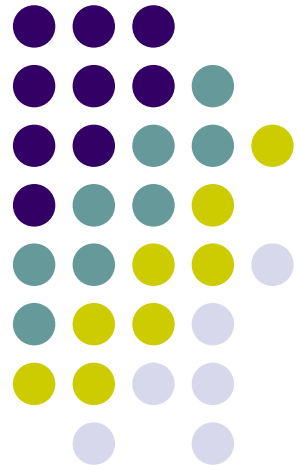
Università di Roma "La Sapienza"
Dipartimento di Matematica
15 – 31 gennaio 2007

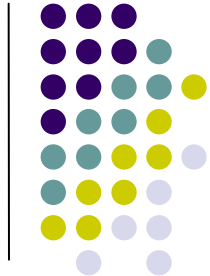
Calcolo parallelo

Claudia Truini



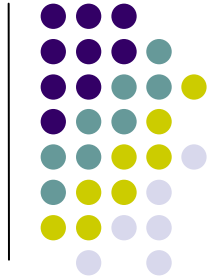
CASPUR

Le finalità del corso



- Descrivere le varie macchine parallele;
  - come funzionano
  - pregi e difetti
- Descrivere (qualitativamente) alcuni linguaggi paralleli:
  - OpenMP
  - SMS, MPI
- Fornire informazioni base per parallelizzare un codice

Argomenti del corso



Prima settimana:

- Introduzione al Calcolo Parallelo (C. Truini)

Seconda settimana:

- Programmazione *Shared Memory* (F. Bonaccorso)

Terza settimana:

- Programmazione *Distributed Memory* (V. Ruggiero)

Università di Roma "La Sapienza"
Dipartimento di Matematica
15 – 31 gennaio 2007

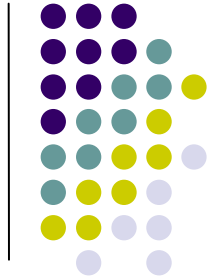
Introduzione al Calcolo parallelo

Prima parte

Claudia Truini
c.truini@caspur.it

CASPUR

Sommario



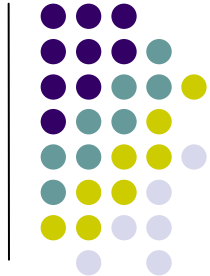
Prima parte:

- Motivazioni per il Calcolo Parallelo
- Classificazione delle architetture parallele
- Modelli di programmazione parallela
- Modelli di esecuzione parallela
- Misura delle prestazioni parallele

Seconda parte:

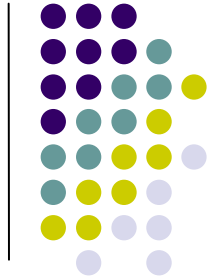
- Come costruire un programma parallelo

Cos'è il calcolo parallelo



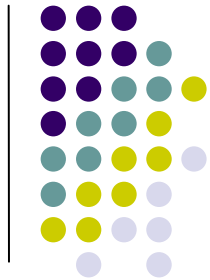
- È una evoluzione del calcolo seriale
- In generale, è l'uso simultaneo di più computer per risolvere un unico problema computazionale
- Per girare su più CPU, un problema è diviso in parti discrete che possono essere risolte *concorrentemente*
- Ogni parte è a sua volta divisa in una serie di istruzioni
- Le istruzioni di ogni parte sono eseguite contemporaneamente su CPU diverse

Il calcolo parallelo



- È l'emulazione di ciò che avviene quotidianamente intorno a noi: "molti eventi complessi e correlati che avvengono in contemporanea"
- È motivato dalla simulazione numerica di sistemi molto complessi, cioè "Grand Challenge Problems":
  - Modellazione del clima globale
  - Simulazioni di reazioni chimiche e nucleari
  - Studio del genoma umano
  - Studio delle proprietà di materiali complessi
  - Simulazione di attività geologica e sismica
  - Progettazione di veicoli più efficienti e sicuri

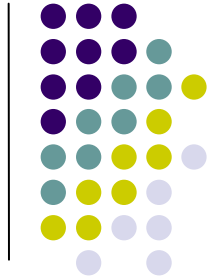
Applicazioni commerciali



- Attualmente diverse applicazioni commerciali sono un elemento trainante nello sviluppo di computer sempre più veloci
- Applicazioni che richiedo il processing di grandi moli di dati con algoritmi spesso molto sofisticati

  esempi:
  - Database paralleli e data mining
  - Esplorazione petrolifera
  - Motori di ricerca per il web
  - Diagnostica medica
  - Grafica avanzata e realtà virtuale
  - Broadcasting su network ed altre tecnologie multimediali
  - Ambienti di lavoro collaborativi virtuali

Perchè usare il calcolo parallelo

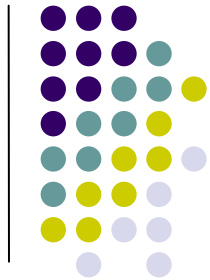


- Risolvere problemi più grandi
- Risparmiare tempo

Ma anche:

- Utilizzare un insieme di risorse locali
- Superare i vincoli di memoria
- Contenere i costi
  - N processori economici invece che 1 più costoso

Perchè usare il calcolo parallelo esempio:



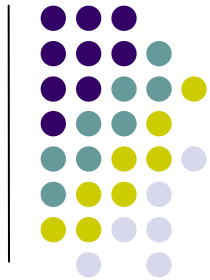
Previsioni del tempo su scala globale

L'atmosfera è modellata mediante una suddivisione in celle tridimensionali. Il calcolo di ogni cella è ripetuto molte volte per simularne l'andamento temporale.

Se supponiamo che:

- l'intera atmosfera sia suddivisa in celle di taglia 1miglio x 1miglio x 1miglio fino ad un'altezza dal suolo di 10 miglia (10 celle in altezza), avremo in totale ~ 5 x 10^8 celle.
- ogni calcolo richieda 2000 operazioni floating point, allora in un singolo time-step dovremo effettuare 10^12 operazioni floating point.

Perchè usare il calcolo parallelo esempio (2)



Previsioni del tempo su scala globale

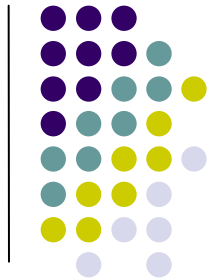
- se vogliamo simulare una previsione fino a 10 giorni usando un passo temporale di 10 minuti

- Un computer che lavora ad una potenza di 1 Gflops (10^9 operazioni floating point/sec) impiegherà ~ 20 giorni.

- Per terminare il calcolo in 20 minuti occorrerà un computer operante a 1.2 Tflops (1.2 x 10^12 operazioni floating point/sec).

Il più potente calcolatore esistente è capace di oltre 300 Tflops di picco (IBM Blue gene/L)

Terminologia



- Mega → 10^6
- Giga → 10^9
- Tera → 10^12
- Peta → 10^15

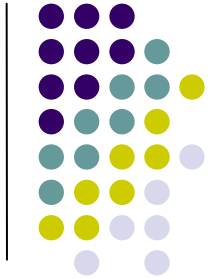
- singolo PC → 1 Gflops
- macchine parallele entry level → 10 Gflops
- macchine parallele livello medio → 100 Gflops
- macchine parallele livello alto → 1000 Gflops = 1 Tflops

- nota:

  1 Mega secondi = 1.000.000 s → 11 giorni

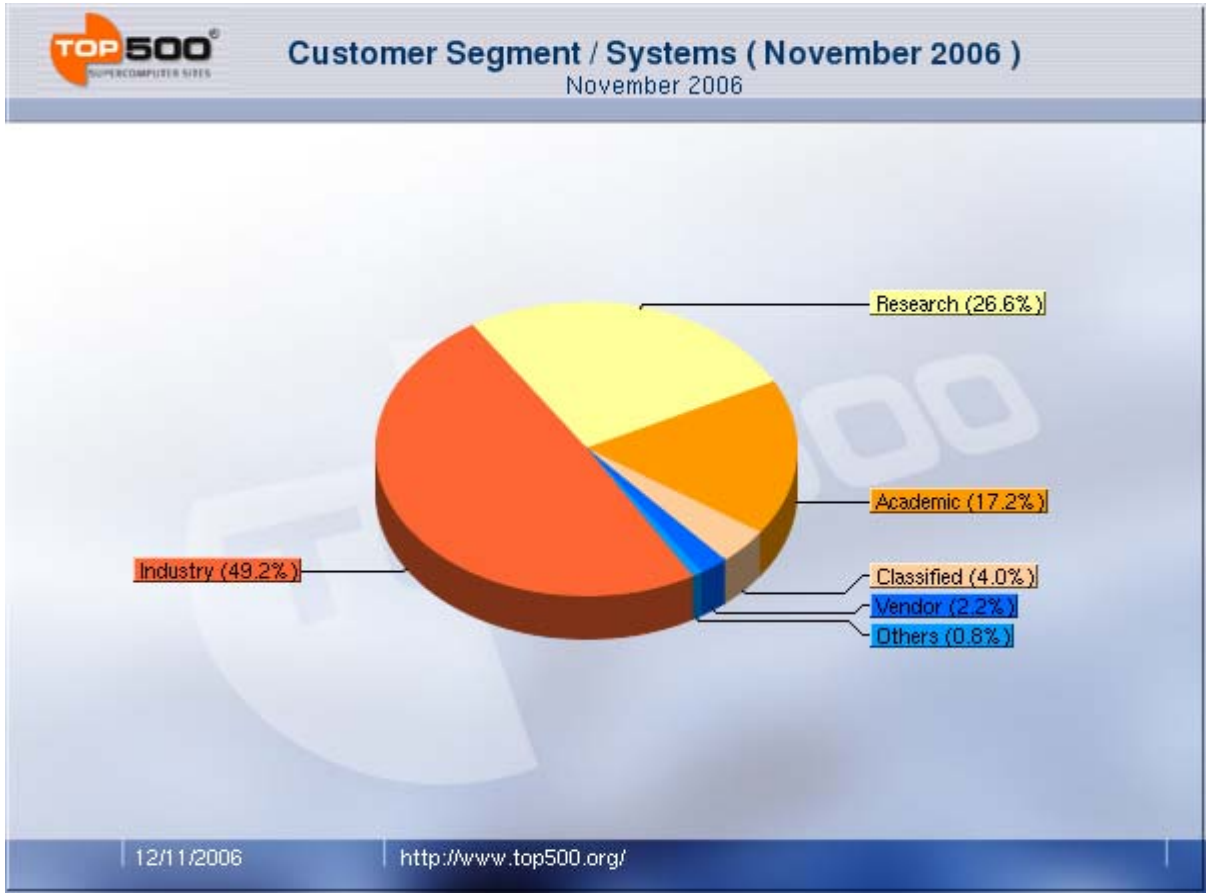
  1 Giga secondi = 1.000.000.000 s → 31 anni

Perchè usare il calcolo parallelo (2)

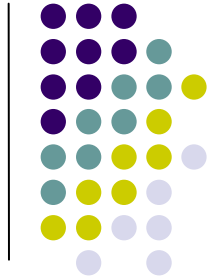


- Superare i limiti strutturali del calcolo seriale:
  - limiti fisici:
    - velocità di propagazione di un segnale
  - limiti tecnici:
    - clock estremamente alti causano alta dissipazione > 100 Watt
  - limiti economici
  - notare che:
    - già adesso i processori seriali presentano un parallelismo interno (funzionale): più pipeline indipendenti
    - già adesso un singolo chip può presentare più processori

Chi utilizza il calcolo parallelo



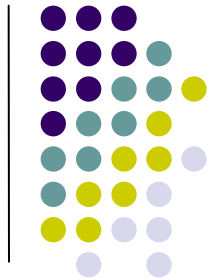
La top 500



http://www.top500.org

- Lista delle più grosse macchine parallele installate (omogenee)
- Ordinate rispetto un benchmark di risoluzione di un sistema lineare (linpack):
  - sustained performance > 50% (→ caso realistico?)
  - È un buon benchmark?
    - È rappresentativo del vostro problema?

