AGENZIA NAZIONALE
PER LE NUOVE TECNOLOGIE, L'ENERGIA
E LO SVILUPPO ECONOMICO SOSTENIBILE

Corso di formazione per il progetto TEDAT
ENEA C. R. Brindisi, 24-25 giugno 2013

Introduzione al calcolo scientifico ad alte prestazioni

Agostino Funel
agostino.funel@enea.it
ENEA Centro Ricerche Portici
P.le Enrico Fermi 1, Portici (Napoli)

Argomenti

PARTE I

- Breve introduzione storica all'informatica
- Esempi concreti della necessità dei supercalcolatori
- Evoluzione tecnologica: verso i sistemi massicciamente paralleli
- Supercalcolo e calcolo distribuito: un accenno a ENEA-GRID
- La top500: la classifica mondiale dei supercalcolatori

PARTE II

- Principi del calcolo parallelo
- Il paradigma MPI (Message Passing Interface)
- Valutazione delle prestazioni
- Esempi

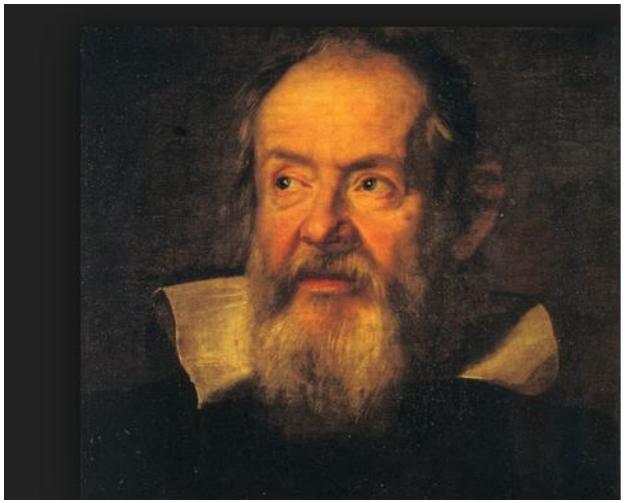
Introduzione

Il metodo scientifico: confronto tra teoria ed esperimenti.

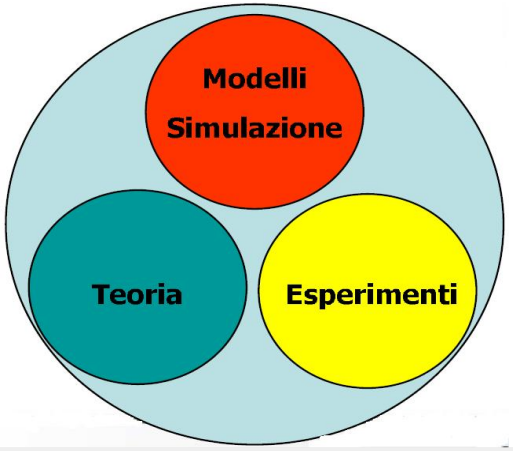
Teoria : problemi complessi, conti complicati, metodi numerici.

Esperimenti : producono moltissimi dati da analizzare.

Modelli ⟷ Simulazioni



Galileo Galilei (Pisa, 1564 – Arcetri, 1642)



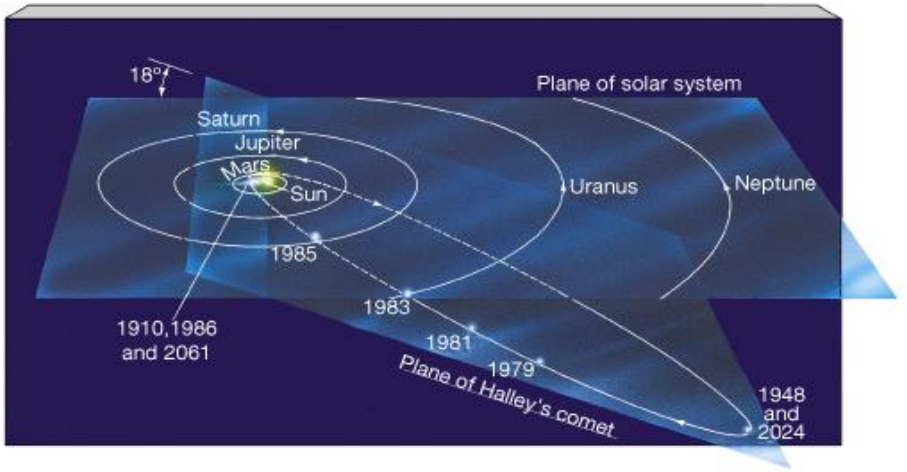
Sono necessari strumenti di calcolo automatico

Un po' di storia/1

1757 : la data di ritorno della cometa di Halley viene calcolata dai tre matematici francesi Alexis Clairault, Joseph Lalande e Nicole-Reine Lepaute : "il ritorno della cometa avverrà con 651 giorni di ritardo per effetto dei pianeti Giove e Saturno".

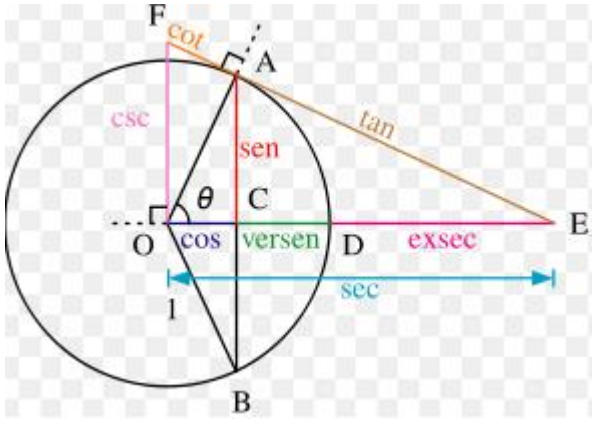
La cometa anticipa di soli 31 giorni la data prevista.

Il calcolo aveva impegnato a tempo pieno per 5 mesi i tre matematici!



Un po' di storia/2

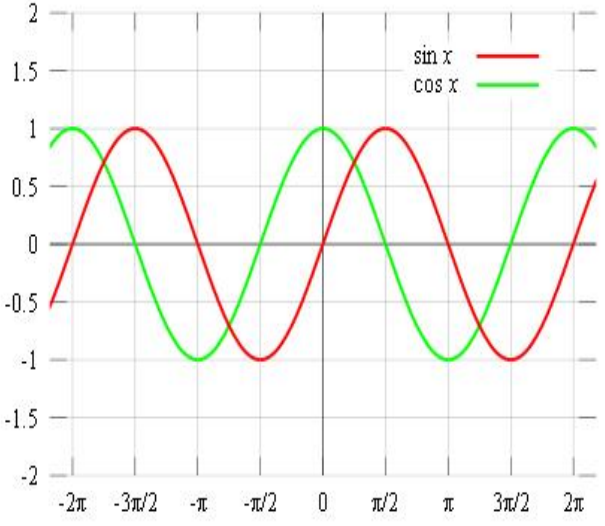
1790 : l'Assemblea Nazionale francese introduce il sistema metrico. La proposta si estende anche alla misura degli angoli : **l'angolo retto viene diviso in 100 parti.**



Servono **nuove tavole trigonometriche!**

L'incarico viene assegnato ad un ingegnere Gaspard de Prony :

Una squadra di 90 "calcolatori" lavora per 3 anni producendo 18 volumi manoscritti.



1795 : la nuova unità viene abbandonata.

L'informatica moderna

Fino alla II guerra mondiale “computer” indica una professione : il “Mathematical Table Project” (USA, 1938) impiega 450 “computers” per preparare tavole matematiche di ogni tipo.

L'evoluzione delle tecnologie elettromeccaniche/elettroniche negli anni di guerra produce i primi “computer” automatici (Z3 Germania 1941, ENIAC USA 1943...)

Durante la II guerra mondiale i computer furono utilizzati in USA nei Los Alamos Laboratories per la costruzione della prima bomba atomica.

L'invenzione del microprocessore (1970) rende possibile l'informatica di consumo (>1980) e della connettività diffusa (>1990).

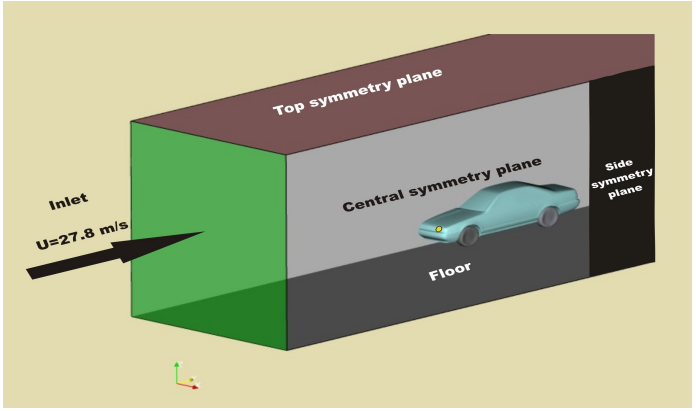
J. Von Neumann
(Budapest 1903 - Washington 1957)

Oggi il PC di casa esegue ~10^9 operazioni/secondo e comunica a 10^5 – 10^6 caratteri/secondo.

Scienze computazionali/1

A partire dalla teoria di base è possibile costruire modelli concreti il cui studio dettagliato richiede spesso l'esecuzione di moltissime operazioni matematiche. In questo caso è assolutamente necessario usare i supercalcolatori. A volte il calcolo consente di predire fenomeni sia su piccola che su grande scala.

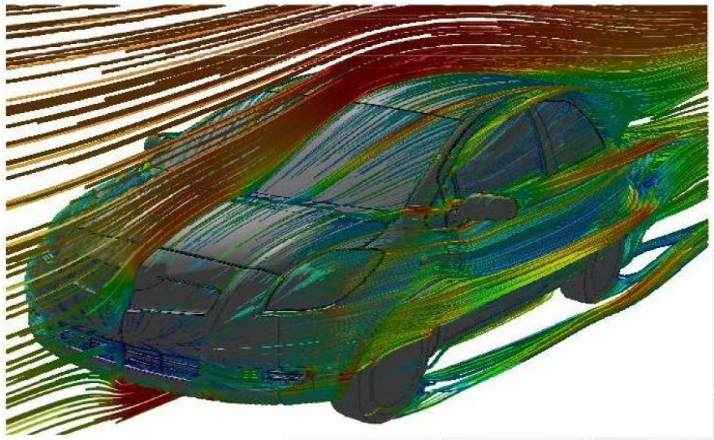
Un caso concreto : simulazione di aerodinamica esterna di interesse industriale.



