MPI_Bcast

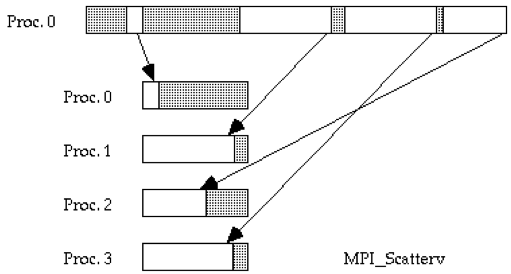
MPI_Gather

MPI_Scatter

MPI_Allgather



MPI_Alltoall



MPI_Scatterv

Altre funzioni disponibili:

- MPI_Barrier (es. per calcolare i tempi sincronizzando i processi);
- MPI_Gatherv (es. per raccogliere i risultati parziali);
- e tante altre.

Implementazione in MPI

- memoria condivisa: t=localindex(1) al posto di t=1 (tiene conto di dove andare a leggere i dati);
- memoria distribuita: si applica l'algoritmo seriale ai dati parziali;

int MPI_Gatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype,void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);

	rank 0	rank 1
sendbuf	$\begin{bmatrix}3 & 12\end{bmatrix}$	$\begin{bmatrix}30 & 31 & 12\end{bmatrix}$
sentcnt	2	3
sendtype	MPI_Double	MPI_Double
*recvbuf	b(1)(1)	b(1)(1)
*recvcnts	$\begin{bmatrix}2 & 3\end{bmatrix}$	$\begin{bmatrix}2 & 3\end{bmatrix}$
*displs	$\begin{bmatrix}0 & 2\end{bmatrix}$	$\begin{bmatrix}0 & 2\end{bmatrix}$
recvtype	MPI_Double	MPI_Double
root	leader	leader
comm	comm_cart	comm_cart

Test e risultati: $Ax = b$

Alcune informazioni sul test effettuato:

- Memoria condivisa:
  - size(A) = (30000,30000) (900.000.000 elementi) (.txt di 21 MB);
  - size(b) = (30000,1);
  - A.nnz = 899544 (min per riga: 11, max per riga 58) = 30 · 30000;
- Memoria distribuita:
  - size(A) = (150000,150000) (oltre 22 miliardi di elementi) (.txt di 26 MB);
  - size(b) = (150000,1);
  - A.nnz = 1049979 (min per riga: 1, max per riga 20) = 7 · 150000;

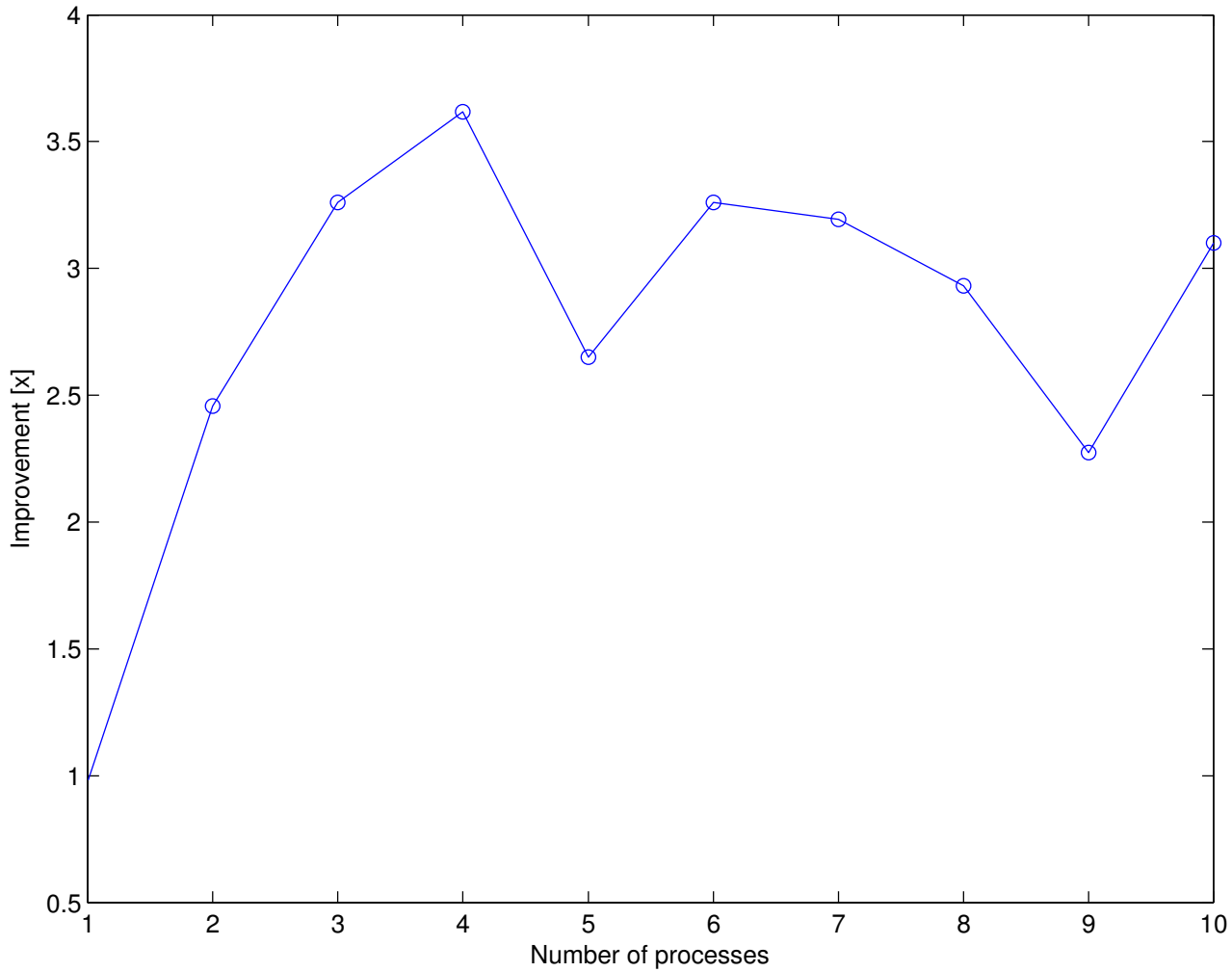
Per compilare:

```
mpicc -O0 -std=c99 -g -Wall -pedantic spmv.c -o ./spmv
```