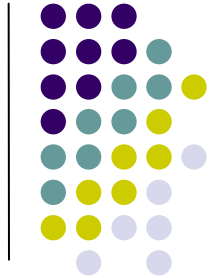
La top 500 (2)



- Le principali istallazioni (Novembre 2006):

N.	Nome	Vendor	CPU	N. Cpu	Peak (GF)	Sustained(GF)
1	BlueGene/L	IBM	powerPC@0.7 Ghz	131072	367000	280600
2	Red Storm	CRAY	Opteron@2,4 GHz	26544	127411	101400
3	BGW	IBM	powerPC@0.7 Ghz	40960	114688	91290
4	ASC Purple	IBM	power5@1.9 Ghz	12208	92781	75760
5	Marenostrum	IBM	powerPC@2.3 Ghz	10240	94208	62630
500	Blade Cluster	HP	Pentium4@3.0 Ghz	800	4896	2736

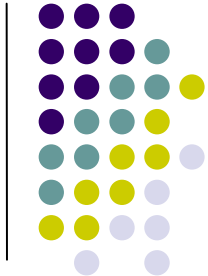
La top 500 (3)



- Solo 2 anni fa:
- le principali istallazioni (Novembre 2004):

N.	Nome	Vendor	CPU	N. Cpu	Peak (GF)	Sustained (GF)
1	Blue gene/L	IBM	powerPC@0.7 Ghz	32768	91750	70720
2	Columbia	SGI	Itanium2@1.5 Ghz	10160	60960	51870
3	Earth Simulator	NEC	SX6@0.5 Ghz	5120	40960	35860
4	Marenostrum	IBM	powerPC@2.2 Ghz	3564	31363	20530
5	Thunder	-	Itanium2@1.5 Ghz	4096	22938	19940
500	Superdome	HP	PA@0.875 Ghz	416	1456	850

Processori, memoria & reti

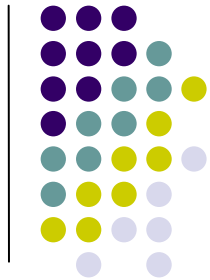


- Rispetto macchine seriali è importante l'interconnessione
  - … soprattutto per i cluster …

Nome	Vendor	Cpu	Cpu	Peak (GF)	Sustained (GF)
K.I.S.T.	IBM	Intel Xeon@2.4 Ghz	1024	4915	3067
TotalFinaElf	IBM	Intel Xeon@2.4 Ghz	1024	4915	1755

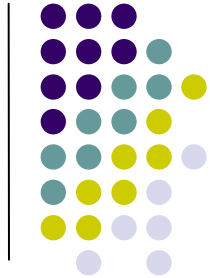
- Dov'è la differenza?
  - Myrinet
  - Gigabit Ethernet

I costi del calcolo parallelo



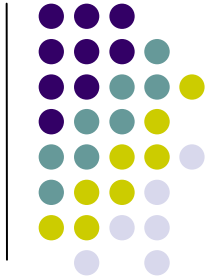
- Le macchine parallele sono:
  - in genere di complessa gestione
  - spesso costose
  - in genere voluminose
  - …
- Il calcolo parallelo implica una modifica “pesante” dei codici
  - codici paralleli non girano su macchine seriali
  - algoritmi intrinsecamente seriali
  - forte legame tra prestazioni ed architettura usata
  - …

Terminologia (2)



- Differenti classi di macchine parallele
  - **MPP** (**M**assive **P**arallel **P**rocessor):
    - molti processori collegati da rete proprietaria
  - **Cluster:**
    - molti processori collegati da rete commerciali (in genere handmade)
  - **Constellation:**
    - macchine parallele interconesse (e.g. cluster di SMP)
  - **COTS:**
    - **C**ommodity **o**ff **t**he **s**helves

Classificazione: Tassonomia di Flynn (1966)

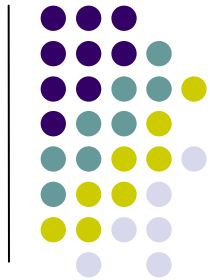


- Classificazione dei calcolatori in base a:
  - flusso di istruzioni (instruction stream)
  - flusso di dati (data stream)

SISD	SIMD
MISD	MIMD

- **SIMD**: **S**ingle **I**nstruction **M**ultiple **D**ata
  - ogni processore ha lo stesso programma che esegue in modo sincronizzato su differenti flussi di dati
- **MIMD**: **M**ultiple **I**nstruction **M**ultiple **D**ata
  - ogni processore può avere un proprio programma ed anche il flusso di dati è differente da processore a processore

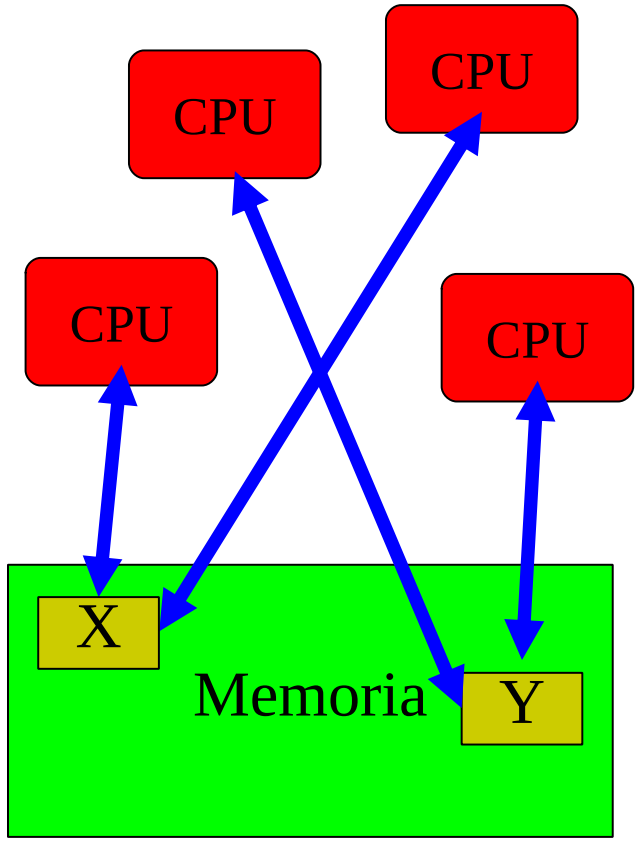
Classificazione: Indirizzamento della memoria



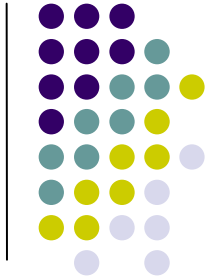
- È possibile indirizzare la memoria dall'indirizzo **0000…0000** fino a **1111…11111**, il che significa:
  - Per architetture a 32 bit → 2^32 = 4 GB
  - Per architetture a 64 bit → 2^64 = 16'000 PB
- È possibile classificare le macchine parallele rispetto a come gestiscono l'indirizzamento della memoria:
  - *Shared Memory*
  - *Distributed Memory*

Architetture Shared Memory

- Ogni processore può accedere tutta la memoria come uno spazio globale di indirizzi
- I processori operano indipendentemente, condividendo le stesse risorse di memoria
- Un cambiamento in una locazione di memoria dovuto ad un processore è visibile a tutti gli altri



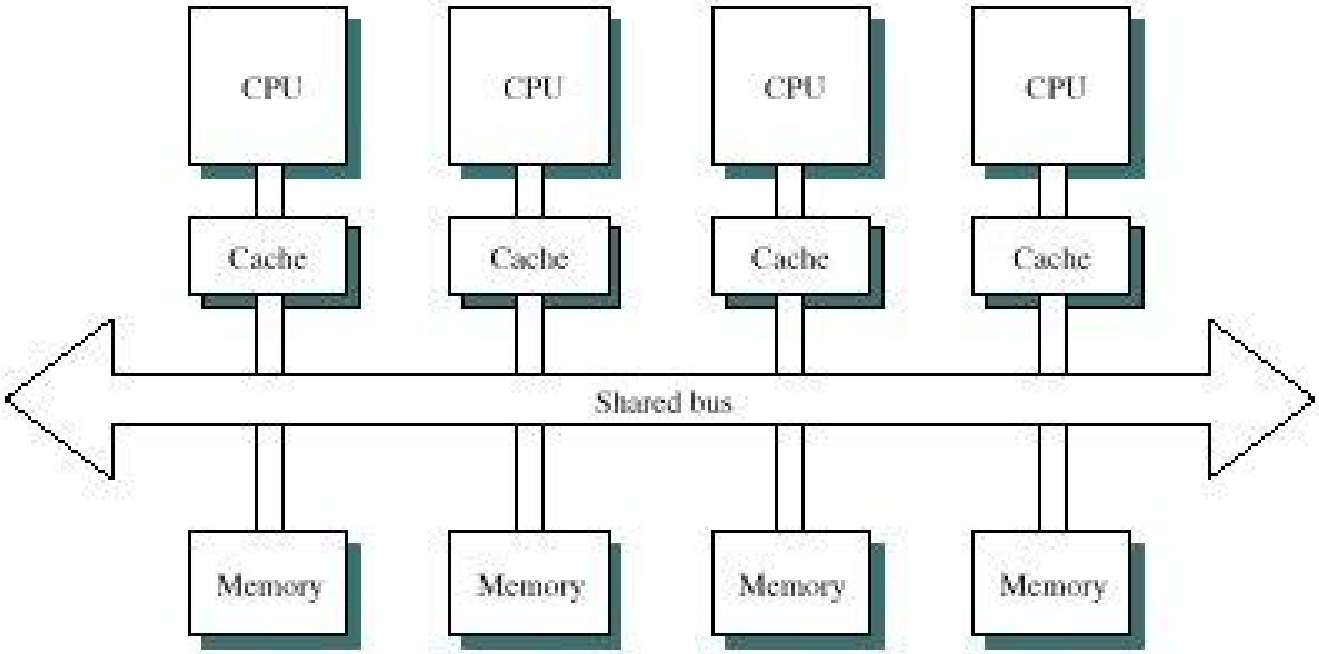
Architetture *Shared Memory* (2)



Si possono dividere in due classi:

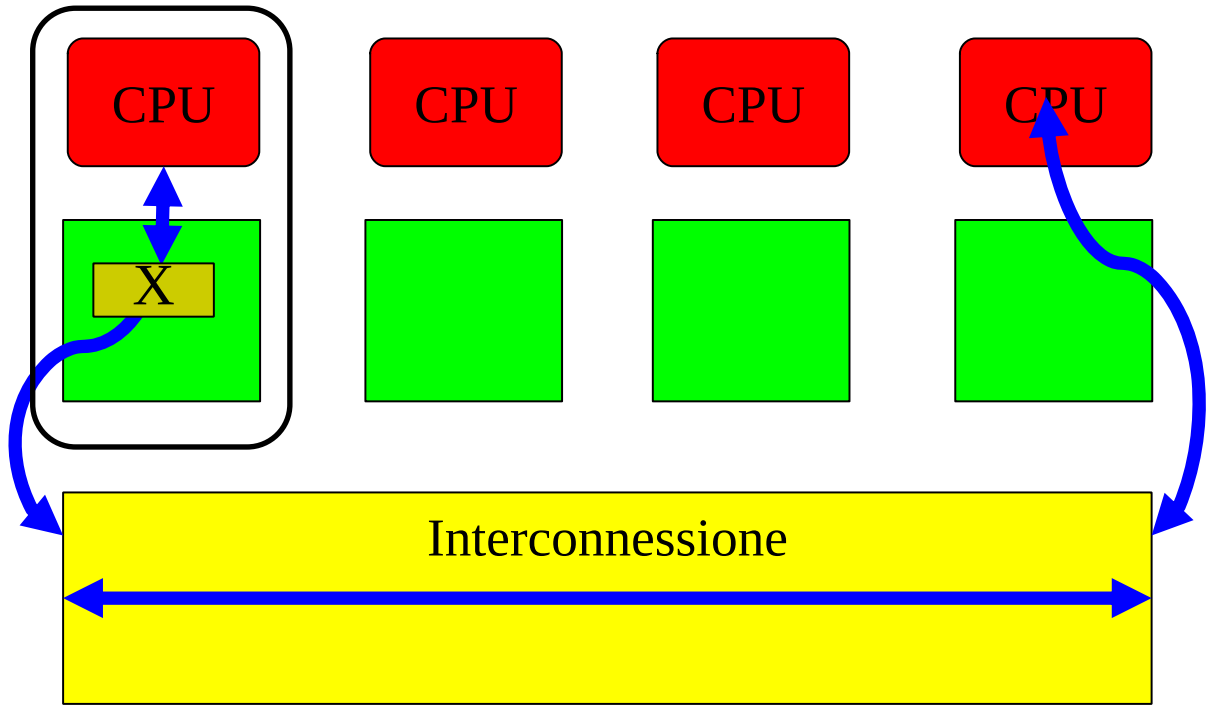
- ***U**niform **M**emory **A**ccess* (**UMA**)
  - Chiamati anche ***S**ymmetric **M**ulti**P**rocessor* (**SMP**)
  - Elaboratori costituiti da più processori identici
  - I tempi d'accesso a tutta la memoria sono uguali per ogni processore
- ***N**on-**U**niform **M**emory **A**ccess* (**NUMA**)
  - Sono realizzate attraverso il collegamento di due o più SMP
  - Ogni SMP può accedere alla memoria degli altri SMP
  - I processori non hanno lo stesso tempo d'accesso a tutta la memoria
  - Se è mantenuta la *cache coherency* si parla di architetture **CCNUMA** (***C**ache **C**oherent* **NUMA**)

Architetture *Shared Memory* (3) implementazione

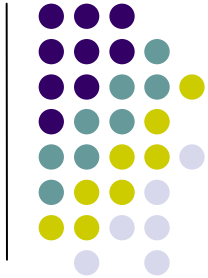


Architetture *Distributed Memory* (1)

Schema logico

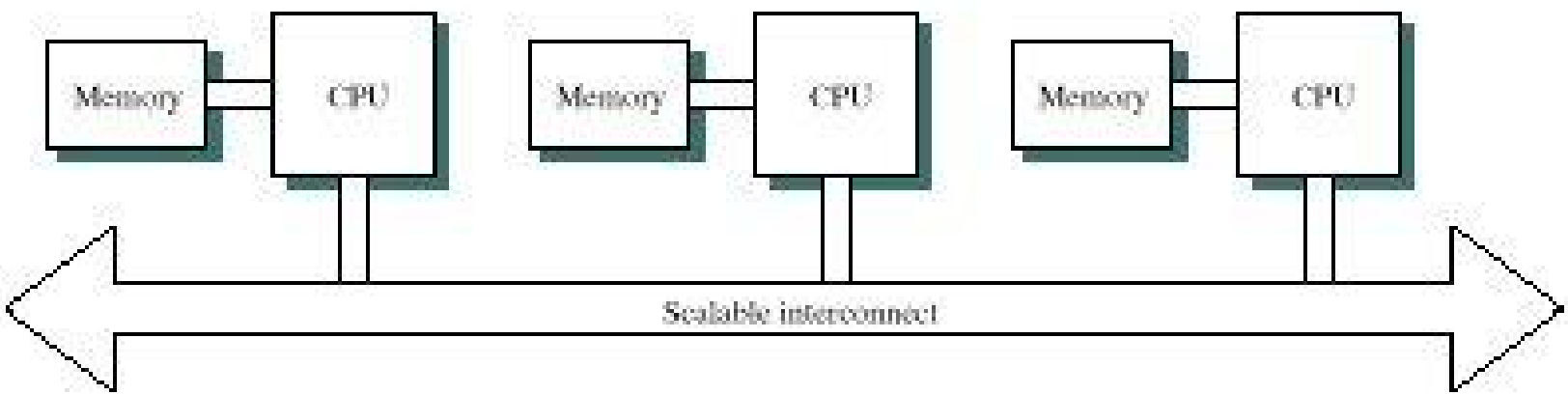


Architetture *Distributed Memory* (2)