La simulazione è stata fatta sul supercalcolatore CRESCO dell'ENEA.

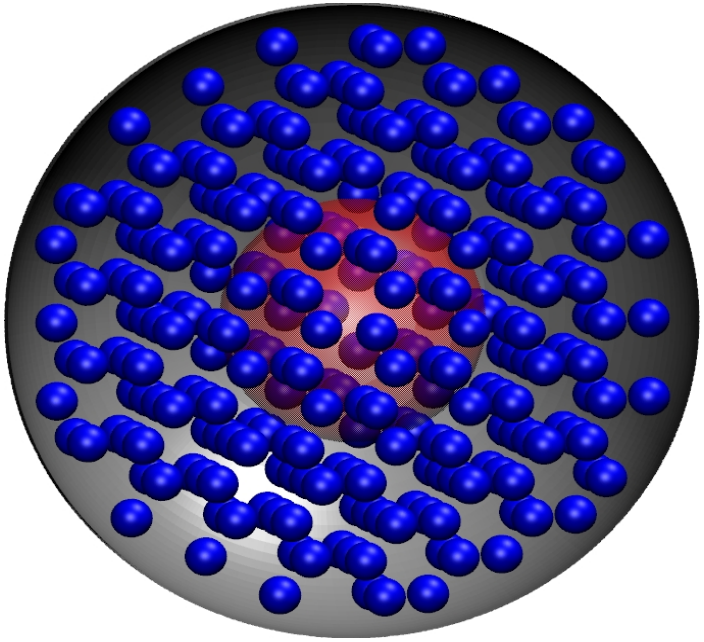
30 milioni di celle

$\sim 5x10^{15}$ operazioni matematiche



Scienze computazionali/2

Un caso concreto : simulazione di dinamica molecolare per studiare l'immagazzinamento dell'idrogeno nei metalli ibridi.

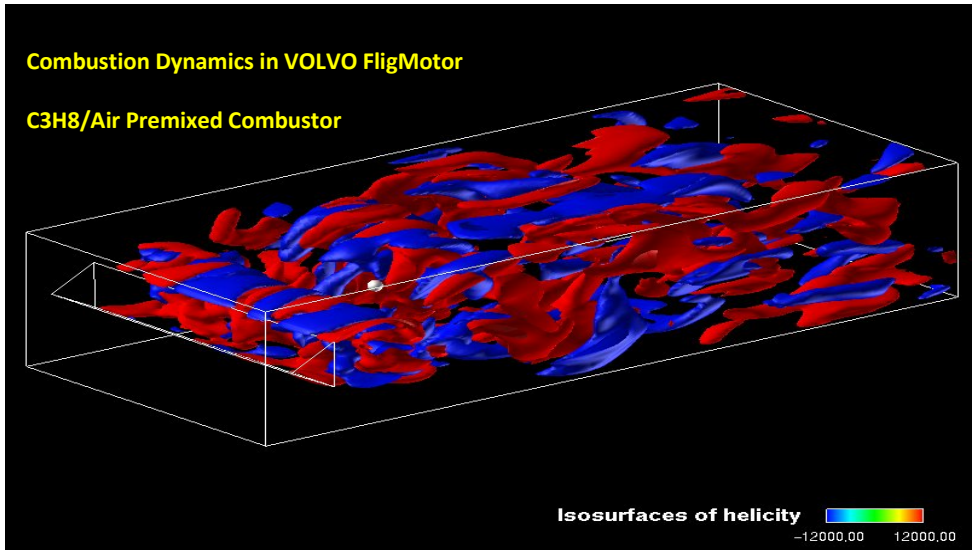


Il sistema è composto da una cavità di nanoparticelle contenente 266 atomi di Mg.

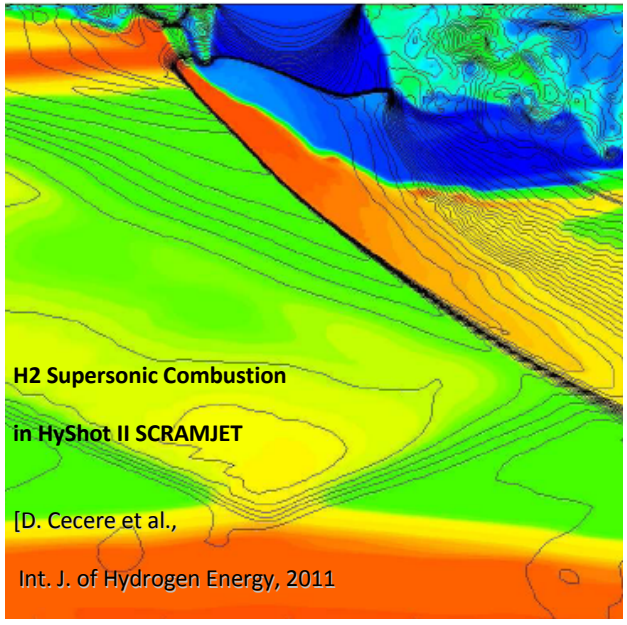
Calcolo della densità elettronica.

La simulazione è stata fatta sul supercalcolatore CRESCO dell'ENEA.

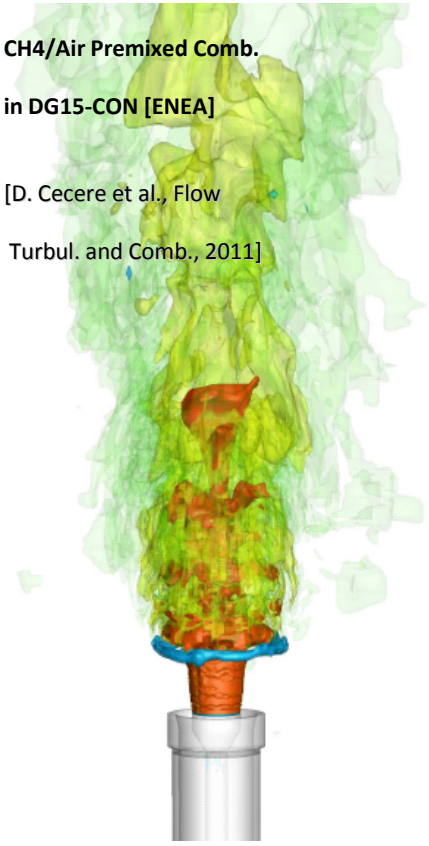
La simulazione ha richiesto in media $\sim 1.8 \times 10^{14}$ operazioni matematiche per fare una sola iterazione.



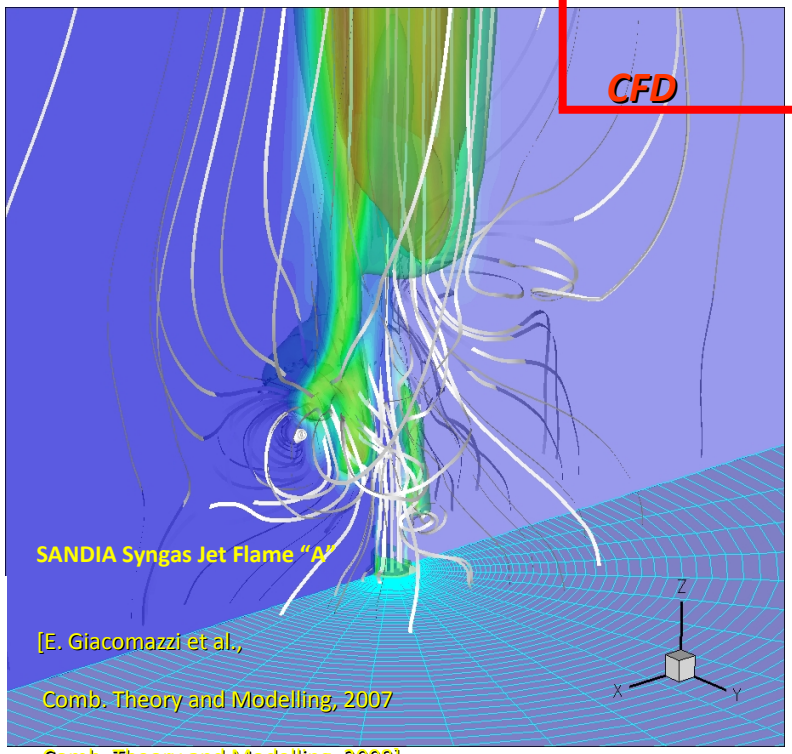
[E. Giacomazzi et al., Comb. and Flame, 2004]



Shock Waves, 2012]

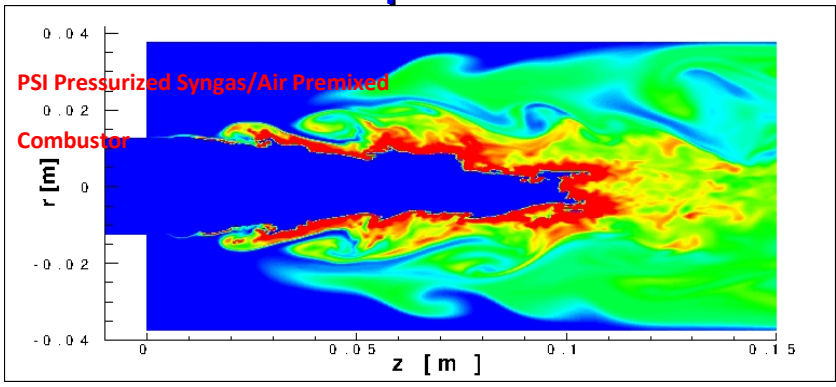


[D. Cecere et al., Flow Turbul. and Comb., 2011]



Comb. Theory and Modelling, 2008]

Combustione: simulazione fatta sul supercalcolatore CRESCO dell'ENEA.3 mesi di calcolo con più di 1000 core