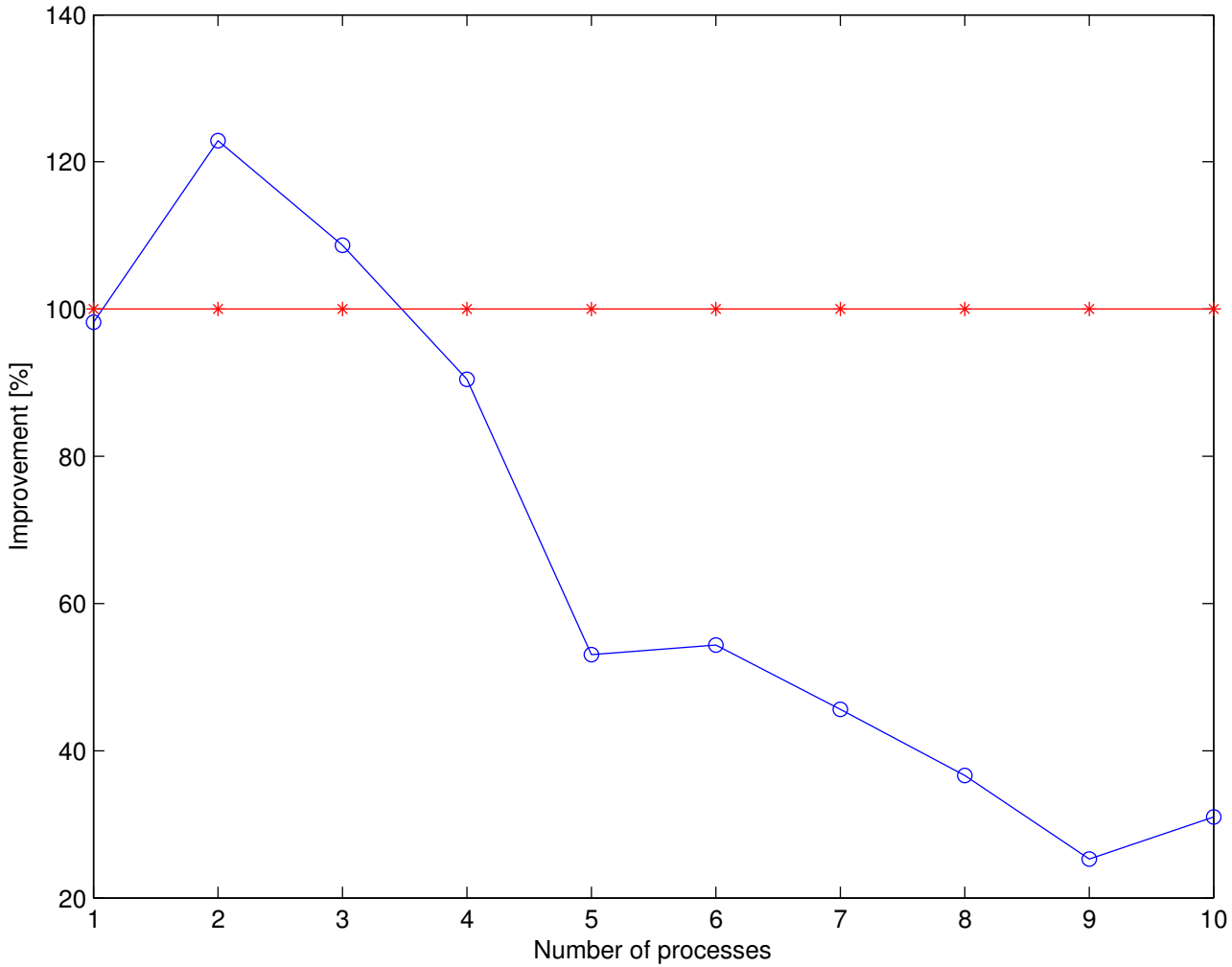
Per eseguire:

```
mpirun -n 2 spmv
```