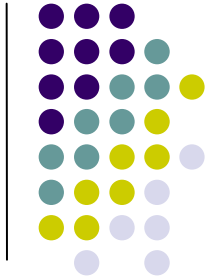


- Ogni processore dispone di un'area di memoria locale
- I processori possono scambiare dati solo attraverso il network di comunicazione
- Non è previsto un concetto di spazio di indirizzi globale tra i processori:
  - il programmatore deve definire in modo esplicito come e quando comunicare i dati e deve sincronizzare i task residenti su nodi diversi
- È importante l'infrastruttura di comunicazione utilizzata

Architetture *Distributed Memory* (3) implementazione

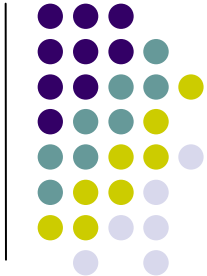


Architetture *Shared Memory* (4)



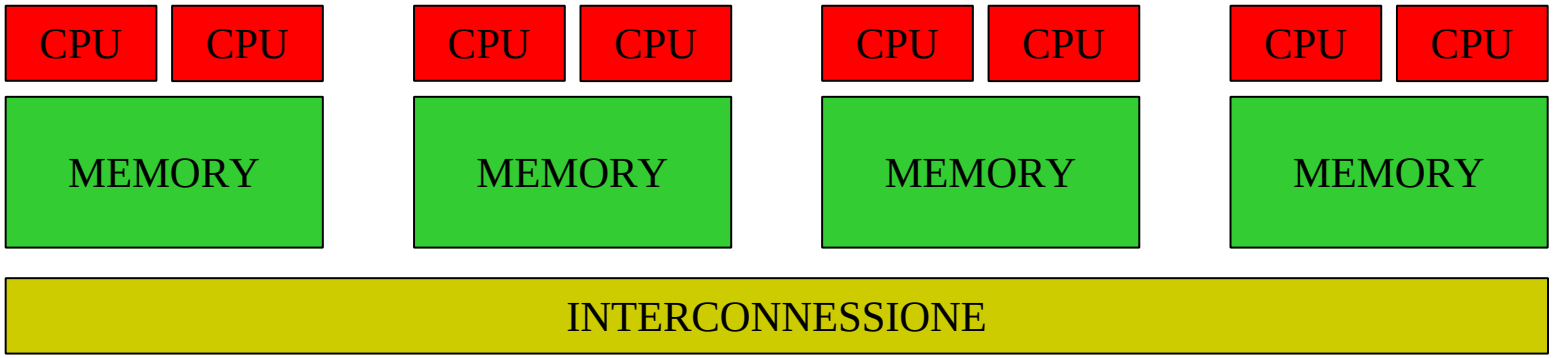
- Vantaggi
  - Programmazione più agevole rispetto agli accessi alla memoria
- Svantaggi
  - Scalabilità limitata
  - È compito del programmatore sincronizzare gli accessi alla memoria
  - Aumentano i costi all'aumentare dei processori

Architetture *Distributed Memory* (4)



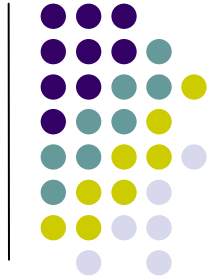
- Vantaggi
  - La quantità di memoria scala con il numero di processori
  - Ogni processore accede rapidamente alla propria memoria
  - Costi contenuti / più semplici da realizzare
- Svantaggi
  - Il programmatore è responsabile di tutti i dettagli legati alla comunicazione di dati tra processi
  - Potrebbe essere complesso “mappare” codici esistenti su un modello di memoria distribuita
  - È cruciale l’interconnessione

Architetture ibride: *Distributed Shared Memory*



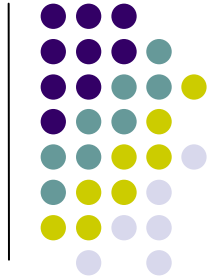
- Il singolo nodo è un sistema *Shared Memory*
- I nodi sono interconnessi tramite un network di comunicazione
- La comunicazione tra i processori avviene
  - tramite un'area di memoria condivisa (se sullo stesso nodo)
  - tramite il network (se su nodi diversi)

Macchine parallele al Caspur



- HP EV7 Shared Memory CCNUMA
- IBM Power5 Distributed Shared Memory
  - Ogni nodo è una macchina shared memory SMP
  - Rete di connessione: HPSwitch
- Cluster Opteron Distributed Shared Memory
  - Ogni nodo è una macchina shared memory NUMA
  - Rete di connessione: InfiniBand e Gigabit

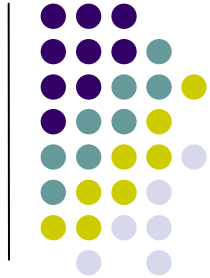
Una rete è definita da:



- **Topologia**: struttura della rete di interconnessione
  - ring
  - ipercubo
  - tree
- **Latenza (L)**: tempo necessario per “spedire” un messaggio vuoto (tempo di start-up)
- **Bandwidth (B):** velocità di trasferimento dei dati (Mb/sec.)

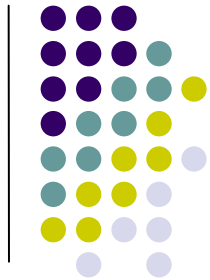
- **Tempo di comunicazione (T)** di un messaggio di N MB:
  - T = L + N/B

Interconnessione di rete



- Latenza vs. Bandwidth
  - grossa bandwidth e grande latenza:
    - poche comunicazioni ma molto grosse;
  - grossa bandwidth e bassa latenza:
    - il migliore dei mondi possibili ☺;
  - piccola bandwidth e grande latenza:
    - il peggiore dei mondi possibili ☹;
  - piccola bandwidth e bassa latenza:
    - molte comunicazioni molto piccole.

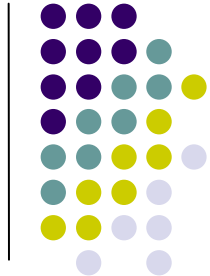
Interconnessione di rete



- Rispetto macchine seriali è estremamente importante (e costosa) l'interconnessione
- Dati scaricati da web

Nome	Latenza (μs)	Bandwidth (MB/s)	Costo scheda
Fast-Ethernet	100	12	< 10
Gigabit-Ethernet	100	125	> 10
Myrinet	7	250	> 100
QSWnet	5	360	> 100
InfiniBand	4	750	> 1000

Modelli di programmazione parallela



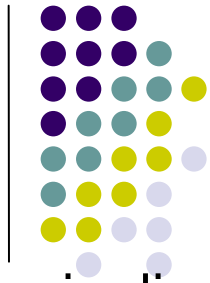
- Shared Memory
  - Multi Threaded
- Message Passing
- Data Parallel
- Ibridi

  In teoria questi modelli possono essere implementati su qualunque architettura parallelo.

  Nessuno è il migliore in assoluto.

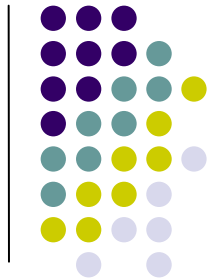
  Il modello migliore dipende dal problema che si deve affrontare.

Modello *Shared Memory*



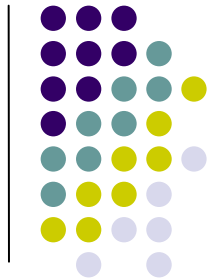
- Oltre alla propria memoria privata, i task condividono uno spazio di memoria da cui leggono e scrivono asincronamente
- È possibile controllare l'accesso alla memoria condivisa tra i processi
- Vantaggi:
  - Non richiede la programmazione esplicita della comunicazione di dati tra processi
  - È facile sviluppare un programma parallelo partendo in modo incrementale dalla versione seriale del codice
- Svantaggi:
  - Inadatto a problemi in cui i processi concorrenti siano "particolarmente interconnessi"
  - È complesso gestire la località dei dati al processo con conseguenti problemi di performance dell'applicazione

Modello *Multi Threaded*

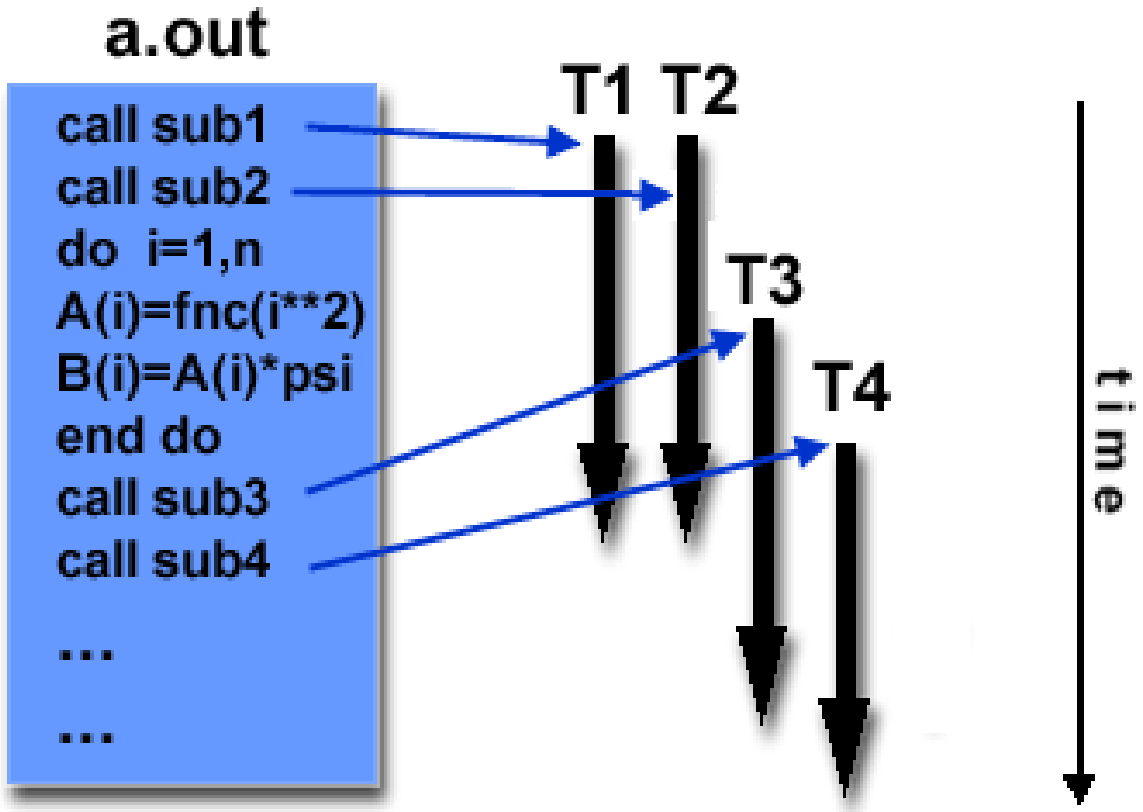


- Un processo singolo può creare alcuni *thread* (unità d'esecuzione), che il sistema operativo *schedula* in concorrenza (magari su CPU diverse)
- Ogni *thread*
  - condivide tutte le risorse associate al processo che lo ha generato (compreso il suo spazio di memoria)
  - ha una sua area di memoria privata
  - comunica con gli altri *thread* attraverso lo spazio di memoria globale
- Sono necessarie operazioni di sicronizzazione per evitare che, nello stesso istante, più di un thread cerchi di aggiornare una locazione di memoria globale
- Il modello *multi threaded* è usualmente associato ad architetture *shared memory*
- Possibili implementazioni:
  - Una libreria di primitive specifiche per la programmazione *multi thread*
  - Un insieme di direttive di compilazione da inserire nel codice seriale

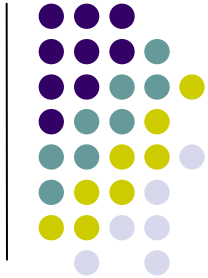
Modello *Multi Threaded* (2)



- Schema:

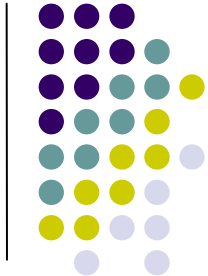


Implementazioni *Multi thread*



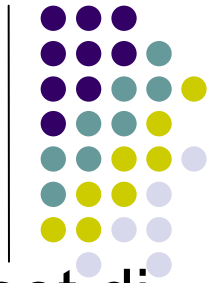
- **POSIX Threads (P**threads**)**
  - È una libreria utilizzabile in codici C
  - Molti *vendor* offrono una versione ottimizzata
  - La gestione del parallelismo è completamente esplicita (a cura del programmatore)
- **OpenMP**
  - Basato su direttive di compilazione
  - Può essere molto semplice da usare, perché consente di effettuare una “parallelizzazione incrementale” a partire dal codice seriale

Modello *Message Passing*



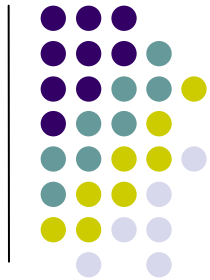
- Un insieme di processi usa la propria memoria locale durante la computazione
- L'insieme dei processi può risiedere sulla stessa macchina o su più macchine diverse interconnesse tramite un network di comunicazione
- La comunicazione inter-processo avviene attraverso lo scambio esplicito di messaggi
- Lo scambio dati è un'operazione cooperativa tra processi
- L'implementazione del modello di programmazione *message passing* richiede una libreria di funzioni da chiamare all'interno del proprio codice