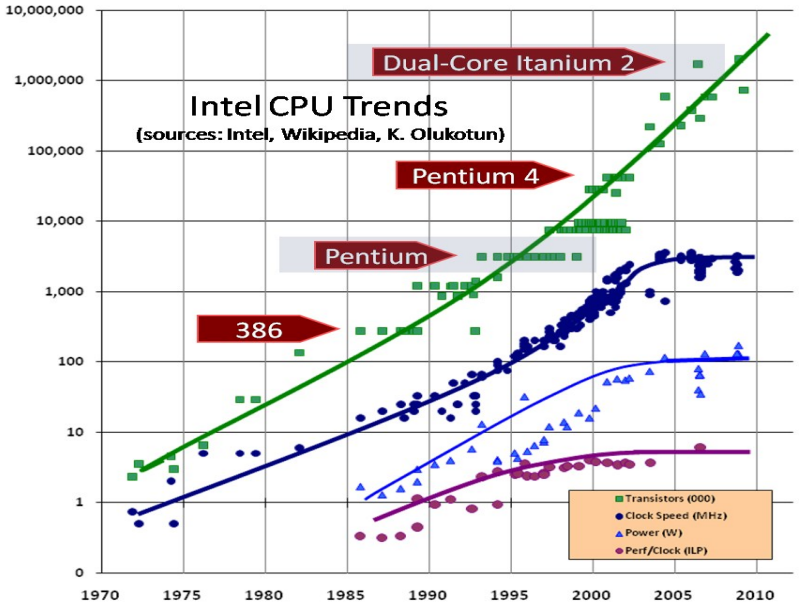
Il supercalcolo (HPC High PerformanceComputing)

Supercomputer: sono i calcolatori più potenti disponibili in un dato arco di tempo. **Potente:** in termini di **velocità di esecuzione**, **capacità di memoria**, **precisione di macchina.**

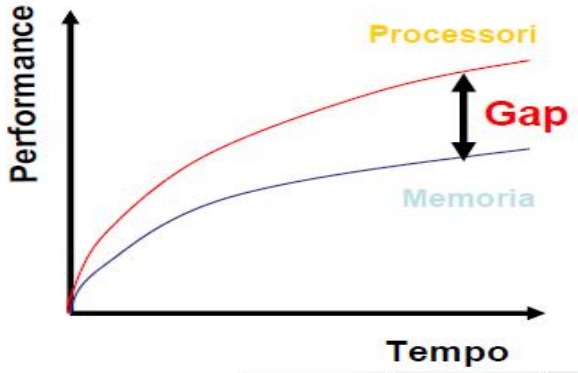
Aumentare la velocità dei componenti elettronici: limite imposto dalla velocità della luce, problemi di dissipazione del calore.

Aumentare il numero di componenti elettronici: sistemi massicciamente paralleli. Sì, ma non basta...velocità diverse per componenti diversi generano cali di prestazioni.

Nuovo paradigma di sviluppo dei codici!



Legge di Moore:il numero dei transistor per unità di area raddoppia ogni 18 mesi.



Supercalcolo e calcolo distribuito

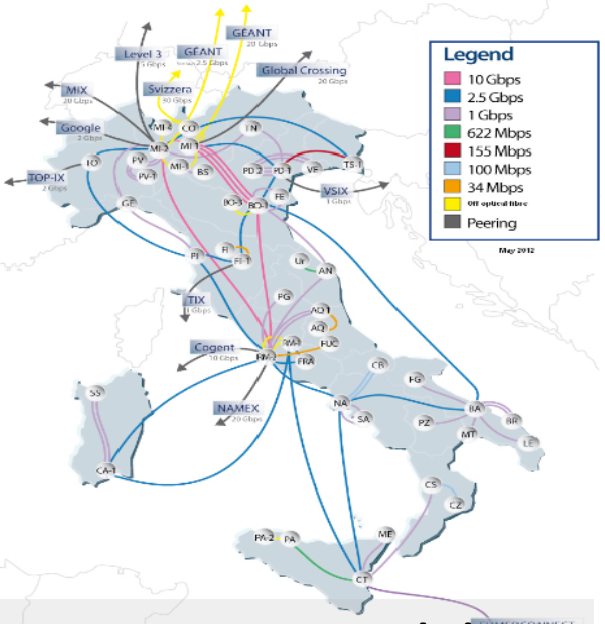
La disponibilità di connessione di rete ad alta velocità ha suggerito la possibilità di ottenere grandi prestazioni di calcolo collegando tra loro sistemi disponibili in siti remoti.

ENEAGRID integra in un'unica infrastruttura l'insieme delle risorse di calcolo scientifico di ENEA, distribuite nei suoi principali Centri di Ricerca.

Le risorse di calcolo più importanti di ENEA sono i sistemi **CRESCO**.

I Centri **ENEA-GRID** sono connessi dalla rete **GARR**.

I core HPC CRESCO
Portici: ~5500 (+ 5000)
Frascati: ~500
Casaccia: ~200
Brindisi: ~100



La classifica dei supercomputer

Il LINPACK Benchmark

È il benchmark utilizzato per classificare la potenza computazionale dei supecomputer in base al numero di operazioni in virgola mobile che il sistema può fare in 1 secondo

FLOPS: FLoating-point Operations Per Second.

Computer performance

Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

Top500 List - June 2013

www.top500.org

R_{max} and R_{peak} values are in TFlops.

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National University of Defense Technology China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808

Principi del calcolo parallelo

Argomenti

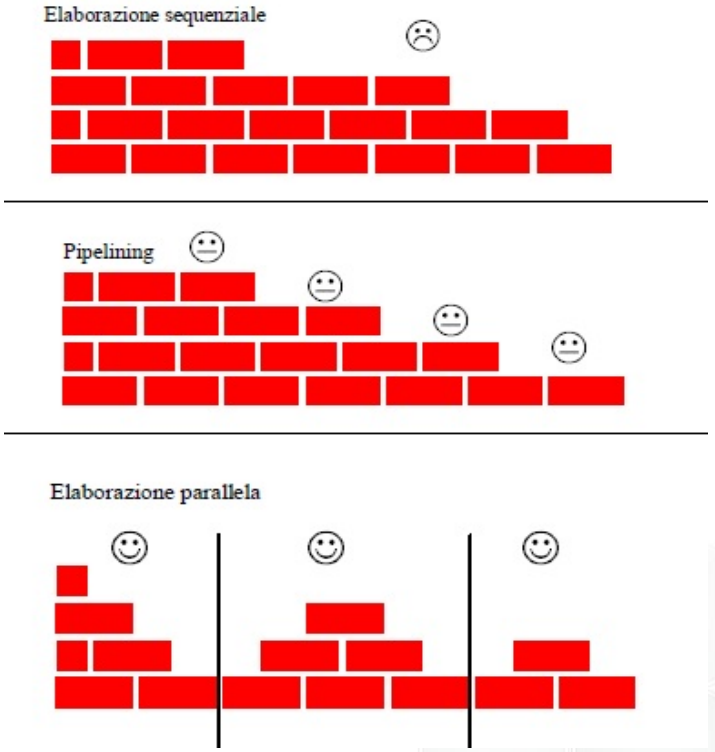
- Concetto di elaborazione parallela
- Principali architetture parallele
- Paradigmi della programmazione parallela
- Modelli di parallelismo
- Gestione dei conflitti

Elaborazione parallela

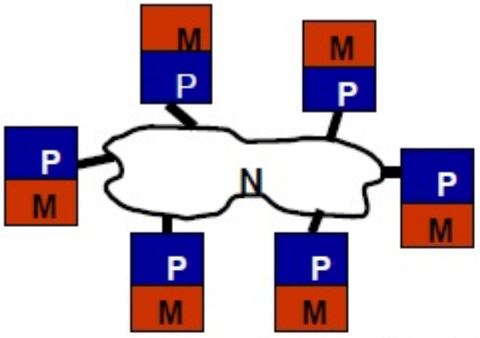
Filosofia: dividere un problema complesso in più parti che possono essere risolte separatamente ed elaborarle simultaneamente ("in parallelo").

Programmazione parallela: programma in cui più processi (task) comunicano tra loro per risolvere un problema mediante algoritmi non sequenziali.

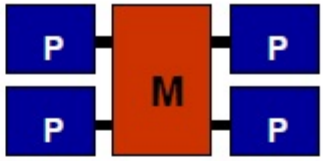
Calcolatore parallelo: sistema multiprocessore in grado di eseguire programmi paralleli.



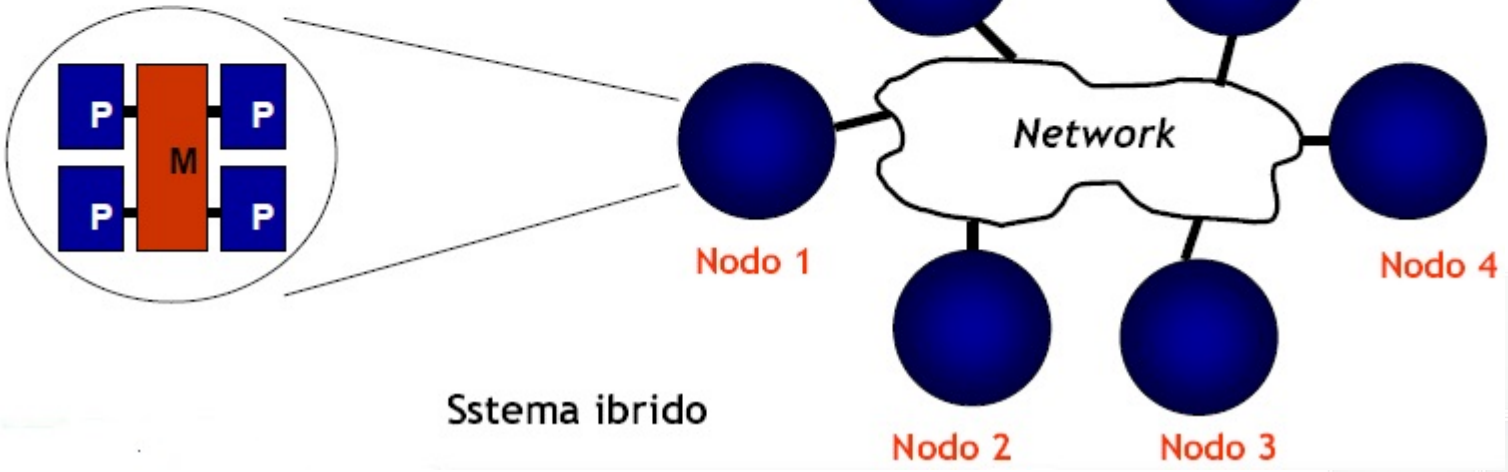
Architetture parallele



Sstema a Memoria Distribuita



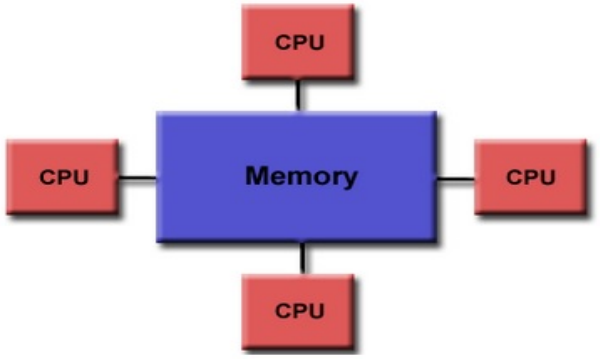
Sistema a Memoria Condivisa



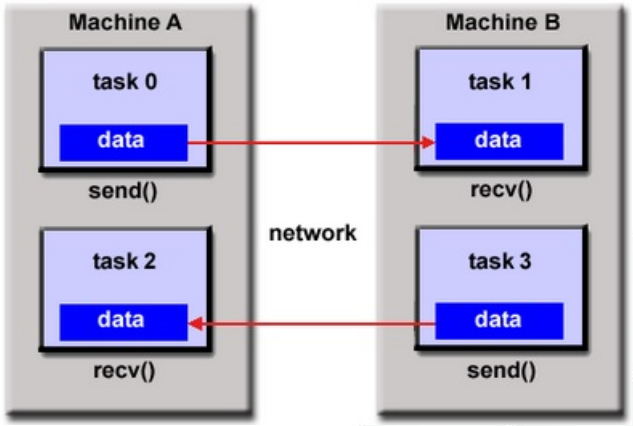
Sstema ibrido

Paradigmi della programmazione parallela/1

Data Parallel (memoria condivisa): tutti i processi, generalmente thread, vedono la stessa memoria condivisa e possono direttamente accedervi. Variabili e strutture dati condivise.



