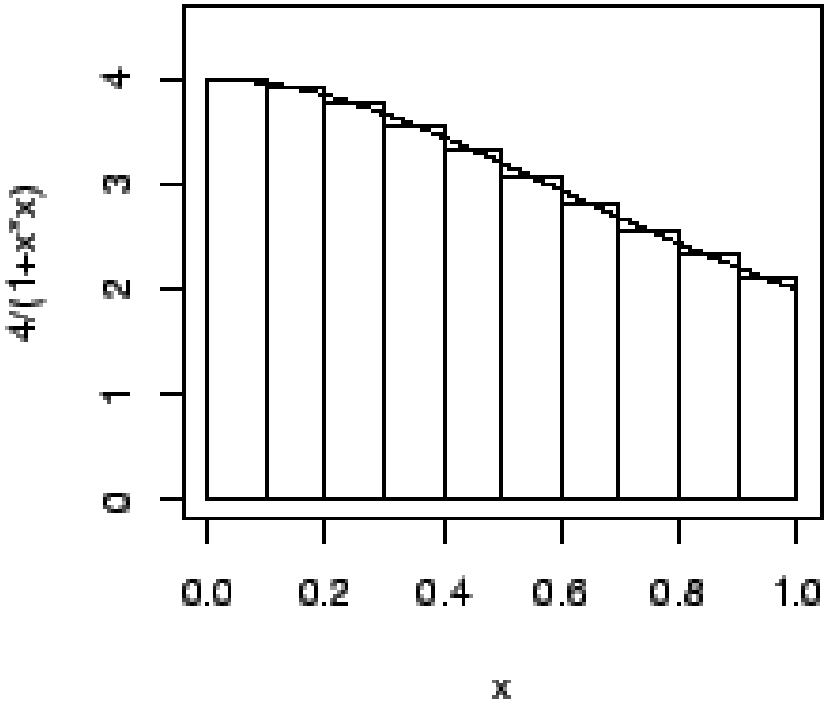
Calcolo di PI

Calcolo PI

$$\pi = \int_0^1 \frac{4}{1+x^2}\mathrm{d}x \sim \frac{1}{n}\sum_{i=1}^{n} \frac{4}{1+\left(\frac{i-0.5}{n}\right)^2}.$$

Per n = 10 vedi la figura:



Provate ad implementarlo

Calcolo PI

```
int n, i;
double d, s, x, pi;

n = NUMOFP;
d = 1.0/n;
s = 0.0;

for (i=1; i<=n; i++){
  x = (i-0.5)*d;
  s += 4.0/(1.0+x*x);
}

pi = d*s;
```

Calcolo PI – Process forking

Un processo e' un cotesto (un'entita') nel quale viene seguito un determinato codice.

Ogni processo ha il proprio stack, le proprie pagine di memoria, la propria file descriptors table, un proprio PID. (due parole su come e' implemtato il multi-tasking).

Proveremo a realizzare un prima parallelizzazione dell'algoritmo visto sopra usando fork(), wait() ed meccanismi base di IPC (Inter-Process Comunication).

Calcolo PI – Process forking

```
$ time ./forking 100000000
computed pi value = 3.14159265359002226603025...
                    3.14159265358979323846264...

real    0m1.472s
user    0m2.872s
sys     0m0.006s

$ time ./seq 100000000
computed pi value = 3.14159265359042638721121...
                    3.14159265358979323846264...

real    0m2.836s
user    0m2.830s
sys     0m0.006s
```

Speedup

Vediamo come e' possibile caratterizzare e studiare le performance di un algoritmo parallelo:

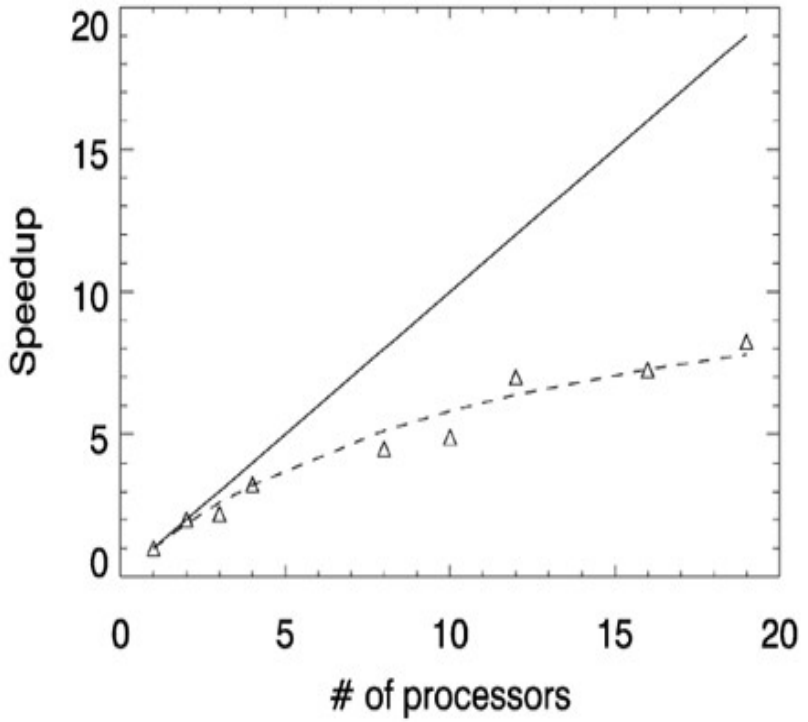
$$S_n = \frac{T_1}{T_p(n)}$$

Dove T_1 e' il tempo impegato dal miglior algoritmo seriale conosciuto, mentre $T_p(n)$ e' il tempo di eseucuzione dell'algoritmo parallelo su n processori.

Si definisce anche lo Speedup relativo in cui, invece di usare T_1 si usa $T_p(1)$.

Speedup

Il miglior algoritmo parallelo possibile presenta uno speedup lineare, quindi $S_n = n$, in pratica questo limite e' difficilmente raggiungibile.



Speedup ed Efficienza

Speedup superlineare: e' il caso in cui $S_n > n$. Casi del genere sono rari, ma ci sono differenti motivazioni che possono portare a rislutati del genere. Ad esempio effetti della cache (le dimensioni totali delle cache dei vari processori possono essere tali da contenere tutti i dati relativi al problema).
Oltre allo speedup e' possibile definire l'**efficienza**:

$$E_n = \frac{S_n}{n}$$

il suo valore tipicamente varia tra zero ed uno, ed e' una misura di quanto bene il nostro algoritmo parallelo sfrutta i processori. Anche in questo caso e' possibile parlare di efficienza relativa.

Scalabilita'

Costo: quanto uso di CPU e' richiesto

$$C = n\,T_p(n) = T_1 / E$$

Scalabilita': capacita' di rimanere efficiente su una macchina parallela aumentando il numero di processori (o comunque aumento le dimensione del problema proporzionalmente al numero di processori usati).

Legge di Amdahl

La legge di Amdahl serve a determinare il miglioramento massimo ottenibile quando solo una parte del codice e' stata migliorata:

$$\frac{1}{\sum_{k=0}^{n}\left(\frac{P_k}{S_k}\right)}$$

- P_k e' la percentuale di codice che puo' essere migliorata (o peggiorata)
- S_k e' lo speedup relativo alla k-esima parte di codice di cui sopra
- k semplicemente indicizza le varie parti di codice

Legge di Amdahl

Nel caso specifico del calcolo parallelo la legge puo' essere semplificata come segue:

$$S_n = n / (n*F + (1 - F))$$

dove F (0 < F < 1) e' la frazione di codice che non puo' essere parallelizzata.

Quindi data una frazione di codice che e' stata parallelizzata, ed il numero di processori che possiamo usare, la legge ci dice quale e' il massimo speedup che possiamo aspettarci.

Legge di Amdahl

Vediamo come si deduce tale legge. Se t_s e' il tempo impiegato dal singolo processore per il completamento del task, nel caso parallelo avremo un tempo pari a $F*t_s + [(1-F)*t_s / n]$ e dunque la legge di Amdahl dice che:

$$S_n = t_s / (F*t_s + [(1-F)*t_s / n])$$

$$S_n = 1 / (F + [(1-F) / n])$$

$$S_n = n / (n*F + (1-F))$$

(1 – F) rappresenta chiaramente la frazione di codice parallelo.