Modello *Data Parallel*



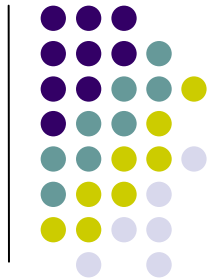
- Un set di *task* lavora collettivamente sullo stesso set di dati
- Ogni *task* lavora su una porzione diversa della stessa struttura dati
- Ogni *task* esegue la stessa operazione sulla propria partizione di dati
- Sulle architetture *Shared Memory* i *task* accedono alla struttura dati attraverso la memoria condivisa
- Sulle architetture *Distributed Memory* la struttura dati è divisa in *chunk* e ciascuno di essi risiede nella memoria locale al *task*
- Linguaggi con costrutti *data parallel*: **Fortran 95** e **HPF (H**igh **P**erformance **F**ortran)

Modello di programmazione parallela Ibrido



- Possono essere combinati insieme due o più modelli di programmazione parallela
- Esempio:
  - Combinazione, su architetture ibride, di un modello Shared Memory (OpenMP) con un modello Message Passing (MPI):
    - OpenMP all'interno del nodo
    - MPI all'esterno del nodo

Modelli d'esecuzione

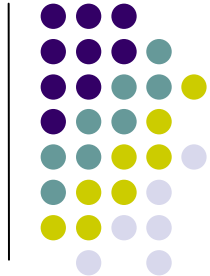


- **SPMD**: **S**ingle **P**rogram **M**ultiple **D**ata
  - Un solo programma è eseguito da più processi
  - Ogni processo opera su dati diversi
  - I processi possono eseguire la stessa istruzione o istruzioni diverse
  - Il generico processo può eseguire solo una parte del codice e/o può operare solo su un sottoinsieme di dati

- **MPMD**: **M**ultiple **P**rogram **M**ultiple **D**ata
  - Un'applicazione MPMD è costituita da diversi programmi
  - Ogni processo esegue un programma
  - Un programma può essere eseguito da più di un processo
  - I dati su cui operano i singoli programmi possono essere diversi

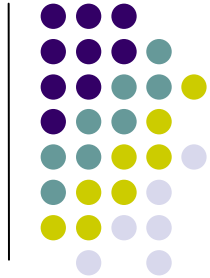
- Entrambi possono essere realizzati con diversi modelli di programmazione

Primi passi



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

Esempi

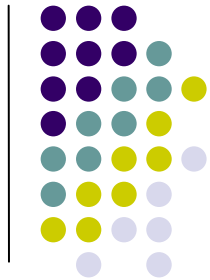


- Problema parallelizzabile

  Calcolare l'energia potenziale di migliaia di conformazioni indipendenti di una molecola.

  Ogni conformazione della molecola è determinabile indipendentemente dalle altre => possono essere calcolate in parallelo.

Esempi (2)



- Problema non parallelizzabile

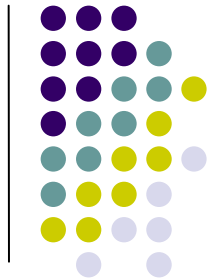
  Calcolare la successione di Fibonacci (0,1,1,2,3,5,8,13,21,34, ...) con la formula

  F(n) = F(n - 1) + F(n - 2) dove F(0)=0 e F(1)=1

  I calcoli sono dipendenti: per calcolare il valore n devo conoscere i due valori precedenti n-1 e n-2

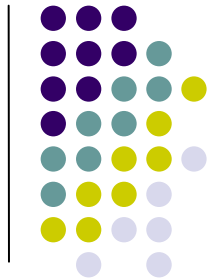
  Questi 3 valori non possono essere calcolati indipendentemente => non possono essere calcolati in parallelo.

Primi passi (2)



- Identificare le parti maggiormente pesanti e su quelle focalizzare l'attenzione
- Identificare i possibili "colli di bottiglia": provare a eliminarli o ridurli
- Identificare inibitori al parallelismo (ad. es. dipendenza dei dati)
- Studiare differenti algoritmi

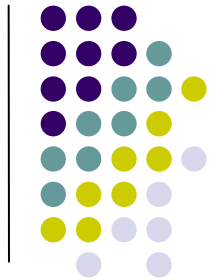
Misurare le prestazioni (1)



- **Speedup:** aumento della velocità (= diminuzione del tempo) dovuto all'uso di n processori:
  - S(n) = T(1)/T(n)
    - T(1) = tempo totale seriale
    - T(n) = tempo totale usando n processori

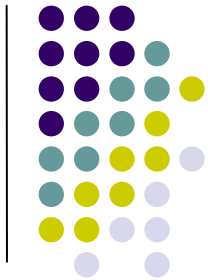
    - S(n) = n → speed-up lineare (caso ideale)
    - S(n) < n → speed-up di una caso reale
    - S(n) > n → speed-up superlineare (effetti di cache)

Misurare le prestazioni (2)



- **Efficienza**: effettivo sfruttamento della macchina parallela:
  - E(n) = S(n)/n = T(1)/(T(n)*n)
  - E(n) = 1 → caso ideale
  - E(n) < 1 → caso reale
  - E(n) << 1 → ci sono problemi …
- **Scalabilità:** capacità di essere efficiente su una macchina parallela: aumento le dimensione del problema proporzionalmente al numero di processori usati.

Legge di Amdhal (1)

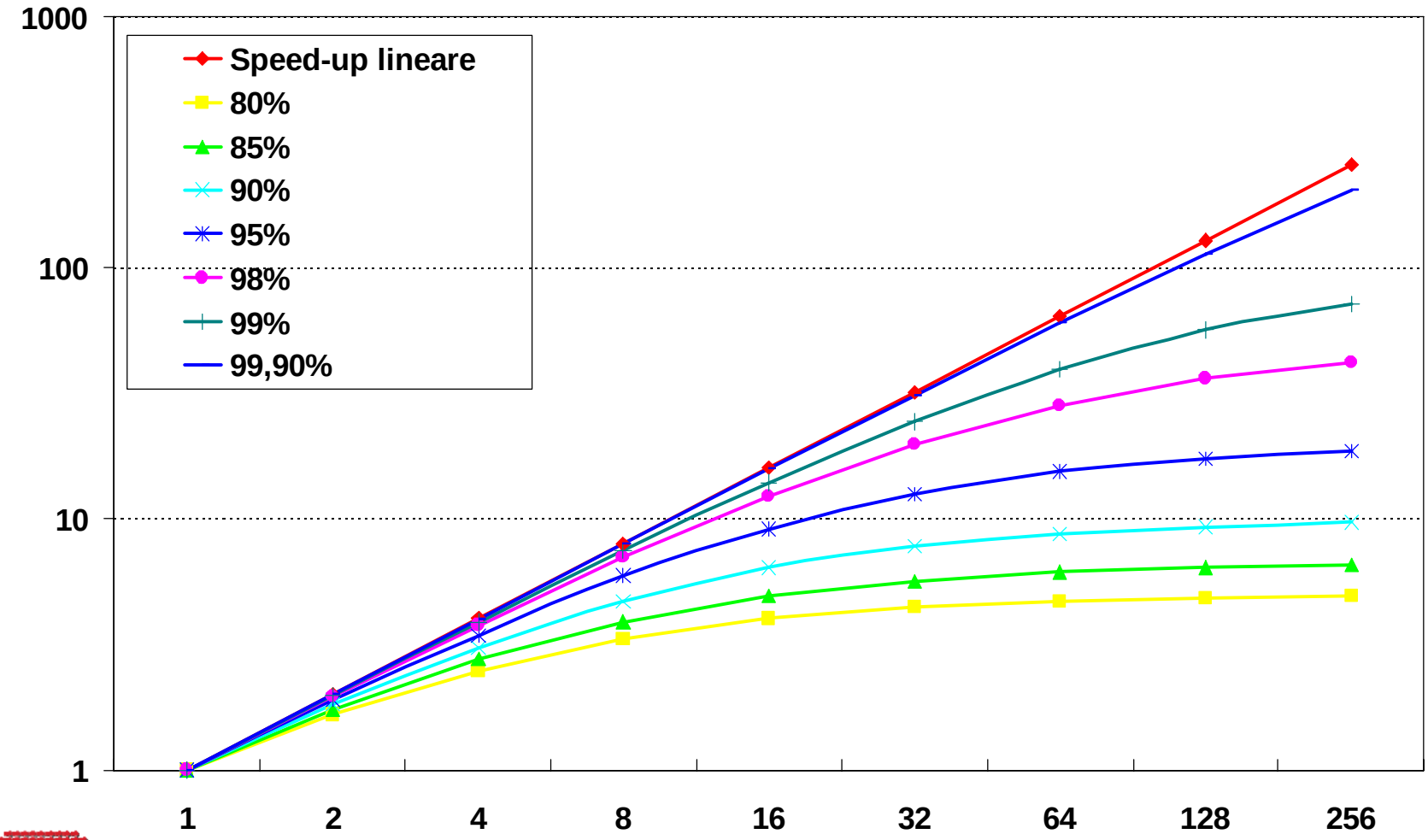


- Quale è lo speed-up massimo raggiungibile?
- Tempo seriale: **T = F + P**
  - F = parte seriale (non parallelizzata/parallelizzabile)
  - P = parte parallela
- Tempo con n processori: **T(n) = F + P/n**
- Speed-up su n processori:

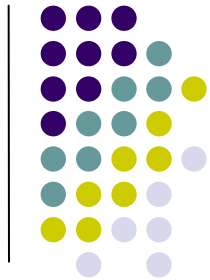
  S(n) = T(1)/T(n) = **(F + P)/(F+P/n)**

  - Per n >> 1 **S(n) = (F + P)/F**
  - esiste un limite asintotico allo speed-up!!!!!

Legge di Amdhal (2)
Speedup



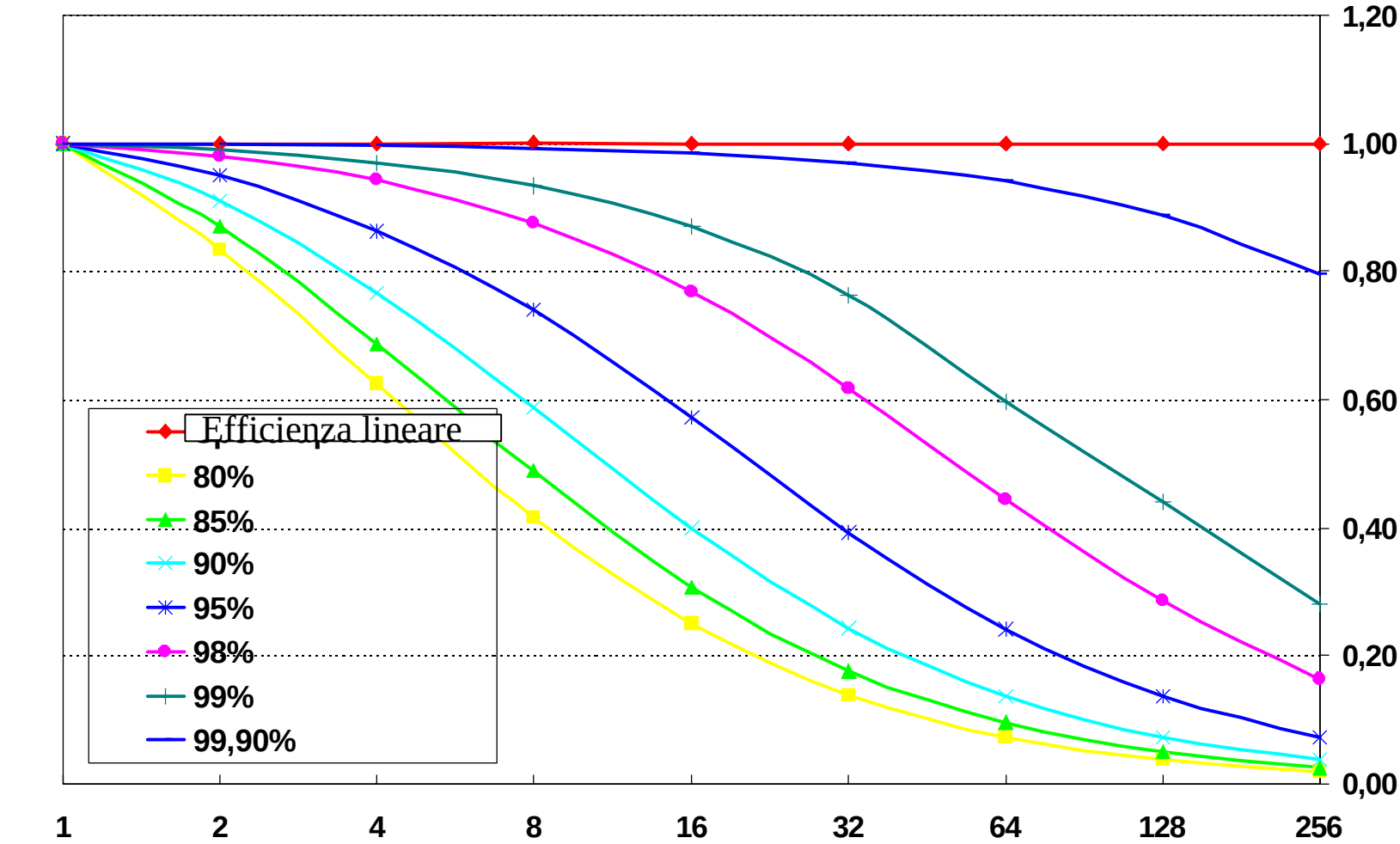
Legge di Amdhal (3) Speedup



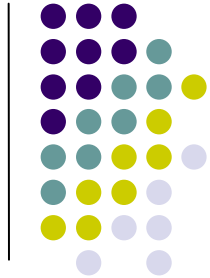
- Massimo speed-up raggiungibile in funzione della percentuale della parte parallele