Message Passing (memoria distribuita): ciascun processo può accedere solo alla sua memoria locale e comunica con gli altri processi scambiandosi messaggi.

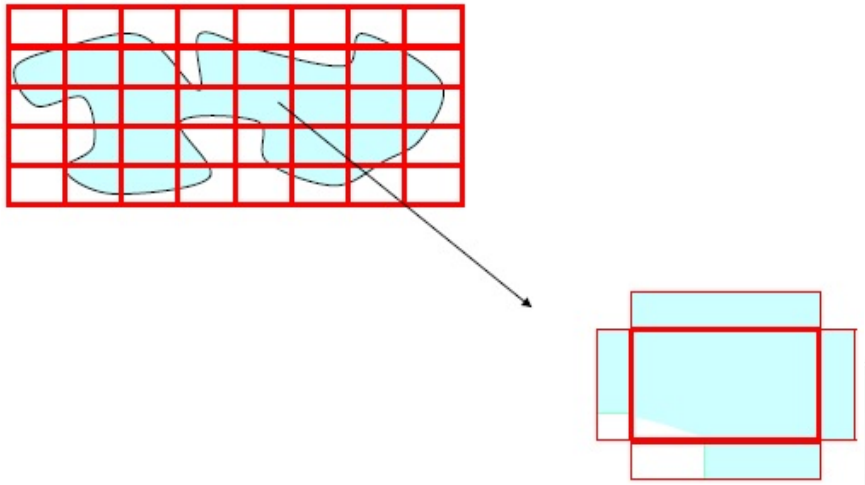


Paradigmi della programmazione parallela/2

Data Parallel	Message Passing
Compilatori ad hoc	Compilatori standard
Linguaggi: Fortran/90, C/C+,...	Linguaggi: Fortran/90, C/C++,...
Direttive al compilatore nel codice sorgente	Librerie per la comunicazione
Eseguibile si lancia come un comando di shell Unix	L'eseguibile si lancia mediante appositi wrapper
Standard: **Open MP**, HPF,...	Standard: **MPI**, PVM,...

Modelli di parallelismo/1

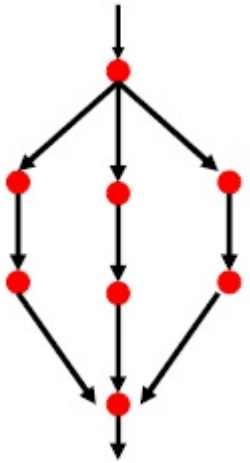
Parallelismo sui dati (domain decomposition) : i dati sono divisi in parti aventi approssimativamente la stessa dimensione e sono mappati su processori diversi. Ciascun processore esegue lo stesso programma sulla sua parte di dati (SPMD Single Program Multiple Data).



-Il posizionamento dei dati è critico
-Problema della gestione del bordo
-Problema del bilanciamento del carico
-Programmazione in message passing

Modelli di parallelismo/2

Parallelismo funzionale (functional decomposition) : il problema è suddiviso in un gran numero di "compiti" o "funzioni" o "sottoprogrammi", che vengono assegnati ai processori. Ogni processore esegue una diversa "funzione" appena è disponibile. Un processore **può operare su dati condivisi (MPSD Multiple Program Single Data) oppure su dati privati (MPMD Multiple Program Multiple Data). Implementazione: client/server oppure master/slave.**



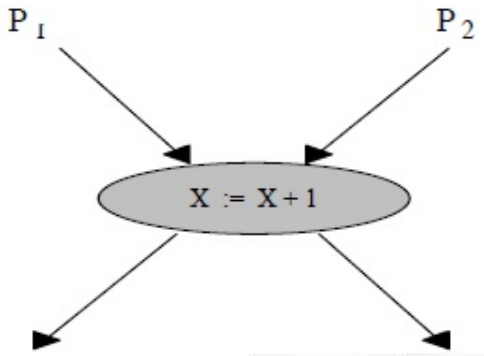
-Identificare le funzioni e i requisiti sui dati
-Gestione dei conflitti sui dati condivisi
-Programmazione in data parallel o message passing

Gestione dei conflitti/1

Modello Shared Memory.

L'accesso concorrente in lettura a una stessa locazione di memoria non causa problemi: ogni processore fa una copia del contenuto della locazione di memoria e la memorizza in un proprio registro.

Il problema si verifica quando c'è un accesso concorrente in scrittura ossia quando più processi scrivono simultaneamente sulla stessa locazione di memoria.



Il programmatore, il linguaggio di programmazione e/o l'architettura devono fornire strumenti per gestire il conflitto.

Gestione dei conflitti/2

Modello Shared Memory.

Il non determinismo (race condition): avviene quando due istruzioni in task concorrenti accedono la stessa locazione di memoria e almeno una delle istruzioni è in scrittura. Non c'è un ordine di esecuzione garantito tra gli accessi.

Il problema si può risolvere sincronizzando l'utilizzo dei dati condivisi.

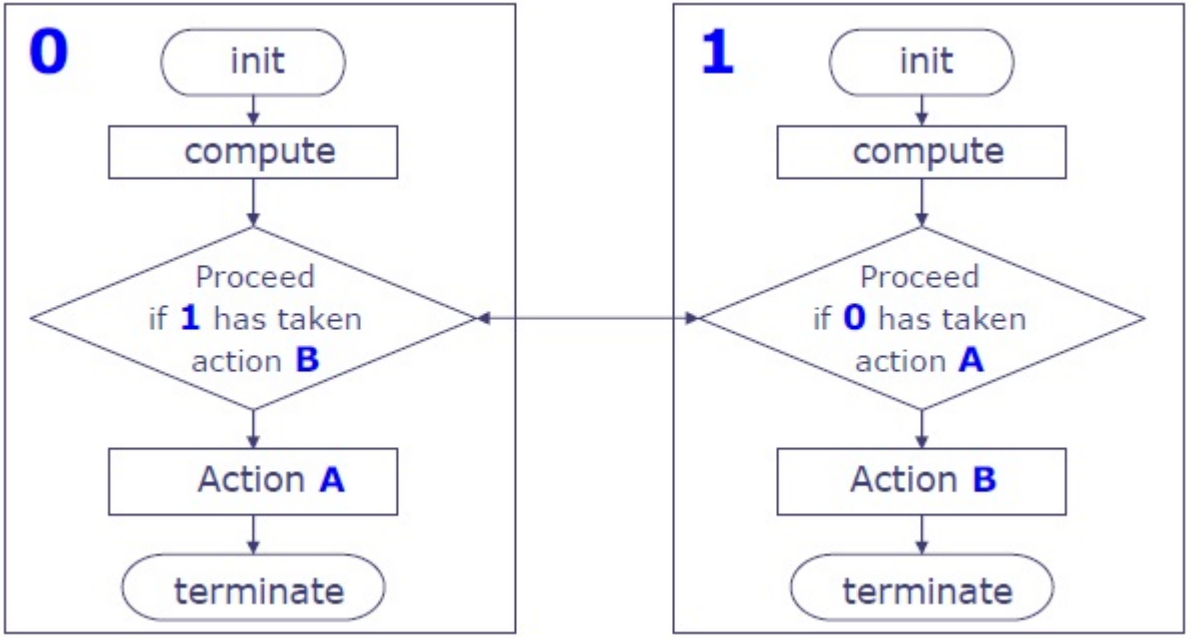
Le porzioni di programma che richiedono sincronizzazione sono dette sezioni critiche. Occorrono costrutti per accedere in modo mutuamente esclusivo alle sezioni critiche.

Processor 1:	Processor 2:
LOCK (X)	*LOCK (X)*
X = X + 1	X = X + 2
UNLOCK (X)	*UNLOCK (X)*

Gestione dei conflitti/3

Deadlock o stallo.

Si verifica quando due (o più) processi sono bloccati perché ciascuno aspetta che l'altro faccia qualcosa.



MPI – Message Passing Interface

Argomenti

- Il paradigma MPI
- Tipi di comunicazione
- Tipi di sincronizzazione
- Aspetti importanti della tecnica di programmazione
- Struttura di un programma MPI
- Comunicazione point-to-point
- Comunicazioni collettive
- Valutazione delle prestazioni
- Esempi

MPI – Message Passing Interface/1

Standard per la programmazione parallela:
http://www.mpi-forum.org/

Ogni processo ha le sue risorse locali (spazio degli indirizzi logici disgiunto). La comunicazione avviene attraverso scambi di messaggi.

Messaggi: istruzioni, dati, segnali di sincronizzazione.

Lo schema MPI si può implementare anche su architetture shared memory, ma non conviene perché il ritardo della comunicazione è maggiore di quello che si ha accedendo a variabili condivise in una memoria comune.