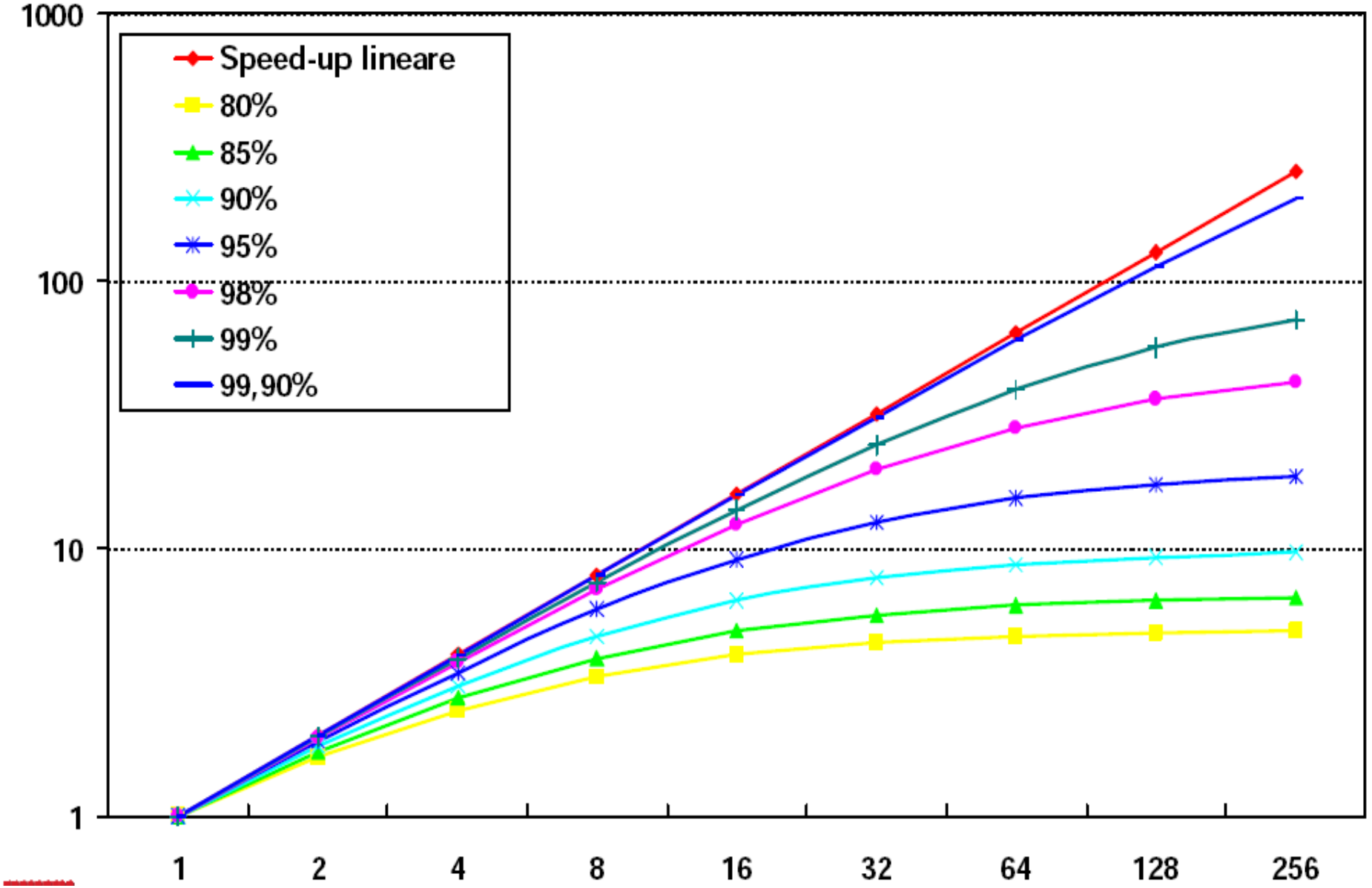
Legge di Amdahl

Quando n tende all'infinito otteniamo:

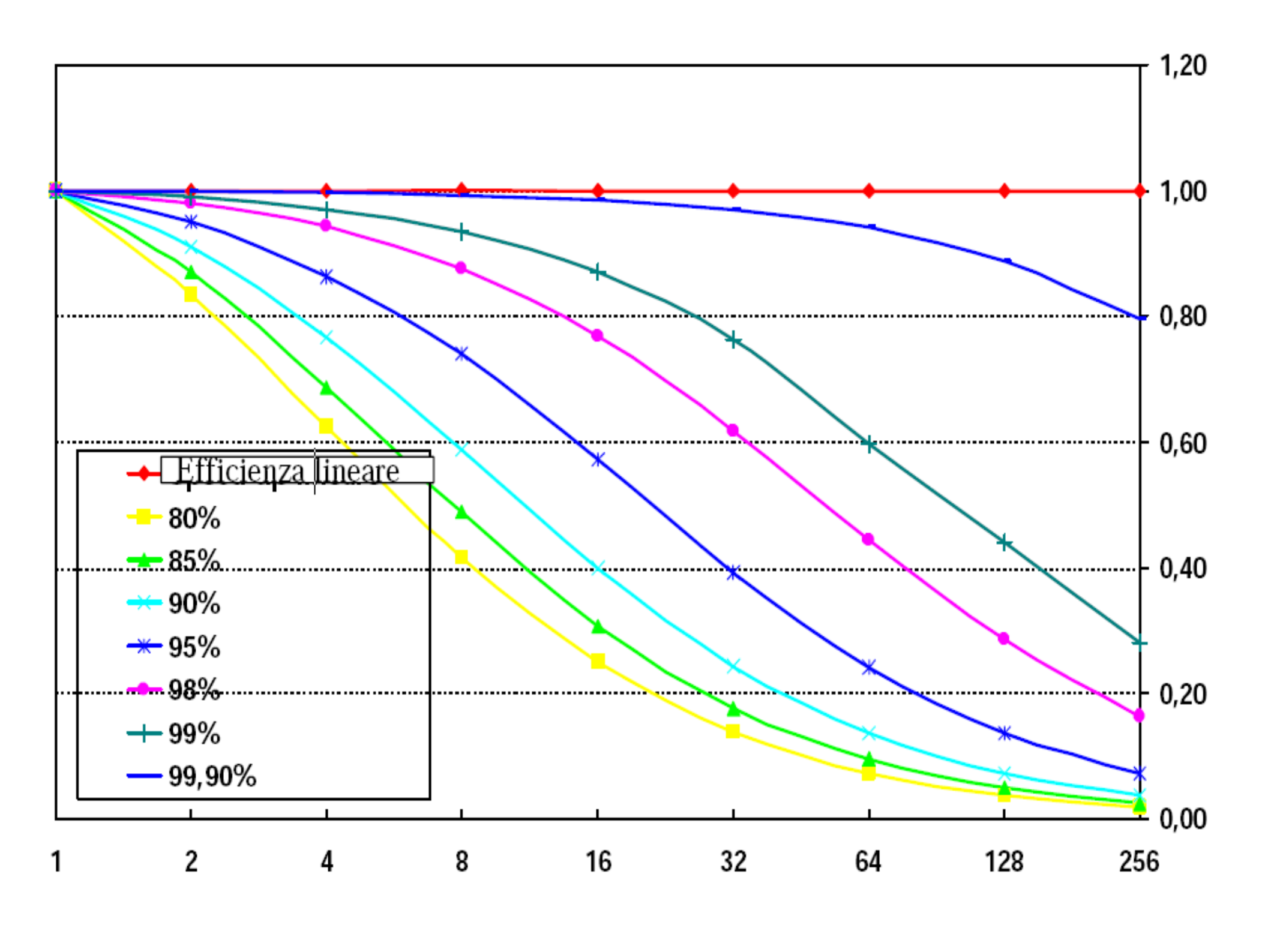
$$S_n = 1 / F$$

che e' il massimo speedup che possiamo aspettarci, a prescindere dal numero di processori nel caso in cui riusciamo a parallelizzare una frazione (1 – F) del codice in questione.