	1	2	4	8	16	32	64	128	256
80%	1.0	1.7	2.5	3.3	4.0	4.4	4.7	4.8	4.9
85%	1.0	1.7	2.8	3.9	4.9	5.7	6.1	6.4	6.5
90%	1.0	1.8	3.1	4.7	6.4	7.8	8.8	9.3	9.7
95%	1.0	1.9	3.5	5.9	9.1	12.6	15.4	17.4	18.6
98%	1.0	2.0	3.8	7.0	12.3	19.8	28.3	36.2	42.0
99%	1.0	2.0	3.9	7.5	13.9	24.4	39.3	56.4	72.1
99.9%	1.0	2.0	4.0	7.9	15.8	31.0	60.2	113.6	204.0
Ideale	1.0	2.0	4.0	8.0	16.0	32.0	64.0	128.0	256.0

Legge di Amdhal (4)
Efficienza

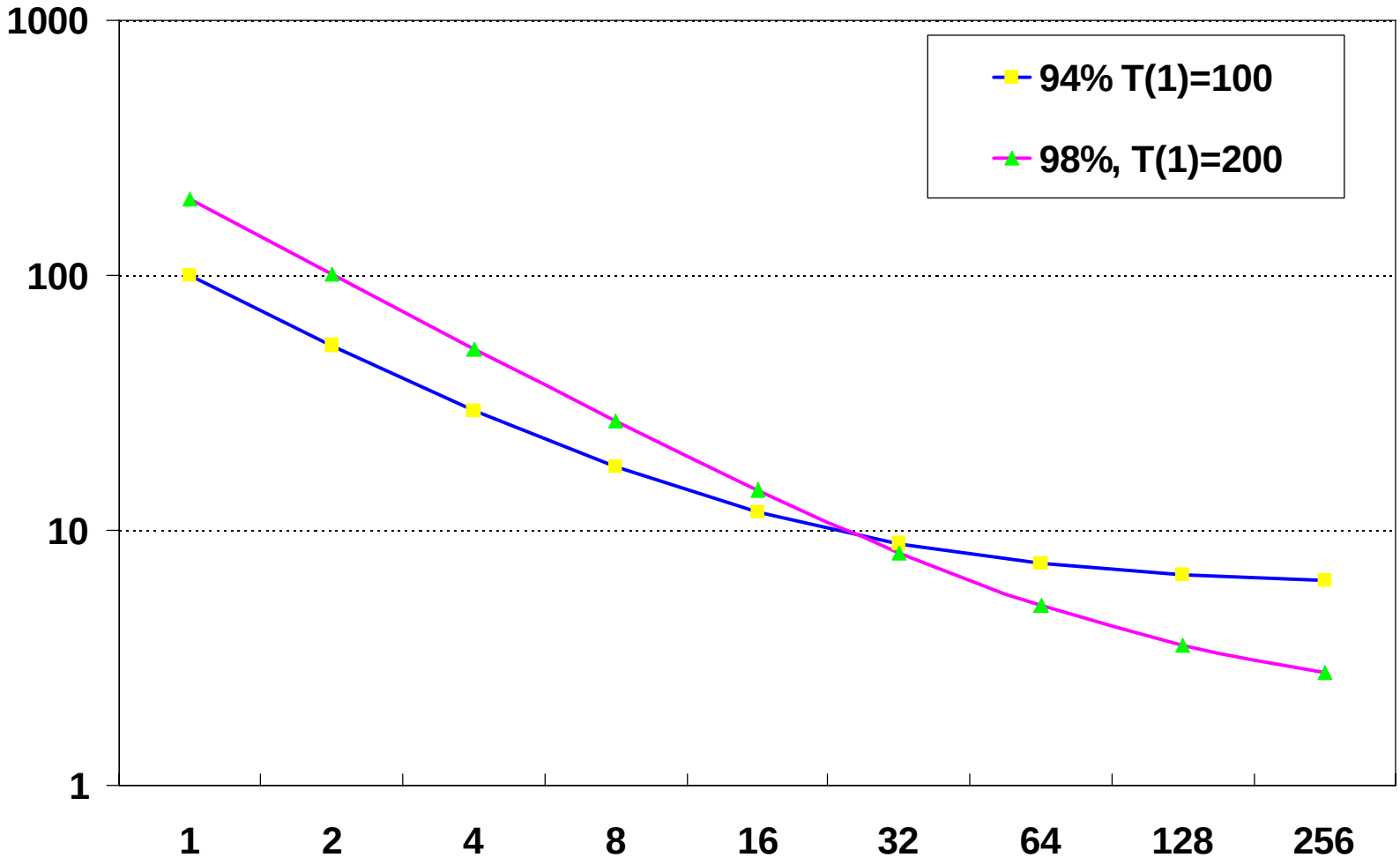


Legge di Amdhal (5)



- Introdotta per spiegare l'inutilità delle macchine parallele (Amdhal lavorava all'IBM)
- Sovrastima la scalabilità: non tiene conto delle sincronizzazioni e delle risorse contese
- Inoltre:
  - la parte sequenziale può dipendere dalle dimensioni del problema;
  - il caso seriale è improponibile (e.g. troppa memoria o troppo tempo);
  - e se il codice parallelo è differente da quello seriale?

Legge di Amdhal (6)



Università di Roma "La Sapienza"
Dipartimento di Matematica
15 – 31 gennaio 2007

Introduzione al Calcolo parallelo

Fine prima parte

Claudia Truini

CASPUR

c.truini@caspur.it

Università di Roma "La Sapienza"
Dipartimento di Matematica
15 – 31 gennaio 2007

Introduzione al Calcolo parallelo

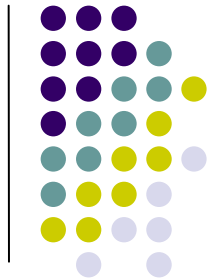
Seconda parte

Claudia Truini

CASPUR

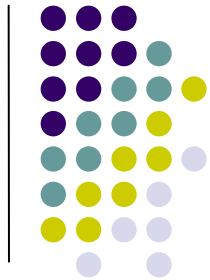
c.truini@caspur.it

Sommario



- Prima Parte
  - Motivazioni per il Calcolo Parallelo
  - Classificazione delle architetture parallele
  - Modelli di programmazione parallela
  - Modelli di esecuzione parallela
  - Misura delle prestazioni parallele
- Seconda Parte: Metodi per costruire un programma parallelo
  - Tecniche di Decomposizione
  - Comunicazioni / sincronizzazioni
  - Bilanciamento del carico
  - Esempi

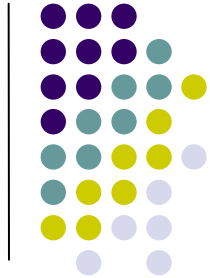
Tecniche di Partizionamento



- Si divide il problema in n parti, ciascuna parte viene assegnata ad un task differente

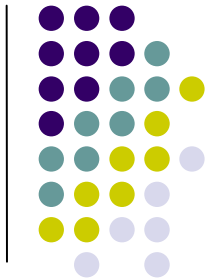
- Distribuzione del lavoro tra più soggetti:
  - tutti eseguono le stesse operazioni, ma su un sottoinsieme di dati
  - → partizionamento dei dati
- Distribuzione delle funzioni tra più soggetti:
  - non tutti eseguono le stesse operazioni
  - → decomposizione funzionale

Partizionamento dei dati

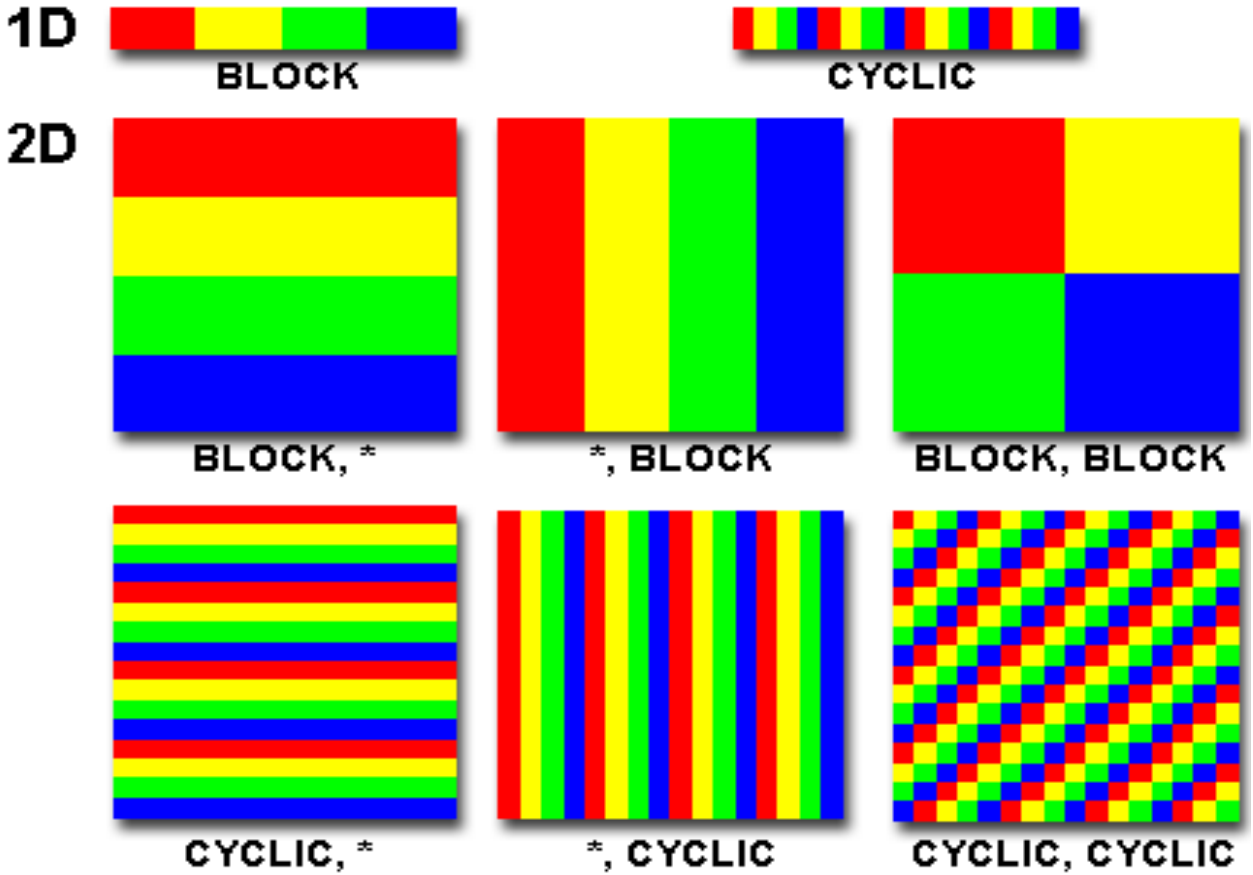


- insieme (ordinato) di dati da elaborare tutti allo stesso modo.
- decompongo i dati associati al problema
- ogni task esegue le stesse operazioni su una porzione dei dati (→ SPMD).
- Pro & Contro:
  - ☺ scalabile con la quantità di dati
  - ☹ vantaggiosa solo per casi sufficientemente grandi

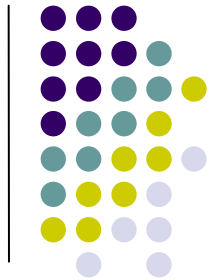
Partizionamento dei dati: esempio



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

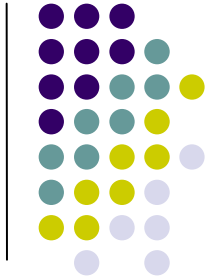


Decomposizione funzionale

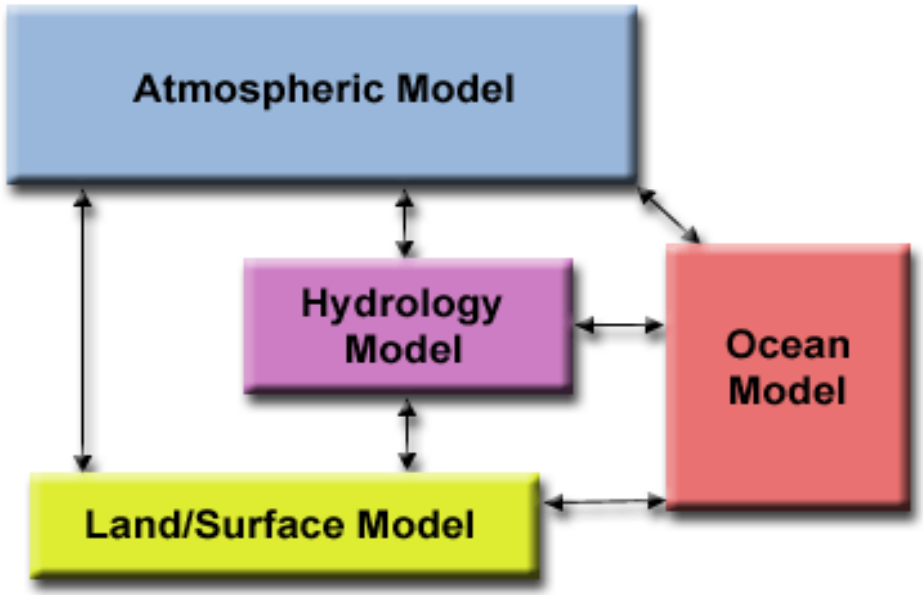


- insieme di elaborazioni differenti ed indipendenti
- decompongo il problema in base al lavoro che deve essere svolto
- ogni task prende in carico una particolare elaborazione (→MPMD).
- Pro & Contro:
  - ☺ scalabile con il numero di elaborazioni indipendenti
  - ☹ vantaggiosa solo per elaborazioni sufficientemente complesse

Decomposizione funzionale: esempio

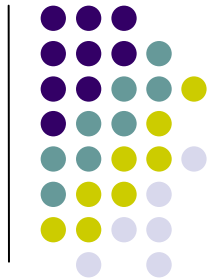


- Modello climatico



Ogni componente del modello puo essere considerata come un task separato. Le frecce rappresentano gli scambi di dati tra le componenti durante il calcolo.