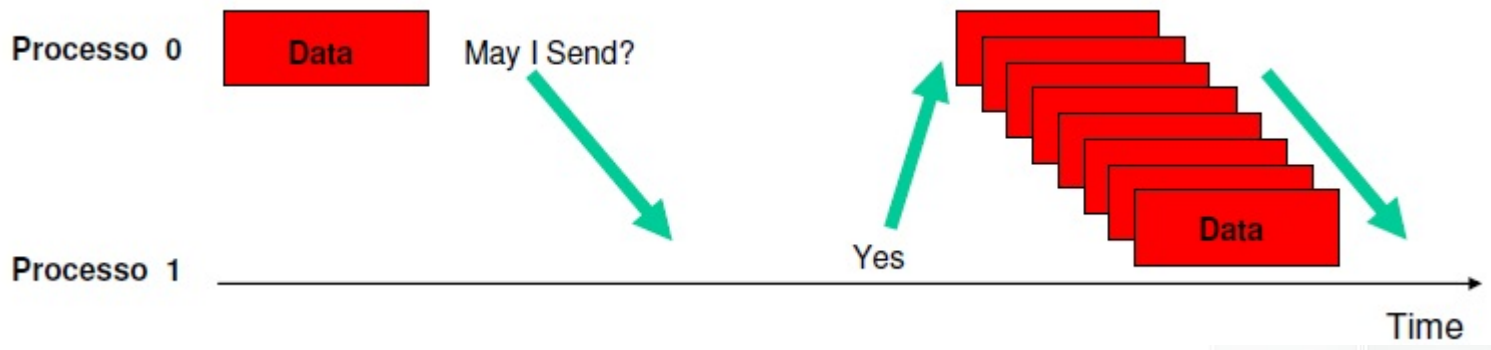
MPI – Message Passing Interface/2

Il processo mittente e quello destinatario devono cooperare.

Tempo della comunicazione: inizializzazione + trasferimento dati + sincronizzazione + chiusura (finalizzazione) dei processi.

Due operazioni fondamentali:

Send(messaggio, parametri)
Receive(messaggio, parametri)



Tipi di comunicazione

Comunicazione simmetrica:

il processo mittente nomina esplicitamente il destinatario
il destinatario nomina il processo con cui vuole comunicare
lo schema di comunicazione è da uno a uno

Comunicazione asimmetrica:

il mittente nomina esplicitamente il destinatario
il destinatario non indica il nome del processo con cui vuole comunicare
schemi di comunicazione:

- **da molti a uno**
- **da uno a molti**
- **da molti a molti**

Tipi di sincronizzazione

Bloccante:

- il processo mittente si blocca e attende che l'operazione richiesta dalla send giunga a compimento
- il processo ricevente rimane bloccato finché sul canale da cui vuole ricevere non viene inviato il messaggio richiesto

Non bloccante:

-il processo che emette una send non bloccante non attende che il messaggio spedito sia andato a buon fine, ma passa ad eseguire le istruzioni successive. Altre primitive consentono di verificare lo stato del messaggio inviato

- il processo che invoca una receive non bloccante consente di verificare lo stato del canale e restituisce il messaggio oppure una flag che indica che il messaggio non è ancora arrivato

Aspetti importanti della tecnica di programmazione parallela

Load Balancing

-Dividere equamente il carico di lavoro tra le risorse disponibili: processori, memoria, network bandwidth, I/O,...

-Più facile nel domain decomposition che nel functional decomposition model

Minimizzare la comunicazione

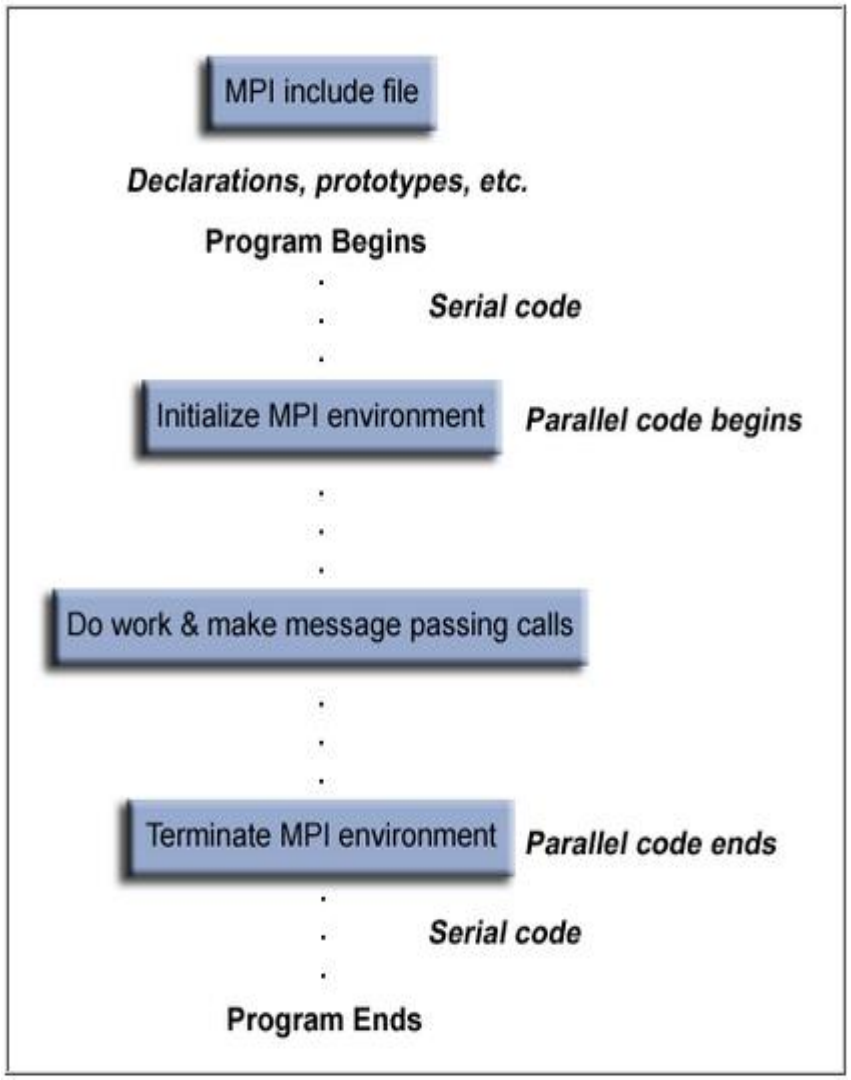
-Raggruppare tante piccole comunicazione in una sola più grande

-Eliminare il più possibile le sincronizzazioni

Sovrapporre comunicazione e calcolo

-Programmare in modo che i processi continuino ad eseguire calcolo mentre comunicano

MPI – Tipica struttura del programma




```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER       0
int main (int argc, char *argv[])
{
int   numtasks, taskid, len;
char hostname[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);
printf ("Hello from task %d on %s!\n", taskid, hostname);
if (taskid == MASTER)
  printf("MASTER: Number of MPI tasks is:
      %d\n",numtasks);
MPI_Finalize();
}
```

MPI – Compilazione ed esecuzione

La compilazione non è standard:

- Si devono specificare i path degli include (-I/mpidir/include)
- Il linker deve conoscere i path delle librerie (-L/mpidir/lib -lmpi)
- Se possibile utilizzare i wrapper per compilare/linkare (mpicc, mpif90)

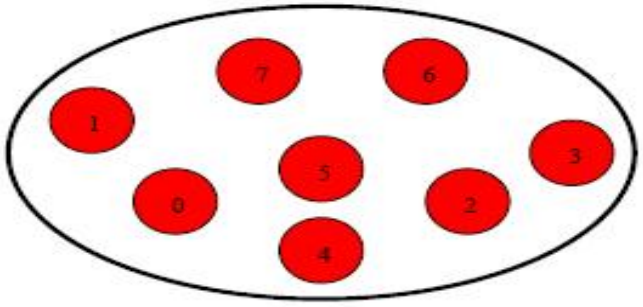
L'esecuzione non è standard:

- Utilizzare degli appositi wrapper (mpirun, mpiexec)

MPI Communicator

Il Communicator è una variabile che identifica un gruppo di processi ai quali è consentito di comunicare tra loro.

Il comunicatore di default è MPI_COMM_WORLD ed include tutti i processi: tutti possono comunicare con tutti.



MPI_COMM_WORLD

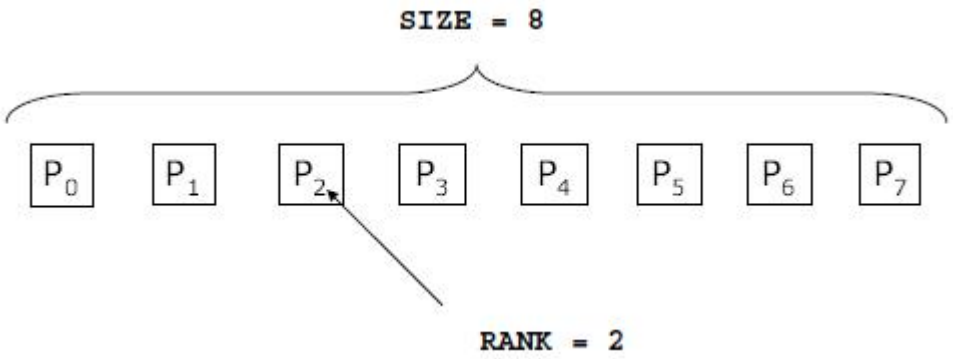
Tutte le funzioni di comunicazione MPI hanno come argomento un comunicatore.

Il programmatore può definire più comunicatori all'interno dello stesso programma.

Dimensione del Communicator e rango di un processo

Size: numero dei processi associati al comunicatore

Rank: è l'indice del processo all'interno del gruppo associato al comunicatore (rank = 0, 1,...,N-1). Il rank è usato per identificare il processo mittente e destinatario in una comunicazione.



La sintassi delle funzioni

C	Fortran
Error = MPI_Xxxx(parametro,...) MPI_Xxxx(parametro,...)	CALL MPI_XXXX(parametro,...,IERROR)

Quanti processi sono associati al comunicatore?

MPI_Comm_size(MPI_Comm, int* size)

OUTPUT : size

Qual è il rank del processo?