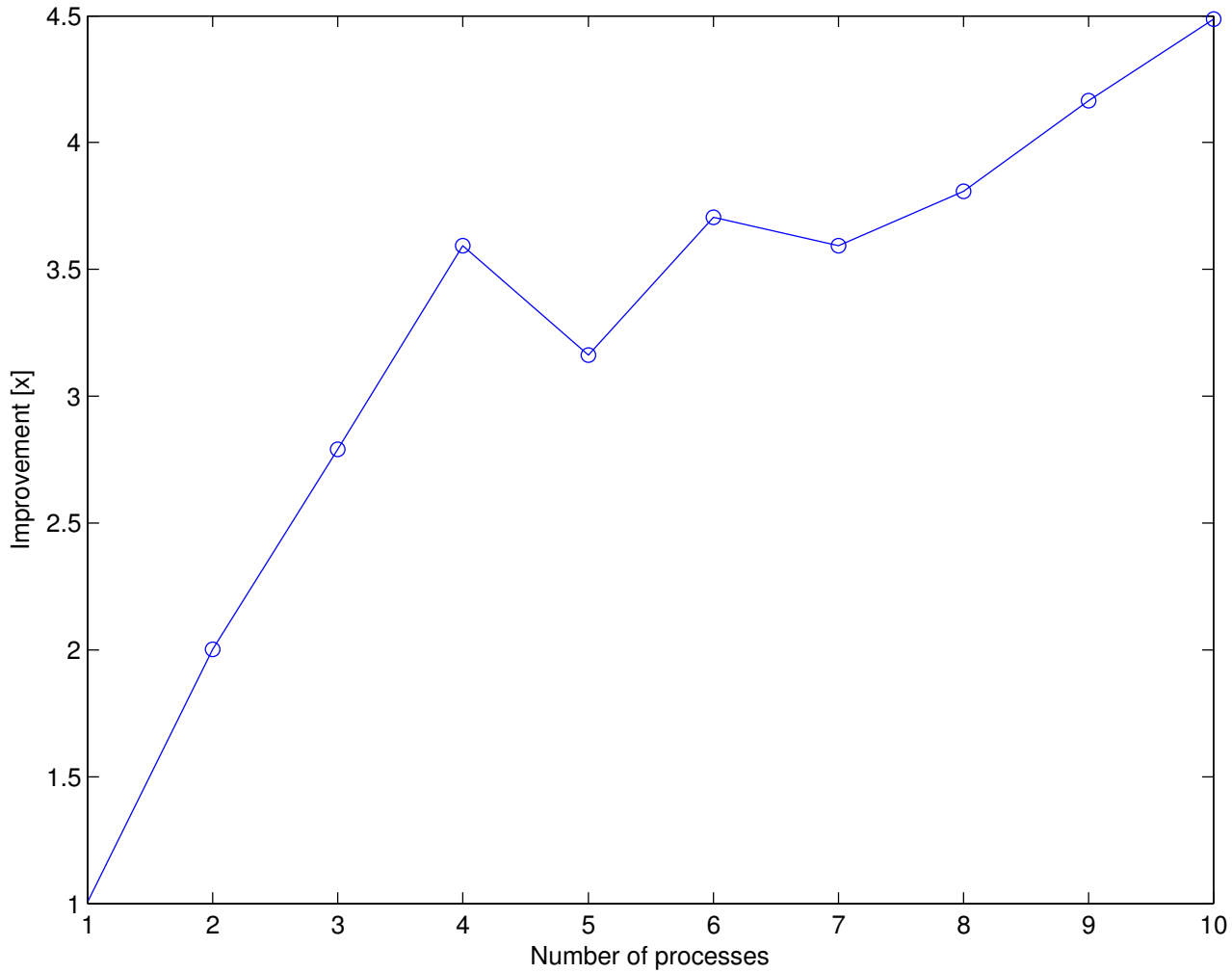
Calcolo parallelo a memoria condivisa: speed up