Operazione di riduzione/scattering

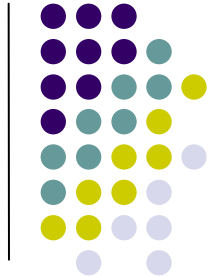


- Mettere tutto assieme / distribuire
- Somma / distribuzione dei vari risultati parziali in un risultato globale (e.g. energia totale)
- necessita di sincronizzazione e barriere tra i task

Pro & Contro:

- ☹ overhead dovuto al parallelo
- ☹ complicata con molti processori (> 100)

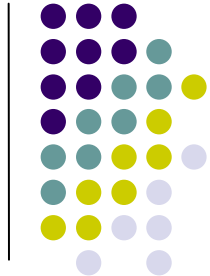
Comunicazioni



- Sono necessarie?
  - dipende dal problema
    - no, se il problema è “imbarazzantemente parallelo”
    - si, se il problema richiede dati di altri task
- Quanto comunicare?
  - dipende dal problema e da come viene partizionato

Tutte le comunicazioni implicano un overhead

Calcolo Parallelo *Imbarazzante*



- Si parla di un calcolo che può essere *in modo ovvio* diviso in un numero di parti completamente indipendenti, ognuna delle quali può essere eseguita da un task separato.
- Non vi sono comunicazioni (o quasi) tra task

Calcolo Parallelo *Imbarazzante*: Esempio

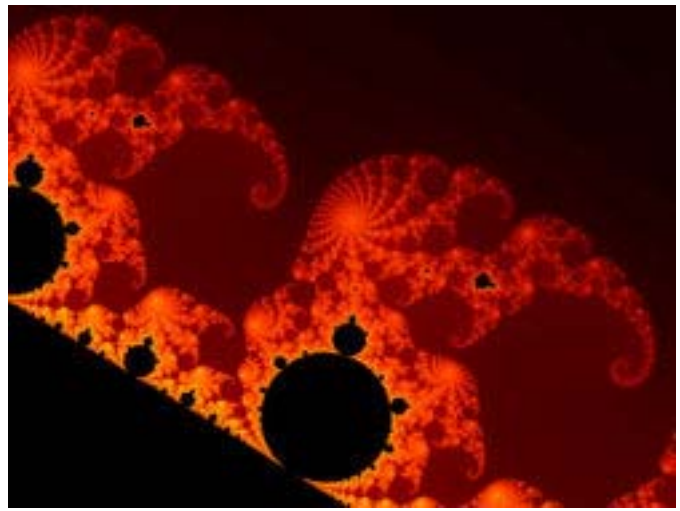
Insieme di Mandelbrot

- è un frattale
- è l'insieme dei numeri complessi tale per cui non è divergente la successione:

$$z_{n+1} = z_n^2 + c$$

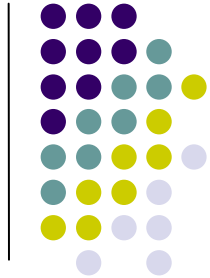
dove $z_0 = 0$, **z e c** sono numeri complessi, c fornisce la posizione del punto nel piano complesso

- La successione diverge se



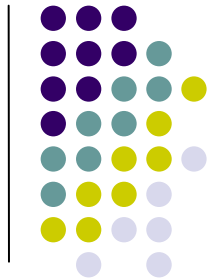
$$|z_n| = \sqrt{a^2 + b^2} > 2$$

Insieme di Mandelbrot: Parallelizzazione



- Non ho dipendenza dei dati
- Partizionamento dei dati
- Assegnazione **statica** del carico computazionale
  - Si divide la regione in n parti, ognuna viene assegnata e calcolata da un processore differente
  - Non è molto efficiente, in ogni punto varia il numero di iterazioni per raggiungere la convergenza e quindi varia il tempo
- Assegnazione **dinamica** del carico computazionale
  - Divido la regione in tante parti, ogni task prenderà in carico una parte
  - Quando un task termina una parte, ne richiede una nuova

Calcolo con comunicazioni: esempio

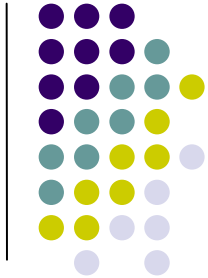


- Equazione di Laplace (1)
  - Dominio cartesiano (2D)
  - **n*m** punti
  - variabile **T(i,j),** fissata al contorno
    - procedura ricorsiva per calcolare **T(i,j)**

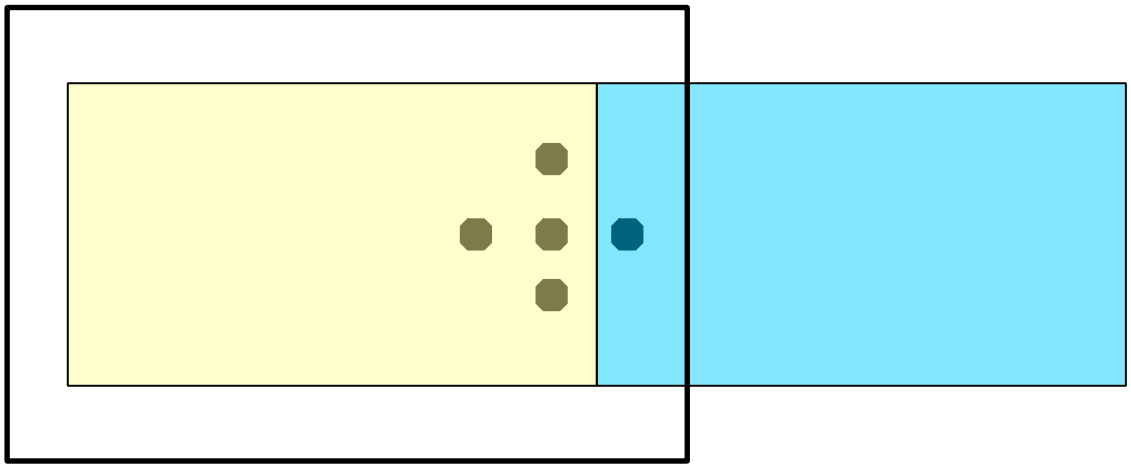
$$T_{n+1}(i,j) = \frac{T_n(i-1,j) + T_n(i,j-1) + T_n(i+1,j) + T_n(i,j+1)}{4}$$

    - stencil a quattro punti

Equazione di Laplace (2)

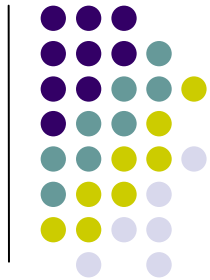


- Per mantenere il risultato corretto:
  - ogni blocco deve avere informazioni della frontiera



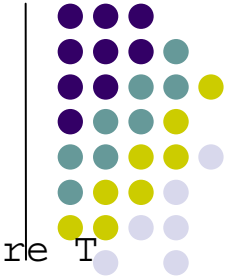
  - dato un blocco n*n devo propagare una “cornice” di 4*(n+1) elementi → overhead

Equazione di Laplace : Soluzione parallela



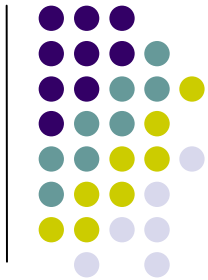
- Implemento un modello SPMD
- Partiziono il dominio e distribuisco T(i,j) a tutti i task
- Il calcolo della soluzione nei punti interni del dominio non dipenderà da valori esterni al task
- Il calcolo della soluzione nei punti del dominio vicini al bordo dipenderà da valori dei task vicini, quindi necessito di comunicazioni
- Il processo master spedisce le informazioni iniziali ai task, controlla la convergenza e collega i risultati
- I vari task calcolano la soluzione, comunicano quanto necessario con i task vicini

Equazione di Laplace:Pseudo codice

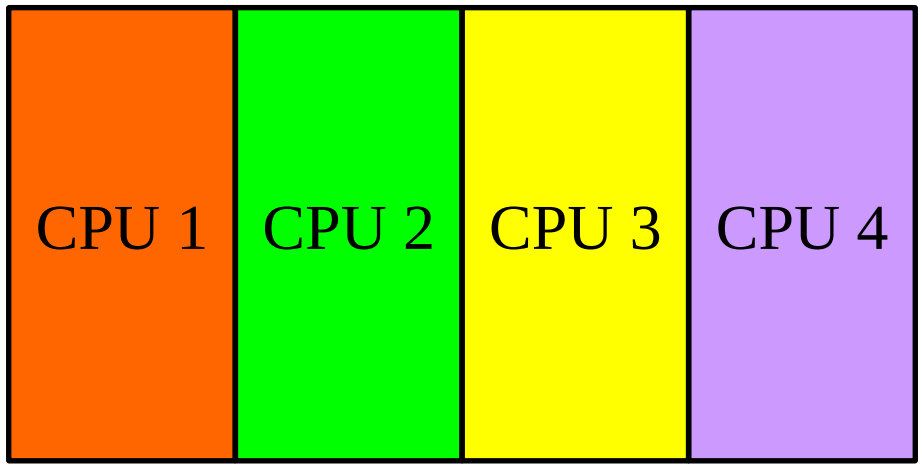


- Determino se sono Master o Slave
- If Master
  - Spedisco ad ogni Slave informazioni iniziali e il sottovettore T
  - do while in tutti gli Slave converge=false
    gather da tutti gli Slave i dati di convergenza
    broadcast a tutti gli Slave un segnale di convergenza
  - end do
  - Ricevo i risultati da ogni Slave e Output
- Else (If Slave)
  - Ricevo dal Master informazioni iniziali e il sottovettore T
  - do while converge=false
    aggiorno il contatore del tempo
    spedisco agli Slave vicini i valori di T nel bordo
    ricevo dagli Slave vicini i loro valori di T nel bordo
    calcolo T(i,j) nei miei nodi
    verifico criterio di convergenza nei miei nodi
    spedisco al Master il mio dato di convergenza
    ricevo dal master il segnale di convergenza
  - enddo
  - Spedisco i risultati al Master
- Endif

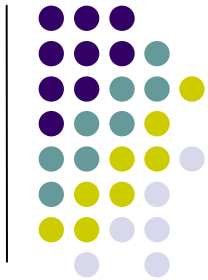
Equazione di Laplace (3)



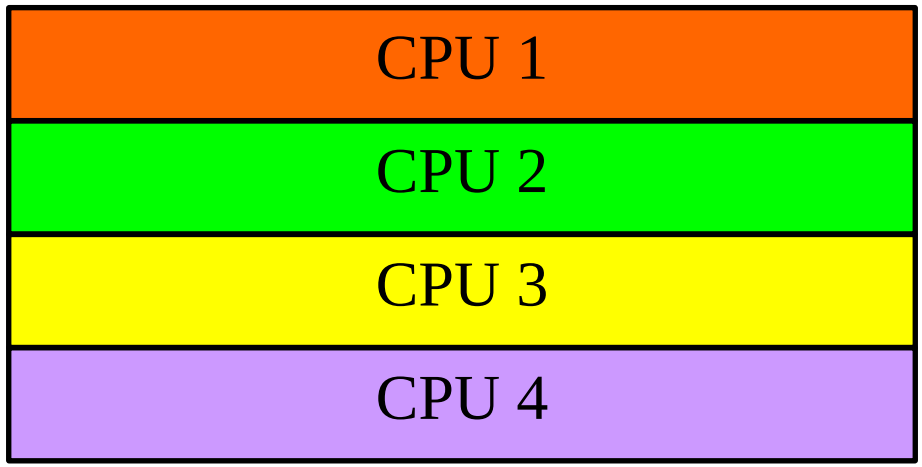
- Divisione su 4 task:
  - verticale
  - calcolo/comunicazioni = m*n / 6*m = n / 6



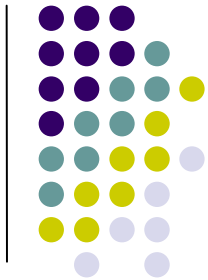
Equazione di Laplace (4)



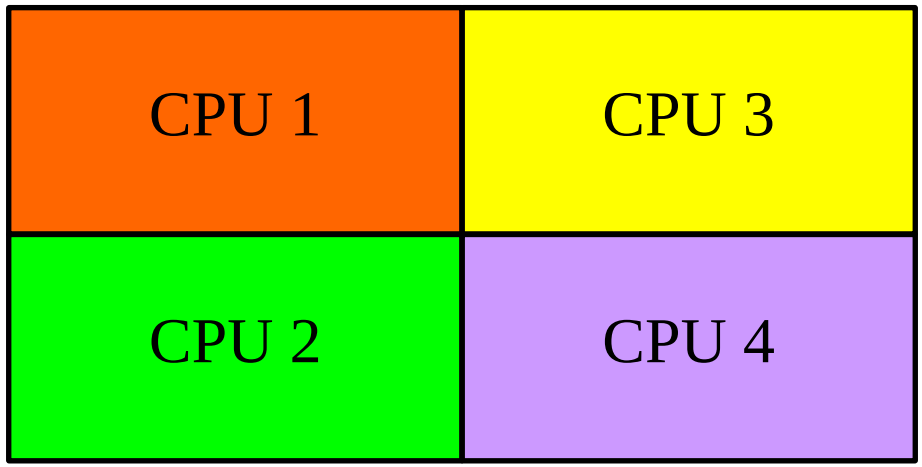
- Divisione su 4 task:
  - orizzontale
  - calcolo/comunicazioni = m*n / 6*n = m / 6