MPI_Comm_rank(MPI_Comm, int *rank)

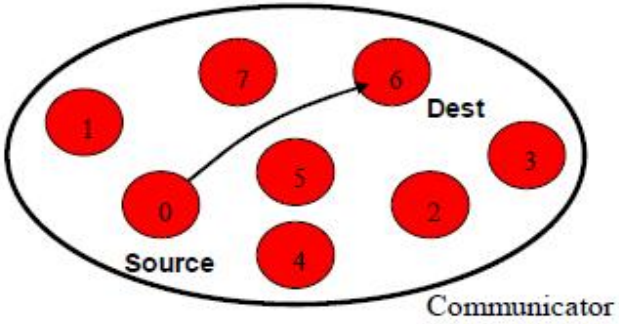
OUTPUT : rank

Comunicazione Point-to-Point

La comunicazione point to point è la base della comunicazione MPI.
Concettualmente semplice: A invia un messaggio a B e B riceve il messaggio da A.
La comunicazione avviene all'interno di un comunicatore.
Il mittente e il destinatario sono identificati dal loro rank all'interno del comunicatore.

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype



-Il messaggio è un array di tipi di dato MPI
-Tipi di dato differenti per C e Fortran
-Il messaggio può essere ricevuto solo se il ricevente specifica correttamente la busta (envelope)

Standard Send e Receive

Fortran:

```
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, dest, tag, comm, status, ierr)
```

Message body — envelope

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int
   dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype type, int
   dest, int tag, MPI_Comm comm, MPI_Status *status);
```

`Buf` array of type `type` see table.

`Count` (INTEGER) number of element of `buf` to be sent

`Type` (INTEGER) MPI type of `buf`

`Dest` (INTEGER) rank of the destination process

`Tag` (INTEGER) number identifying the message

`Comm` (INTEGER) communicator of the sender and receiver

`Status` (INTEGER) array of size `MPI_STATUS_SIZE` containing communication status information

`Ierr` (INTEGER) error code (if `ierr=0` no error occurs)

Sia in C che in Fortran MPI_RECV accetta le widcard MPI_ANYSOURCE per ricevere da ogni mittente e MPI_ANYTAG per ricevere con qualsiasi tag.

Send e Receive bloccanti e non bloccanti

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	Message sent (receive state unknown)	MPI_SEND	MPI_ISEND
receive	Completes when a message has arrived	MPI_RECV	MPI_IRECV
Synchronous send	Only completes when the receive has completed	MPI_SSEND	MPI_ISSEND
Buffered send	Always completes, irrespective of receiver	MPI_BSEND	MPI_IBSEND
Ready send	Always completes, irrespective of whether the receive has completed	MPI_RSEND	MPI_IRSEND

Send e Receive : un esempio

```
#include <stdio.h>
#include <mpi.h>

void main (int argc, char * argv[])
{
  int      nproc, myid;
  MPI_Status status;
  float a[2];

        MPI_Init(&argc, &argv);
 nproc= MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 myid = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if( myid == 0 ) {
    a[0] = 3.0, a[1] = 5.0;
    MPI_Send(a, 2, MPI_FLOAT, 1, 10, MPI_COMM_WORLD);
  } else if( myid == 1 ) {
    MPI_Recv(a, 2, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
    printf("%d: a[0]=%f a[1]=%f\n", myid, a[0], a[1]);
  }

        MPI_Finalize();
}
```

Comunicazioni collettive

Coinvologono un gruppo di processi

In un programma MPI le funzioni di comunicazione collettiva sono chiamate da tutti i processi appartenenti al comunicatore

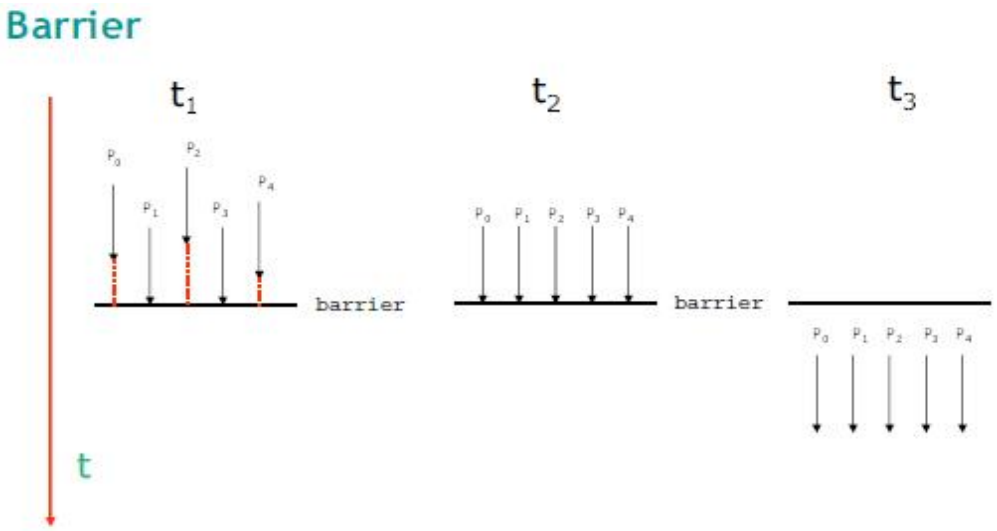
Le comunicazioni collettive non interferiscono con le comunicazioni point-to-point e viveversa

Sono tutte bloccanti

I buffer di ricezione devono avere l'esatta dimensione

Comunicazioni collettive : Barrier

Ferma i processi fino a quando tutti i processi del comunicatore raggiungono la barriera. Si usa per la sincronizzazione.



Fortran:

```
CALL MPI_BARRIER( comm, ierr)
```

C:

```
int MPI_Barrier(MPI_Comm comm)
```

Comunicazioni collettive : Broadcast

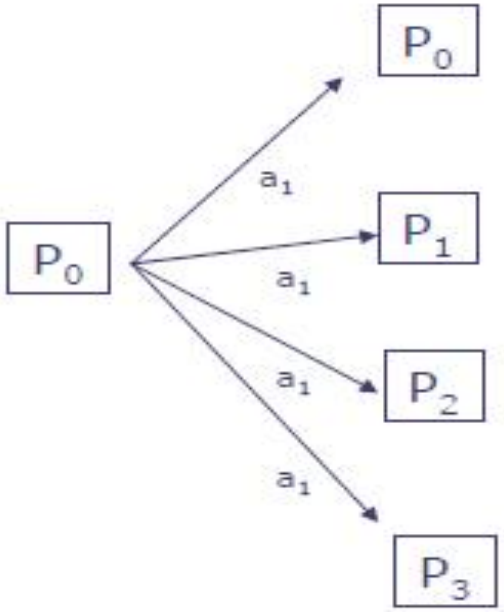
Broadcast: uno a tutti. Gli stessi dati sono inviati dal processo root a tutti gli altri processi che fanno parte del comunicatore.

Fortran:

```
CALL MPI_BCAST(buf, count, type, root, comm, ierr)
INTEGER count, type, root, comm, ierr
Buf array of type type
```

C:

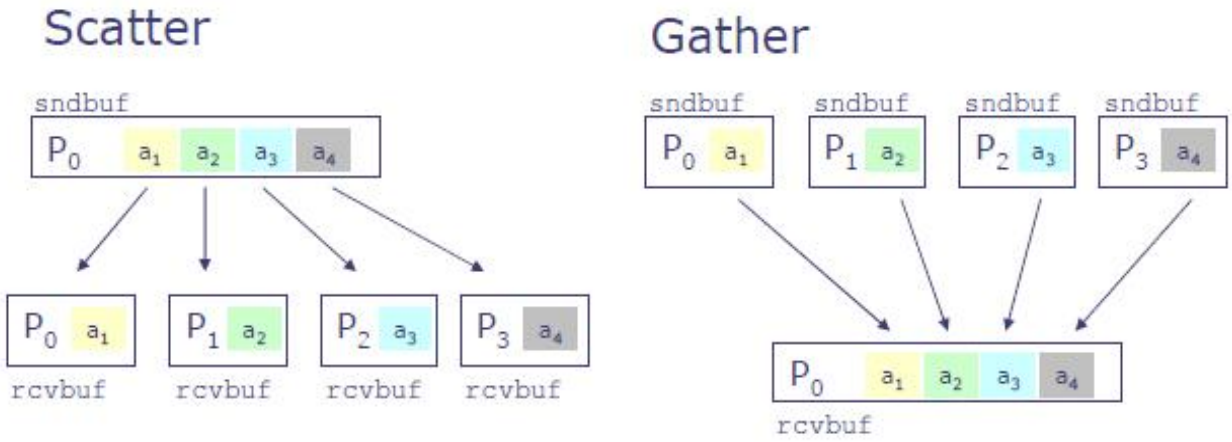
```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatypem int
root, MPI_Comm comm)
```



Comunicazioni collettive: Scatter/Gather

Scatter: uno a tutti. Dati differenti inviati dal processo root a tutti gli altri processi del comunicatore

Gather: tutti a uno. Dati differenti, inviati da tutti i processi del comunicatore, sono raccolti dal processo root.



Comunicazioni collettive: Scatter/Gather

Scatter: uno a tutti. Dati differenti inviati dal processo root a tutti gli altri processi del comunicatore

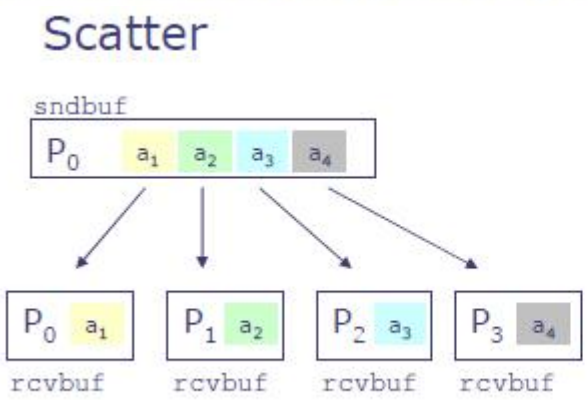
Gather: tutti a uno. Dati differenti, inviati da tutti i processi del comunicatore, sono raccolti dal processo root.

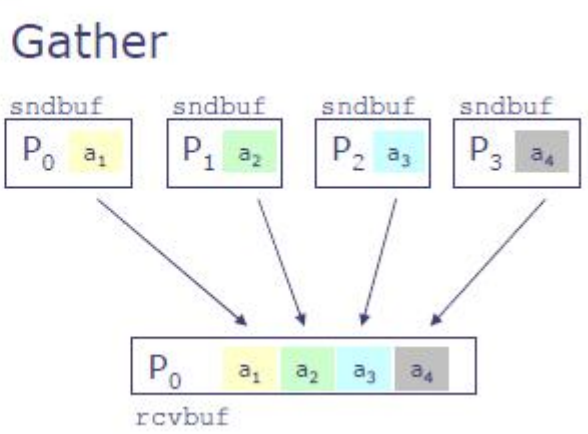
sender receiver

Fortran:

```
CALL MPI_SCATTER(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype, root, comm, ierr)
```

- Arguments definition are like other MPI subroutine
- **sndcount** is the number of elements sent to each process, not the size of **sndbuf**, that should be **sndcount** times the number of process in the communicator
- The sender arguments are significant only at root





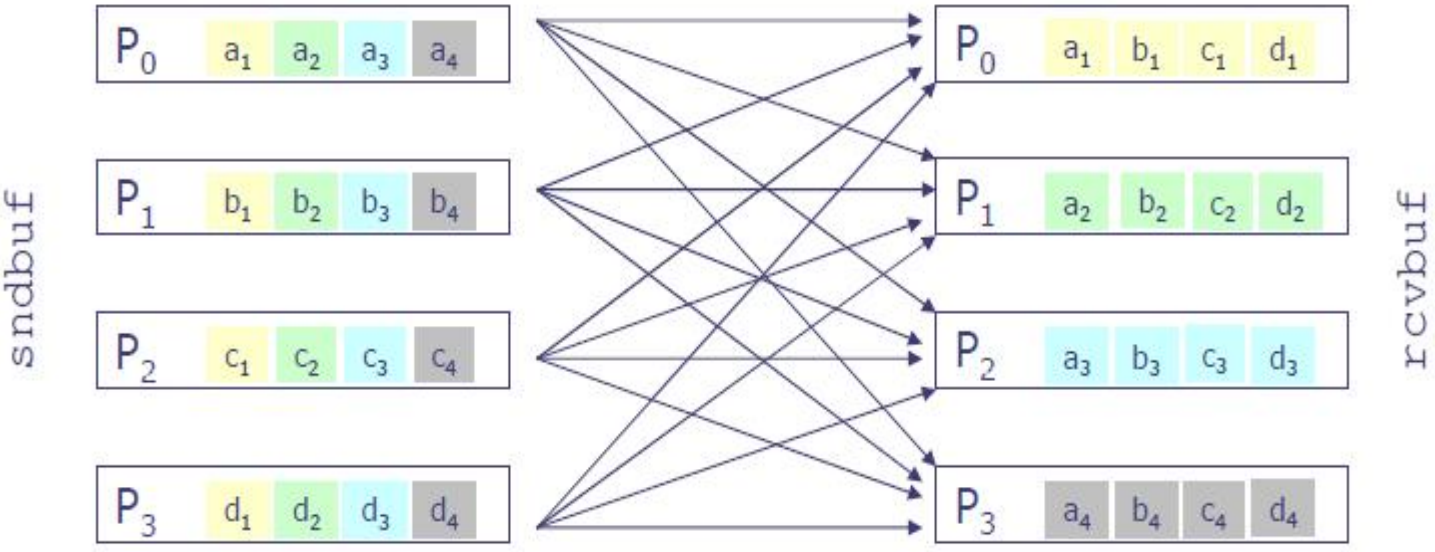
Comunicazioni collettive: Alltoall

Alltoall: tutti a tutti. Ciascun processo invia i suoi dati a tutti gli altri. Molto impegnativa per il network.

Fortran:

sender receiver

```
CALL MPI_ALLTOALL(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype,
comm, ierr)
```



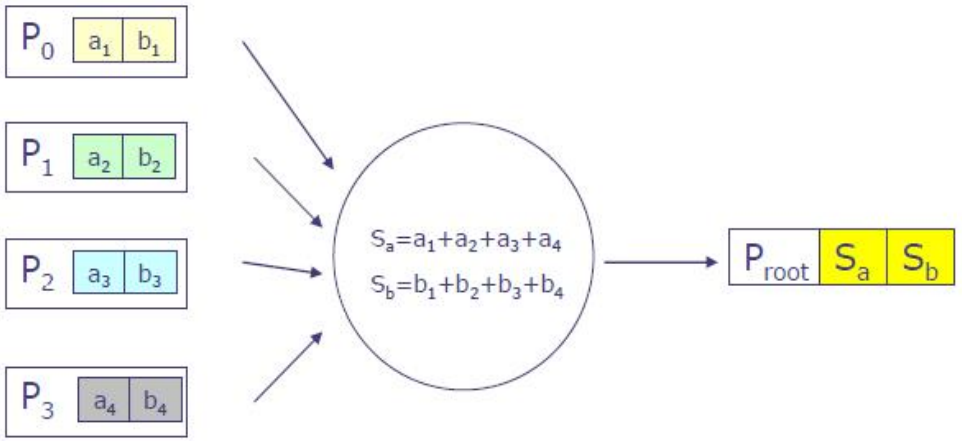
Comunicazioni collettive: Reduction

-raccogliere i dati da ciascun processo
-ridurre i dati a un singolo valore e immagazzinarli sul processo root
-diverse operazioni (somma, prodotto, min, max, operatori logici)

C:

```
int MPI_Reduce(void * snd_buf, void * rcv_buf, int count, MPI_Datatype type,
MPI_Op op, int root, MPI_Comm comm)
```

snd_buf	input array of type type containing local values.
rcv_buf	output array of type type containing global results
Count	(INTEGER) number of element of snd_buf and rcv_buf
type	(INTEGER) MPI type of snd_buf and rcv_buf
op	(INTEGER) parallel operation to be performed
root	(INTEGER) MPI id of the process storing the result
comm	(INTEGER) communicator of processes involved in the operation
ierr	(INTEGER) output, error code (if ierr=0 no error occours)



Valutazione delle prestazioni/1

Un'architettura si dice scalabile se continua ad avere le stesse prestazioni per processore al crescere del numero dei processori e della dimensione del problema.

Due “criteri”: speedup ed efficienza.

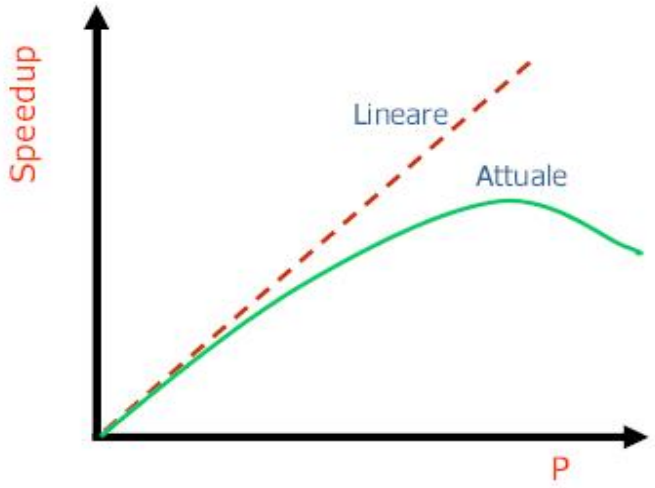
Tempo di esecuzione seriale Ts: tempo di esecuzione del programma su di un singolo processore.

Tempo di esecuzione parallelo Tp: tempo di esecuzione del programma su un dato numero P di processori.

Molti fattori influenzano Tp: l'hardware, l'algoritmo, comunicazione, I/O,

Valutazione delle prestazioni/2

Speedup: S=Ts/Tp



Speedup lineare S=p: corrisponde al caso (ideale) in cui Tp=Ts/P.

Speedup sub-linare S<P: più normale. Dovuto a overhead di inizializzazione, sincronizzazione, comunicazione etc.

Speedup super-lineare S>P: non è frequente. Si può avere perché si presentano condizioni particolari: il programma sfrutta in maniera ottimale le gerarchie di memoria (dati in cache fit), miglior scheduling delle istruzioni, algoritmo particolarmente efficiente.

Valutazione delle prestazioni/3

Un programma scala, per un mumero di processori P, se passando da P-1 a P processori non si osserva una diminuzione dello speedup.

Efficienza: E=S/P

Speedup lineare => E=1 ciascun processore lavora in maniera ottimale.

Lo speedup tende a raggiungere un plateau (saturazione) quando si raggiunge un certo numero di processori. Aumentare ulteriormente il mumero dei processori non migliora lo speedup.

Valutazione delle prestazioni/4

Sia Cs un codice seriale e Ts il suo tempo di esecuzione.

La parallelizzazione non sarà mai totale.

Frazione seriale fs= (tmp. esec. parte seriale su 1 P)/Ts
Frazione parallela fp=(tmp. esec. della parte par. su 1 P)/Ts (fs + fp) = 1

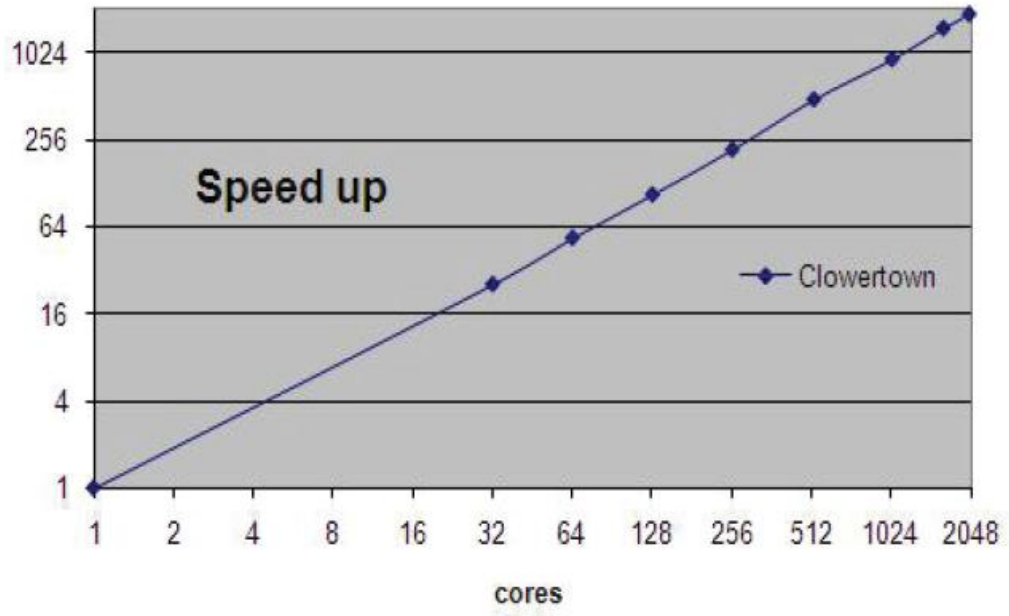
$$Tp = Ts\ fs + (Ts\ fp)/P \text{ (caso ideale)}$$

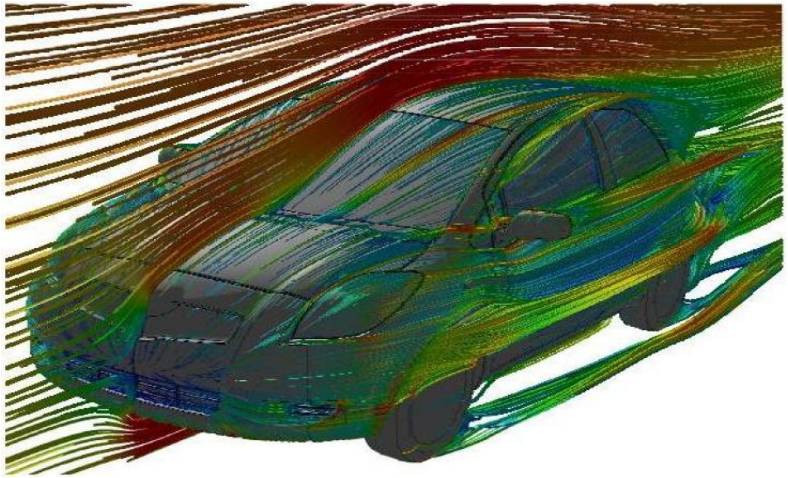
$$S = Ts/Tp = 1/[fs+(fp/P)] \quad P \to \infty \quad S \to 1/fs$$

"Legge" di Amdhal: se un codice parallelo ha una frazione seriale (non parallelizzabile) lo speedup ha un limite superiore.

Per es. se il 5% del codice è sequenziale lo speedup non potrà mai superare 20 anche aumentando indefinitamente il numero dei processori.

Valutazione delle prestazioni/5





cores	1	32	64	128	256	520	1024	1600	2000
speed-up	1	25.5	53.4	106.2	217.2	482.2	913.3	1491.4	1867.5
efficiency	1	0.80	0.83	0.83	0.85	0.93	0.89	0.93	0.93
time	62.7 days	2days 11h	1 day 4h	14.2 hours	7 hours	3h 7min	1h 38min	1 hour	48 min

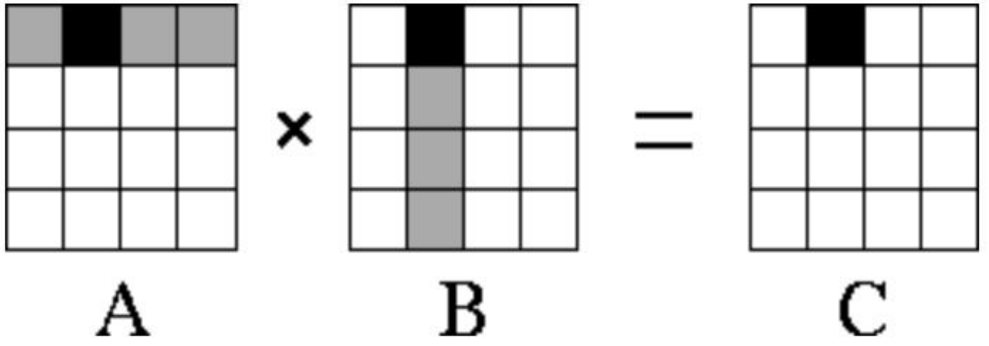
Moltiplicazione di matrici MPI

- Il processo master invia una riga di A e una colonna di B al processo worker