Se la percentuale di codice seriale e' il 10% (F = 0.10) il massimo speedup ottenibile sara' 10 a prescindere dal numero di processori utilizzato. Dunque in generale se ne deduce che il parallelismo e' utile o quando si usa un piccolo numero di processori, o per quei problemi cosi' detti embarrassingly parallel.



Chiaramente se F tende a zero S_n tende ad n


1,20
1,00
0,80
0,60
0,40
0,20
0,00
1
2
4
8
16
32
64
128
256
Efficienza lineare
80%
85%
90%
95%
98%
99%
99,90%

Altri Costi

Load Balancing

Se dividiamo un lavoro in tanti “task” piu' piccoli e li eseguiamo in parallelo dobbiamo alla fine aspettare fino che tutti i “task” non sono stati completati.

La veolcita' del lavoro e' dunque determinata da quello del “task” piu' lento. Tutti i task devono essere disegnati di modo da avere piu' o meno lo stesso tempo di esecuzione. Se cosi' non fosse, potremmo trovarci nella situazione di avere qualche processore inutilmente in “idle”.

Ottenere un buon “load balancing” non e' sempre semplice.

Sincronizzazione

In tutti i sistemi shared memory lo scambio di dati tra i processori avviene mediante variabili memorizzate nella memoria condivisa.

Chiaramente spesso si pone il problema della sincronizzazione, cioe' dell'accesso simultaneo in lettura e scrittura di tali variabili, da parte di piu' processori.

Questo problema di sincronizzazione e' tipicamente gestisto mediante meccanismi di “lock-unlock”. Questo comporta in generale una certa perdita di tempo.

Altri problemi sorgono a causa della necessita' di mantenere la coerenza della “cache” (Questo spesso comporta una certa perdita dei vantaggi dati dalla presenza stessa della cache)

Tempi di comunicazione (memoria distribuita)

Rispetto alle macchine seriali i tempi di comunicazione sono fondamentali, in particolare per i cluster:

Nome	Vendor	CPU	N. CPU	Peak (GF)	Sustained (GF)
K.I.S.T	IBM	Intel Xeon (2.4 Ghz)	1024	4915	3067
TotalFinalElf	IBM	Intel Xeon (2.4 Ghz)	1024	4915	1755

Dove e' la differenza ? Esclusivamente nelle rete di interconnessione:

1. Myrinet
2. Gigabit Ethernet

Parametri importanti: latenza, banda,

Limiti della legge di Amdahl

La legge di Amdahl (1967) pare mettere un limite considerevole allo sviluppo del calcolo parallelo. Tuttavia da quando e' stata formulata i fatti hanno mostrato che anche altri fattori vanno considerati:

La legge di Amdahl si basa su alcuni presupposti che non sono sempre del tutto veri:

- basta pensare a casi di speedup superscalari (cumulative cache size)
- la legge di Amdhal parte dal presupposto che l'argoritmo seriale sia sempre e comunque la miglior soluzione al problema. Alcuni problemi si prestano meglio ad essere risolti con algoritmi paralleli
- **la legge di Amdahl assume che la dimensione del problema rimanga la stessa al crescere del numero di processori. Nella maggior parte dei casi piu' processori implicano anche la possibilita' di affrontare problemi piu' grandi.**

La legge di Gustafson

Questa legge e' datata 1988. Indichiamo con con s il tempo impiegato all'esecuzione del codice sequenziale, mentre con p il tempo impegato nella parte parallela. Se usiamo n processori immaginiamo anche di crescere la dimensione del problema di n volte. Al crescere di n supponiamo che s rimanga fisso (ad esempio s e' il tempo di inizializzazione del task), allora avro':

$$T_1 = s + n*p$$
$$T_p(n) = s + p$$

dunque:

$$S_n = (s + n*p) / (s + p)$$

Dove S_n e' lo speedup per n processori. (Se usiamo F = s / (s + n * p) riotteniamo la legge di Amdahl)