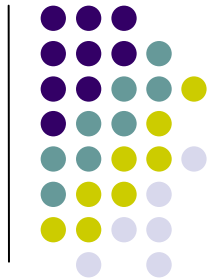
Equazione di Laplace (5)



- Divisione su 4 task:
  - blocchi
  - calcolo/comunicazioni = m*n / (4*(n/2)+4*(m/2))



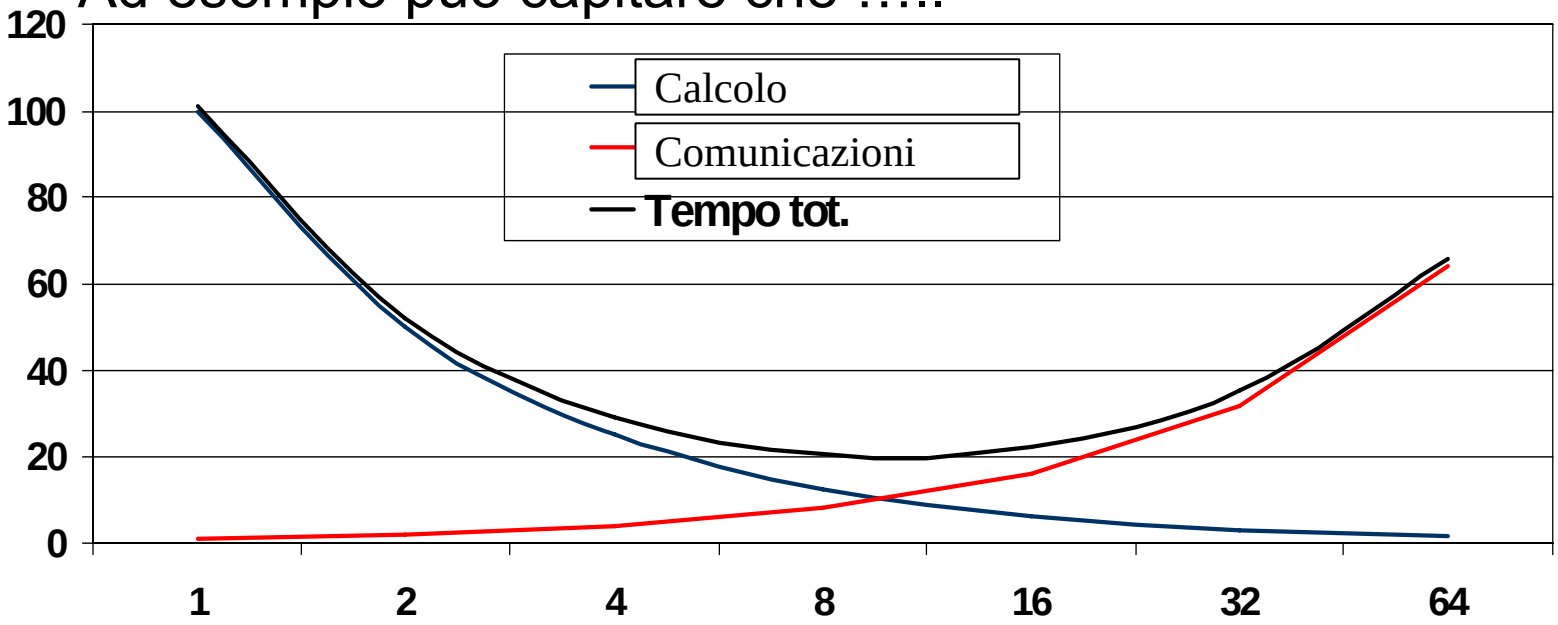
Equazione di Laplace (6)



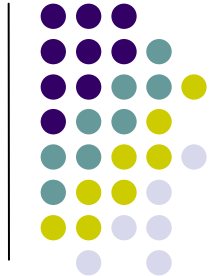
- Divisione su p task:
  - trascuro boundary conditions
  - rapporto calcolo/comunicazione per ogni processore
    - verticale = **`(m*n/p)/ 2*m`**
    - orizzontale = **`(m*n/p)/ 2*n`**
    - blocchi = **`(m*n/p)/2*(m/sqrt(p)+n/sqrt(p))`**
  - come scala all'aumentare di p?
    - ☺ rapporto >> 1
    - ☹ rapporto ≈ 1
  - Qual'è il migliore?
    - come accedo ai dati (stride unitario?)
    - verticale/orizzontale: pochi messaggi ma grandi
    - blocchi: più messaggi ma più piccoli
    - latenza v.s. bandwidth

Equazione di Laplace : Comunicazione vs. calcolo

- Equazione del calore (decomposizione a blocchi)
- Comunicazione $\propto$ p
- Calcolo $\propto$ 1/p
- Ad esempio può capitare che …..

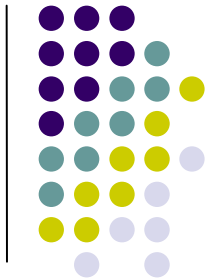


Come si parallelizza un codice?



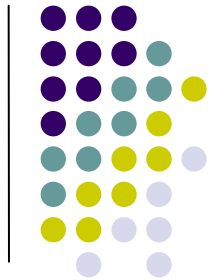
- Inserimento di direttive (e.g. OpenMP, SMS)
  - ☺ permette di mantenere un solo codice
  - ☹ dipende fortemente dal compilatore
- Uso di chiamate di librerie (e.g. MPI)
  - ☹ codice non più seriale
  - ☺ meno dipendente dal compilatore
- Attenzione:
  - Non è detto che il codice seriale sia parallelizzabile
  - Non è detto che il codice seriale parallelizzato sia il più efficiente

Come si parallelizza un codice?



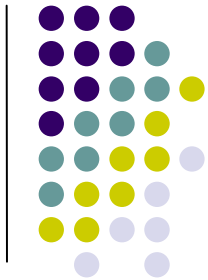
- Stare attenti a:
  - indipendenza dei dati
  - ordine delle operazioni
  - ordine (ed efficienza) di accesso in memoria
  - passaggio dati/informazioni tra task
  - efficienza di comunicazione
  - sincronizzazione
  - bilanciamento del carico
- Attenzione:
  - possibile perdita di portabilità
  - possibile perdita di generalità

Costrutti parallelizzabili



- Costrutti (potenzialmente) parallelizzabili:
  - parallelizzabili
    - **`cicli do`**
    - **`for`**
    - **`forall`**
    - **…**
  - non parallelizzabili
    - **`do while`**
    - **`if`**
    - **`goto/continue`**

Calcolo serie



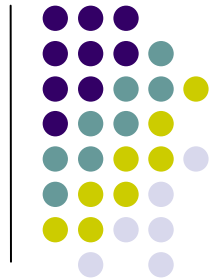
$$\frac{\pi^2}{6} = \sum_{i=1}^{\infty} \frac{1}{n^2}$$

- Codice:

```
do i = 1,n
  somma = somma + 1.0/float(i*i)
enddo
pi_greco = sqrt(somma*6.0)
```

- Come parallelizzare su p processori?

Calcolo serie (2)



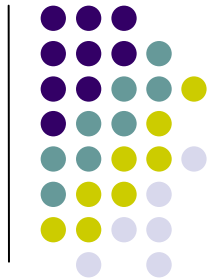
$$\frac{\pi^2}{6} = \sum_{i=1}^{\infty} \frac{1}{n^2}$$

- Codice:

```
dividi n tra p processori
do i = i_start(p),i_end(p)
 somma = somma + 1.0/float(i*i)
enddo
metti insieme le somme parziali
pi_greco = sqrt(somma*6.0)
```

- Servono:
  - distribuzione informazioni (quanti processori?)
  - sincronizzazioni
  - somme parziali

Prodotto matrice-matrice



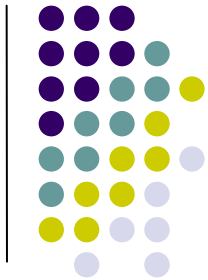
$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

- Codice:

```
do j = 1,n
   do k = 1,n
        do i = 1,n
           c(i,j) = c(i,j)+a(i,k)*b(k,j)
        enddo
   enddo
enddo
```

- Come parallelizzare su p processori?

Prodotto matrice-matrice (2)



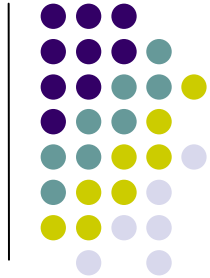
$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

- Codice:

```
distribuisci tra i processori le matrici a,b,c
dividi il lavoro tra p processori
do j = j_start(p),j_end(p)
   do k = 1,n
          do i = 1,n
             c(i,j) = c(i,j)+a(i,k)*b(k,j)
          enddo
   enddo
enddo
ricostruisci la matrice c(i,j)
```

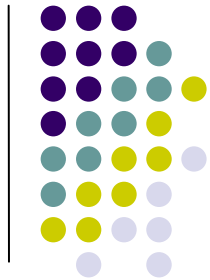
- Servono
  - sincronizzazioni/distribuzione informazioni
  - inizializzazioni di matrici (→ overhead parallelo)

Sincronizzazione: barriere (1)



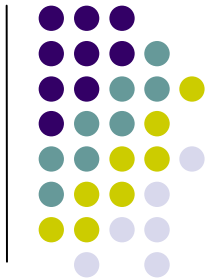
- Le barriere sono meccanismi di sincronizzazione
  - un task si ferma quando arriva ad una barriera
  - quando l'ultimo task arriva alla barriera, i task sono sincronizzati
  - l'elaborazione può ripartire
- servono per:
  - scambiare dati / informazioni
  - mantenere coerente tutta la simulazione
- Sincronizzazione implicita in alcuni costrutti paralleli
- ☹ struttura parallela → overhead
- ☹ problematica con molti processi

Sincronizzazione: barriere (2)



- ☹ Problematica con molti processi
- Si immagini di avere una barriera “ingenua” con 100 processi
  - ogni processo che arriva alla barriera manda un messaggio (a tutti? Al processo master?)
  - in totale una barriera genera > 100 messaggi che potrebbero saturare la rete di interconnessione, e non se ne può perdere nessuno …
  - E se ho 1000 processori?
- Implementazioni reali
  - ad albero → comunicazioni O(log n)
  - butterfly → comunicazioni O(log n)

Sincronizzazione: barriere (3)

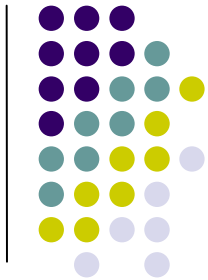


- Operazioni che necessitano di sincronizzazioni:
  - somme di variabili (e.g. somma energia totale)
  - medie (e.g. calcolo velocità media)
  - massimi e minimi

- Qualitativamente:

```
barriera
scambio informazione
calcolo media/somma/max/min
scambio informazione
barriera
```

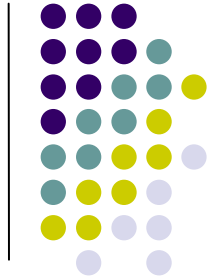
Legge di Amdhal (II)



- Quale è lo speed-up massimo raggiungibile?
- Tempo seriale: T = F + P
  - F = parte seriale (non parallelizzata/parallelizzabile)
  - P = parte parallela
  - C(n) = comunicazione, sincronizzazione etc….
  - T(n) = F + P/n + C(n)
  - S(n) = T(1)/T(n) = (F + P)/(F+P/n+C(n))

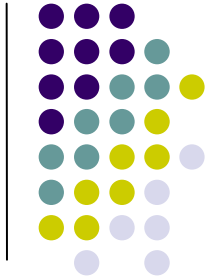
  - per n >> 1 **S(n) = (F + P)/(F+C(n))**
  - Il limite asintotico allo speed-up è peggiore!!!!!
  - ☺ Però potrebbe essere possibile nascondere comunicazione nel calcolo ….

Load balancing



- È importante per le performance dei programmi paralleli
- Distribuire il carico di lavoro in modo che tutti i task siano occupati per tutto il tempo
- Minimizzare i tempi morti (tempi di attesa) dei task
- Partizionare ugualmente il lavoro di ogni task
  - Assegnazione statica: esempi:
    - Per operazioni su array/matrici dove ogni task svolge operazioni simili, uniformemente distribuisco i dati sui task
    - Per cicli dove il lavoro fatto in ogni iterazione è simile, uniformemente distribuisco le iterazioni sui task

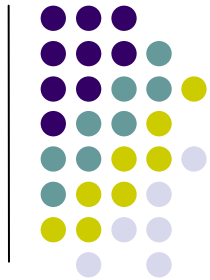
Load balancing: problema tipico



- Ho 4 processi:
  - tempo seriale = 40
  - tempo parallelo (teorico) = 10

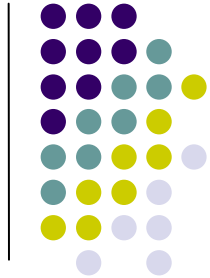
  - tempo parallelo (reale) = 12 → 20% più lento
- la velocità dell'esecuzione dipenderà dal processore più lento

Load balancing: Assegnazione dinamica



- Alcuni partizionamenti anche se uniformi risultano sempre non bilanciati
  - Matrici sparse
  - Metodi con griglie adattative
  - Simulazioni N-corpi
- In questi casi solo l'assegnazione dinamica può riuscire a bilanciare il carico computazionale
- Vista nell'esempio dell'insieme di Mandelbrot
- Divido la regione in tante parti, ogni task prenderà in carico una parte
- Quando un task termina una parte, ne richiede una nuova

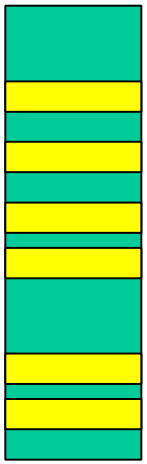
Load balancing: confronti

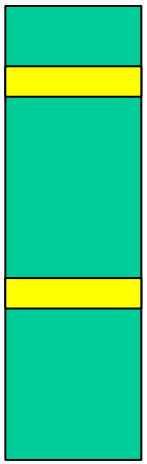


- Assegnazione statica:
  - ☺ in genere semplice, proporzionale al volume
  - ☹ soffre di possibili sbilanciamenti
  - → domain decomposition
- Assegnazione dinamica:
  - ☺ può curare problemi di sbilanciamenti
  - ☹ introduce un overhead dovuto alla gestione del bilanciamento

Granularità

- Misura qualitativa del rapporto tra calcoli e comunicazioni
- Parallelismo a grana fine
  - Pochi conti tra le comunicazioni → rapporto piccolo
  - Può essere facile bilanciare il carico
  - Overhead di comunicazioni
- Parallelismo a grana grossa
  - Molti conti tra le comunicazioni → rapporto grande
  - Può essere difficile bilanciare il carico
  - Probabili aumenti nelle performance
- Qual'è il migliore?