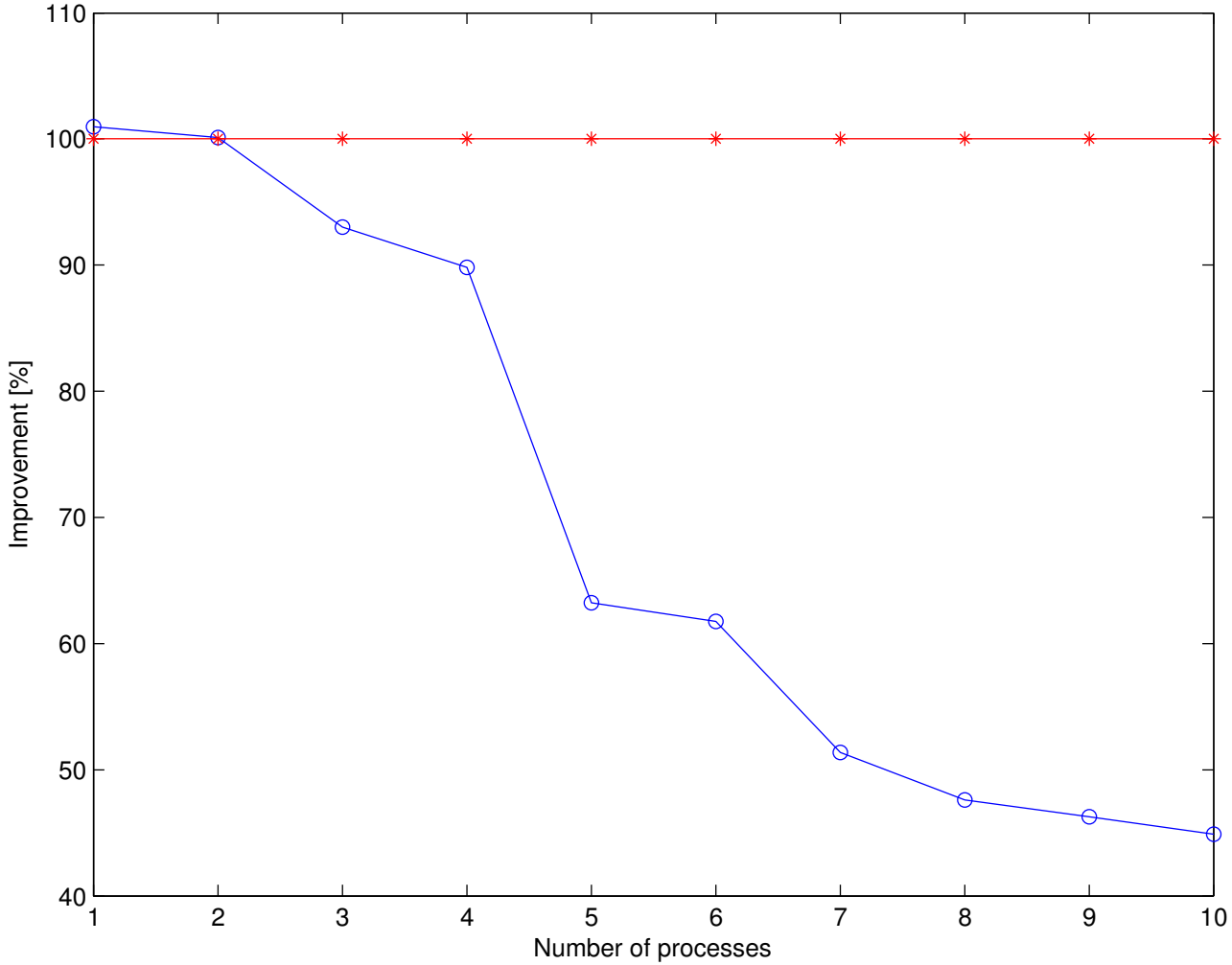
Calcolo parallelo a memoria condivisa: efficienza parallela



Calcolo parallelo a memoria distribuita: speed up



Calcolo parallelo a memoria distribuita: efficienza



Future works

- Test affidabili fino ad $n = 4$ (poi multithreading non controllabile);
- algoritmo pensato per essere eseguito su N *infiniti* processi ($N \leq \text{num\_righe}$).

In linea teorica si potrebbero migliorare le prestazioni:

- bilanciando i calcoli rispetto ai processi (gestendo meglio resti ed eccezioni);
- disponendo di un cluster (rispetto ai 4 core disponibili) su cui fare i test;
- CPU più performanti o sfruttando la GPU.

Conclusioni

Pensare in parallelo non è facile, soprattutto in fase di debug.

Scrivere un codice che funzioni in parallelo permette di:

- sfruttare totalmente la potenza della macchina;
- usare risorse distribuite su tutto il pianeta;
- abbattere i tempi di calcolo;
- abbattere i costi;
- fare calcoli più accurati;
- fare calcoli più complessi.

Grazie per l'attenzione.