La legge di Gustafson

Quando n cresce S_n cresce linearmente. Se s tende a zero allora otteniamo:

$$S_n = n$$

Gustafson's law argues that even using massively parallel computer systems does not influence the serial part and regards this part as a constant one. In comparison to that, the hypothesis of Amdahl's law results from the idea that the influence of the serial part grows with the number of processes.
(vedi **gust**)

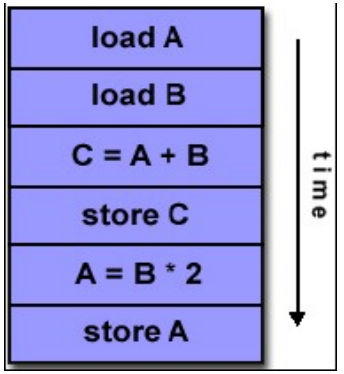
Tassonomia di Flynn

Ci sono diversi modi per classificare i computer paralleli, una delle classificazioni storicamente più usate è nota come tassonomia di Flynn ed e' usata fin dagli anni ’60:

- La tassonomia di Flynn distingue le architetture multi processore secondo due parametri indipendenti: Istruzioni e Dati.
- Ognuno di questi parametri può avere due stati: Single o Multiple:
  - **MIMD** Multiple Instruction, Multiple Data
  - **MISD** Multiple Instruction, Single Data
  - **SIMD** Single Instruction, Multiple Data
  - **SISD** Single Instruction, Single Data

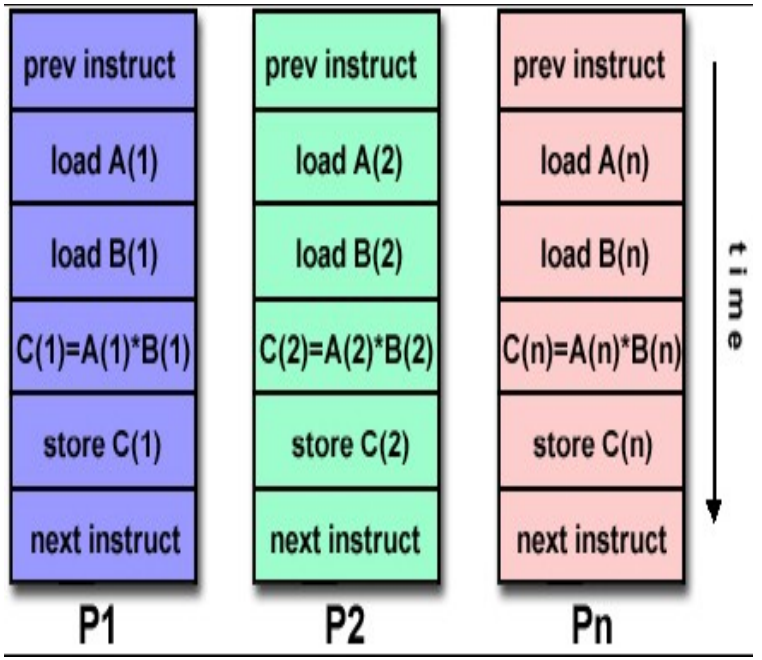
SISD

- Un computer seriale
- Single instruction: solo un flusso di istruzioni e' attivo nella CPU in ogni ciclo di clock
- Single data: un flusso di dati puo' essere usato come input durante ogni ciclo di clock
- L’architettura prevalente dei computer
- Esempi: i PC ordinari, workstation a singola CPU e tradizionali mainframe



SIMD

- Single instruction: tutti i processori eseguono la stessa istruzione ad ogni ciclo di clock
- Multiple data: ogni processore può operare su dati diversi