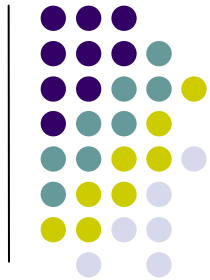




CALCOLI

COMUNICAZIONI

Input e Output

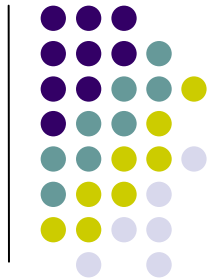


- Le operazioni di I/O sono generalmente seriali
- Possono creare “colli di bottiglia”

- La gestione dell’I/O porta all’utilizzo di costrutti specifici del linguaggio e del modello di programmazione usato:

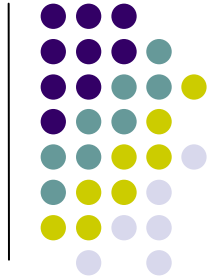
  - Gather/Scatter per rimettere insieme/distribuire i dati su modelli message-passing (MPI,SMS)
  - La gestione è trasparente all’utente in SMS
  - Utilizzo di un solo thread nel modello di programmazione multi-threaded (OpenMP)

Domain decomposition



- Come associare (staticamente) il carico computazionale ad ogni processore?
  - Decomporre il problema originale in sottoproblemi, possibilmente in modo ricorsivo:
    - Tecniche *"Dividi et impera"*
      - Esempio: Calcolo del Fattoriale
    - **O**rthogonal **R**ecursive **B**isection (ORB)
      - Esempio: Problemi N-body
    - **M**etodi di ***D**omain **D**ecomposition* (DDM) www.ddm.org

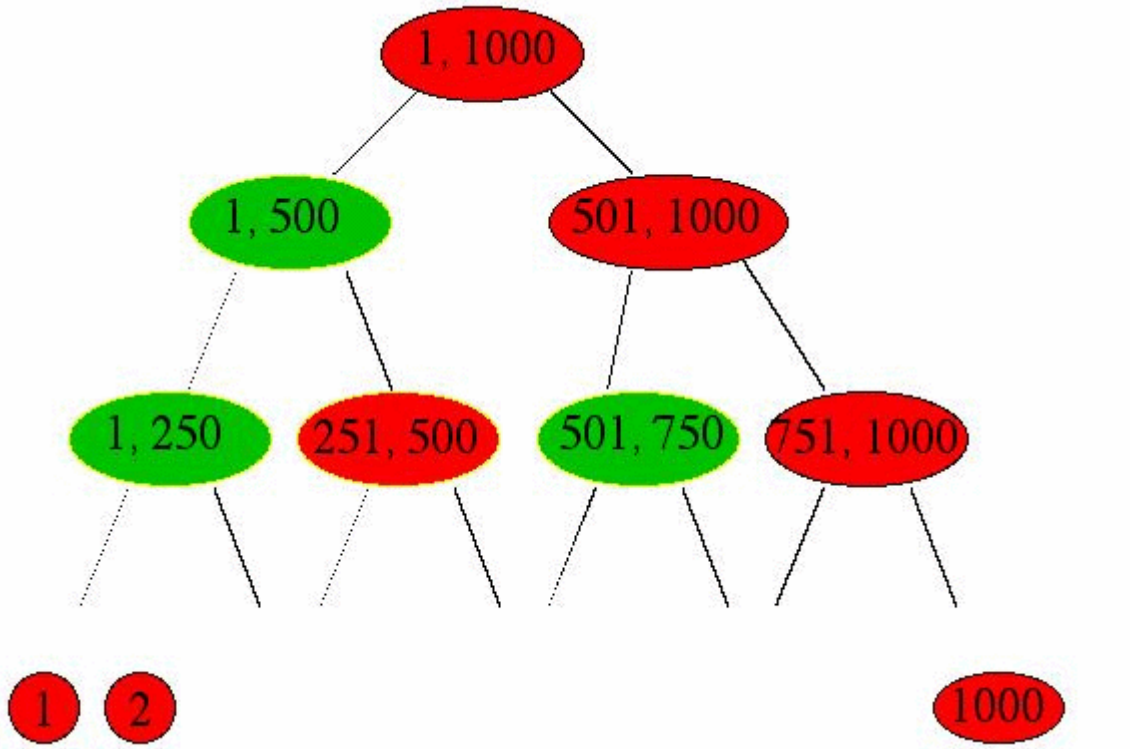
Tecniche *Divide et Impera*



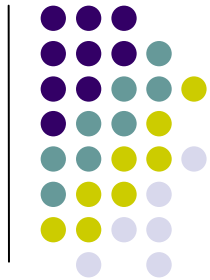
- Si divide in modo ricorsivo il problema originario in sottoproblemi dello stesso tipo
- Sono parallelizzabili in quanto processori separati possono essere utilizzati per parti distinte. I dati sono localizzati in modo naturale.

Tecniche *Divide et Impera:* esempio

- Calcolo del fattoriale



Metodi di *Domain Decomposition*

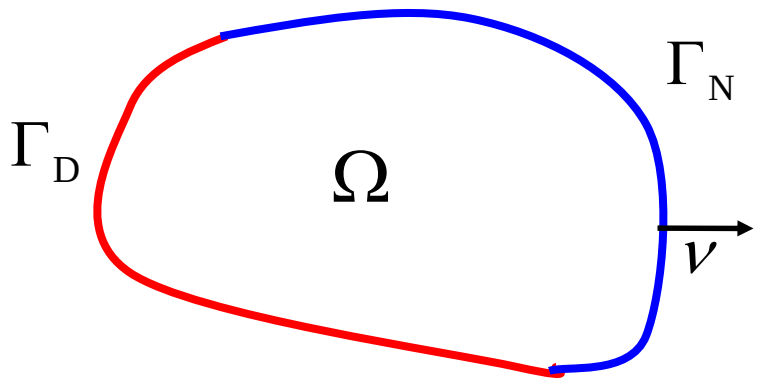


- Introdotti per risolvere problemi differenziali (PDE) su domini a geometria complessa
- Si suddivide il dominio computazionale in sottodomini di forma più semplice
- La soluzione del problema originale è ridotta alla soluzione di problemi dello stesso tipo sui sottodomini
- La consistenza tra i due problemi è assicurata imponendo opportune condizione di trasmissione tra sottodomini adiacenti
- Metodi con Overlapping (Metodi di Schwarz - 1869)
  - I sottodomini si sovrappongono
- Metodi senza Overlapping
  - I sottodomini non si sovrappongono
- www.ddm.org

Il metodo di Schwarz a 2 domini (1)

Supponiamo di voler risolvere il classico problema:

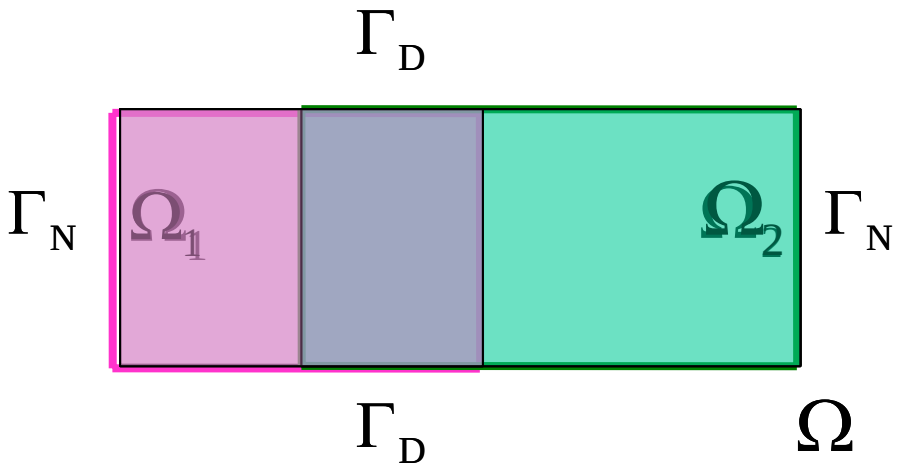
$$\textbf{(P)} \quad \begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{su } \Gamma_D \\ \nabla u \cdot \nu = g & \text{su } \Gamma_N \end{cases}$$



$$\partial\Omega = \Gamma = \Gamma_D \cup \Gamma_N$$

Il metodo di Schwarz a 2 domini (2)

Consideriamo per semplicità un rettangolo Ω e suddividiamolo

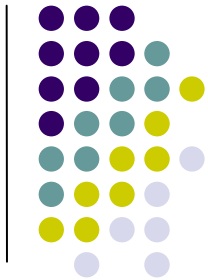


$$\Omega_1 \cup \Omega_2 = \Omega$$

$$\Omega_1 \cap \Omega_2 \neq \emptyset$$

in due sottodomini Ω_1 e Ω_2 che si sovrappongono e sostituiamo la soluzione diretta di **(P)** con la soluzione della procedura iterativa nei due sottodomini.

Il metodo di Schwarz a 2 domini (3)



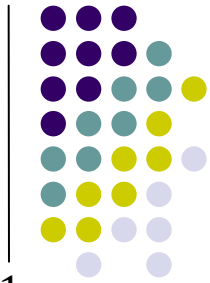
(P1)

$$\begin{cases} -\Delta u_1^{n+1} = f & \text{in } \Omega_1 \\ u_1^{n+1} = 0 & \text{su } \Gamma_D \text{ I } \partial\Omega_1 \\ \nabla u_1^{n+1} \cdot \nu = g & \text{su } \Gamma_N \text{ I } \partial\Omega_1 \\ u_1^{n+1} = u_2^{n} & \text{su } \partial\Omega_1 \text{ I } \Omega_2 \end{cases}$$

(P2)

$$\begin{cases} -\Delta u_2^{n+1} = f & \text{in } \Omega_2 \\ u_2^{n+1} = 0 & \text{su } \Gamma_D \text{ I } \partial\Omega_2 \\ \nabla u_2^{n+1} \cdot \nu = g & \text{su } \Gamma_N \text{ I } \partial\Omega_2 \\ u_2^{n+1} = u_1^{n+1} & \text{su } \partial\Omega_2 \text{ I } \Omega_1 \end{cases}$$

Il metodo di Schwarz a 2 domini (4)



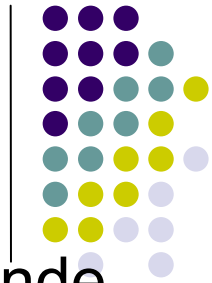
Noti u_1^n e u_2^n al passo *n*, si risolve **(P1)** e si calcola u_1^{n+1} e da questo, risolvendo **(P2)**, si può calcolare u_2^{n+1}

Si costruisce così una procedura che a ogni iterazione risolve il problema originale su un sottodominio per volta. Ovviamente la risoluzione di ogni sottoproblema richiede una condizione al bordo in più sulla frontiera $\partial\Omega_i \text{ I } \Omega_j$ interna e per questo si impone la condizione di Dirichlet

$$u_i^{n+1} = u_j^n$$

Questo è il metodo di *Schwarz moltiplicativo*: accoppia i sottoproblemi. È facile da implementare e si applica a un qualunque operatore differenziale.

Il metodo di Schwarz a 2 domini (5)



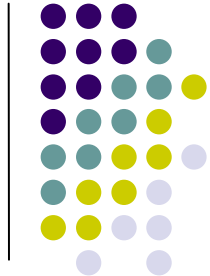
Esiste anche una *variante additiva* che rende completamente indipendenti i sottoproblemi. In questa versione la relazione $u_2^{n+1} = u_1^{n+1}$ su $\partial\Omega_2$ I Ω_1 diventa:

$u_2^{n+1} = u_1^{n}$ su $\partial\Omega_2$ I Ω_1

Essendo i problemi tra loro disaccoppiati, non c'è scambio di informazioni tra un sottodominio e l'altro.

Questi problemi si prestano ad essere eseguiti in parallelo.

Il metodo di Schwarz a N domini

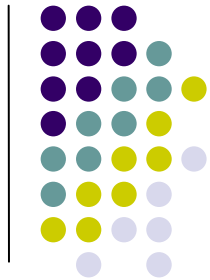


Il metodo può essere generalizzato al caso in cui il dominio di partenza sia suddiviso in un numero N qualunque di sottodomini, che si sovrappongono.

Oltre ai problemi di convergenza, l'unica difficoltà nella realizzazione di questi metodi sta nella costruzione della overlapping partition del dominio originale.

Esistono diverse tecniche per costruirla (www.ddm.org)

Riferimenti Web e Bibliografici (parziali)



- www.ddm.org
- www.top500.org
- Akhter & Roberts **“Multi-Core Programming”**, Intel Press
- Pacheco**: “Parallel Programming with MPI”**,MK Publisher
- Hennessy & Patterson: “**Computer Architecture: A Quantitative Approach**”, MK Publisher, (3rd edition)