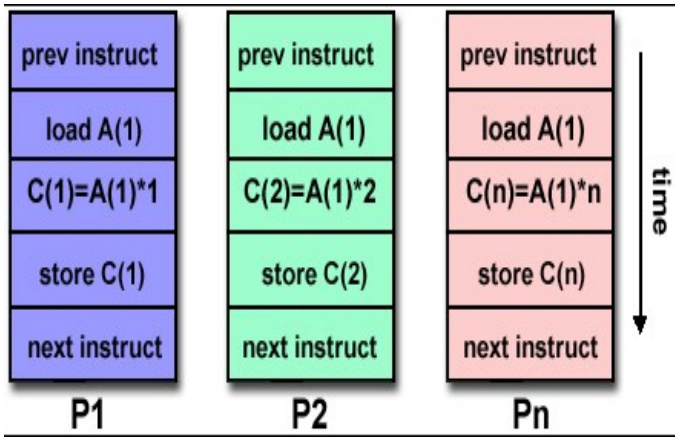
SIMD

- Tipicamente implementata con un nodo controller (responsabile del flusso d'istruzioni), una grande array di unità di calcolo di capacita' medio piccola ed un network d'interconnessione veloce
- Si adatta a specifici problemi, con un alto grado di regolarita' (es. image processing)

- Esempi:
  - Le macchine APE, progettate e realizzate dall'INFN (commercializzata poi dalla societa' Quadrics. Linguaggio di programmazione TAO che e' una versione estesa del Fortran)
  - I processori multimediali (mostrano una forma limitata di parallelismo di tipo SIMD)
  - I calcolatori vettoriali

MISD

- Un singolo flusso di dati e' elaborato da piu' unita' di processing.
- Ogni unita' di processing opera sui dati indipendentemente attraverso un proprio flusso di istruzioni

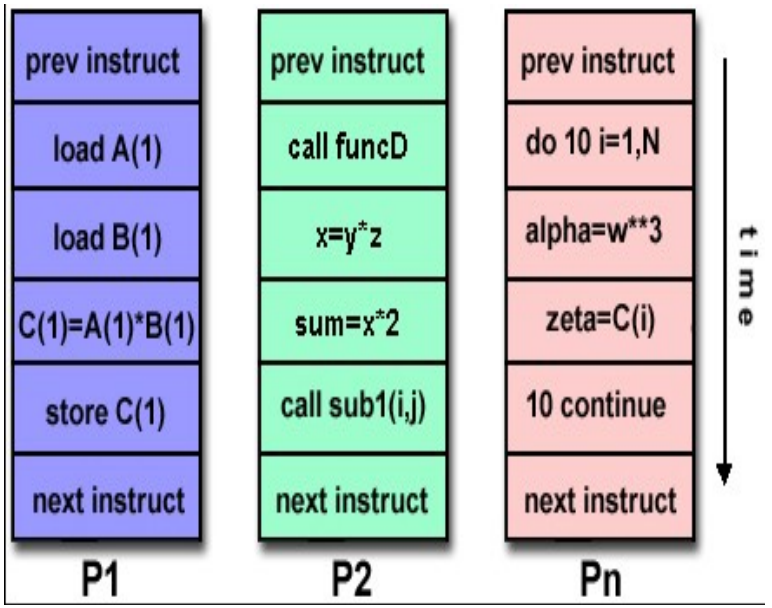


MISD

- Sono esistiti pochissimi esempi di questo tipo d’architettura, come il computer sperimentale Carnegie-Mellon C.mmp (1971)
- Usi concepibili:
  - Filtri multi frequenza da applicare su un singolo segnale di input
  - Applicazione di diversi algoritmi per decrittare lo stesso set di dati codificato
- Architetture tipo Pipeline possono essere considerate MISD (anche se ad ogni stage della pipeline il dato e' differente)

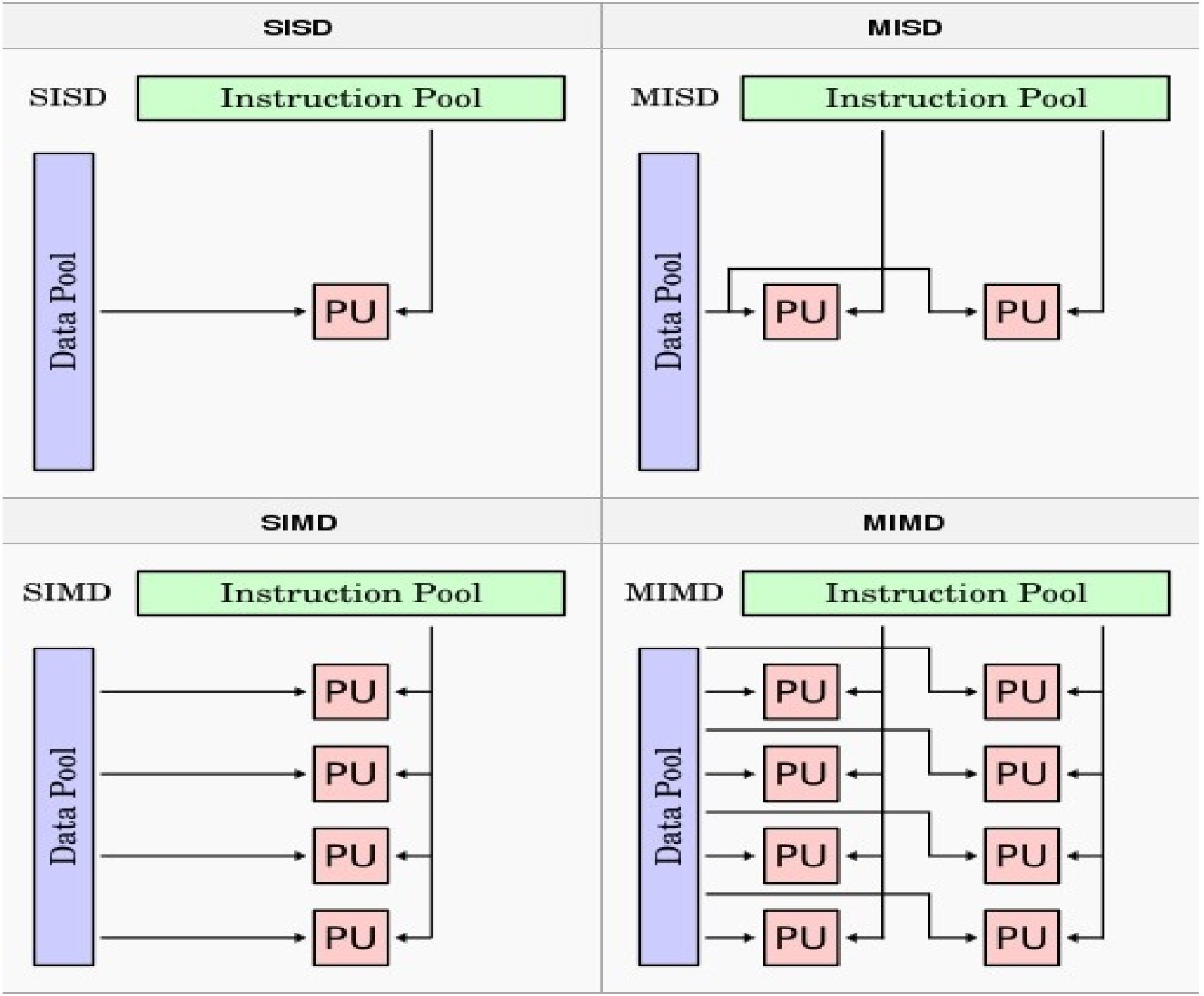
MIMD

- L’architettura di computer parallelo piu' diffusa
- I piu' moderni elaboratori paralleli sono disegnati secondo questa architettura



MIMD

- Multiple Istruction: ogni processore puo' eseguire un flusso di istruzioni diverso
- Multiple Data: ogni processore puo' lavorare su un flusso di dati diverso
- L’esecuzione puo' essere sincrona o asincrona
- Esempi:
  - molti degli attuali supercomputer, i cluster, …


SISD
SISD
Instruction Pool
Data Pool
PU
MISD
MISD
Instruction Pool
Data Pool
PU
PU
SIMD
SIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
MIMD
MIMD
Instruction Pool
Data Pool
PU
PU
PU
PU
PU
PU
PU
PU