```
MPI_Send(&a,…); MPI_Send(&b,..);
```

- Il worker riceve e calcola l'elemento di C

```
MPI_Recv(&a,…); MPI_Recv(&b,..);

  for (k=0; k<NCB; k++)
      for (i=0; i<rows; i++)
      {
        c[i][k] = 0.0;
        for (j=0; j<NCA; j++)
          c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
```



- Il worker invia il risultato al master

```
MPI_Send(&c,..);
```

- Il master riceve , stampa il risultato e chiude tutti i processi

```
MPI_Recv(&c,..);
```

Moltiplicazione di matrici: risultati

- **Compilazione e lancio programma seriale**

icc –o ser_mm_c ser_mm.c

time ./ser_mm_c

- **Compilazione e lancio programma parallelo**

mpicc –o mpi_mm_c mpi_mm.c

time mpirun –np numero_core ./mpi_mm_c

Dimensione matrice N	Tempo seriale	Tempo 4 core	Tempo 8 core	Tempo 16 core
1000	3s	2.6s	2s	3s
2000	24s	12s	8s	9s
5000	6m 30s	2m 47s	2m 4s	2m 11s

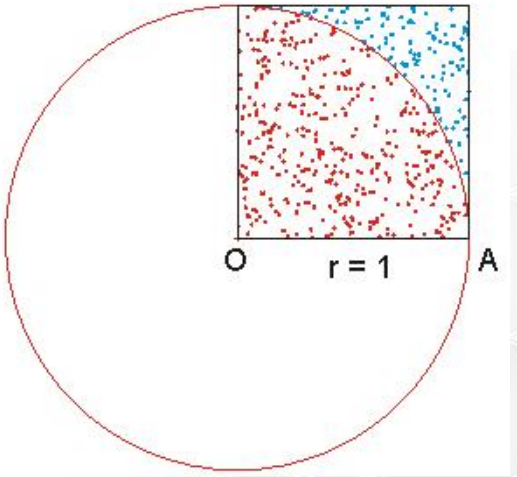
- **Al crescere di N il programma parallelo è più vantaggioso**

Calcolo di Pi col metodo Monte Carlo

- Area cerchio/Area quadrato = Pi/4
- Si generano N punti a caso nel quadrato
- Si conta il numero Nc di punti che cadono nel cerchio
- Pi≈4 Nc/ N
- Ciascun processo genera N /num. proc. punti e conta (mysum) quanti di essi capitano nel cerchio
- Ciascun processo invia al master la sua conta parziale (mysum); il master le somma (MPI_SUM), calcola Nc e quindi Pi

MPI_Reduce(&mysum,&Nc,..,..,MPI_SUM,master_rank,..);

	Tempo seriale	Tempo 4 core	Tempo 8 core	Tempo 16 core
$N= 10^{10}$	4m 57s	1m 15s	43s	33s



Ricerca condizionale

- Trovare gli interi J tra A e B che soddisfano la relazione F(J)=C dove F è una funzione assegnata e C una costante
- Programma seriale: esamina uno per volta tutti gli interi J tra A e B e controlla se ciascuno di essi soddisfa F(J)=C
- Programma parallelo (p processi): ciascun processo esamina un sottoinsieme di [A,B]

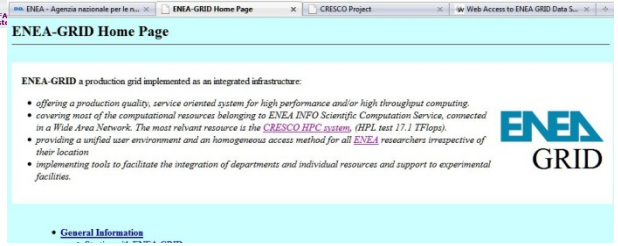
  id=0 [A, A+p, A+2p,…]

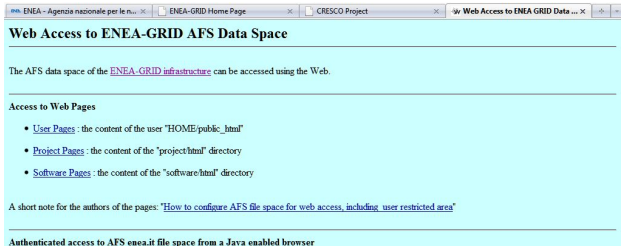
  id=1 [A+1, (A+1) + p, (A+1) + 2p,…]

Tempo seriale	Tempo 4 core	Tempo 8 core	Tempo 16 core
58s	20s	12s	7s

Riferimenti

- **www.enea.it**
- **www.cresco.enea.it**
- **www.eneagrid.enea.it**
- **www.afs.enea.it**

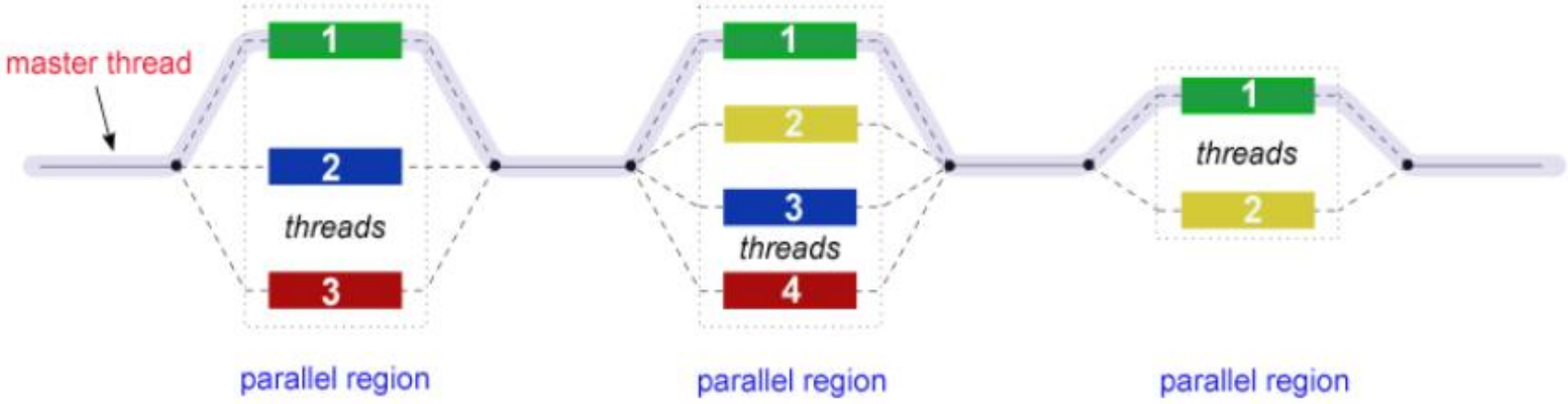




Back slides

OpenMP (Open Multi Processing)

- Interfaccia di programmazione basata sul multithreading utilizzata su sistemi a memoria condivisa.
- Programmazione basata su « direttive » al compilatore, librerie run time, variabili di ambiente.
- Regioni parallele Fork-Join.



OpenMP – Processi e thread/1

Processo: è un programma in esecuzione.

Istruzioni eseguite in sequenza come specificato nel testo del programma.

Il SO può gestire l'esecuzione di più processi.

Contex switch: l'uso della CPU viene commutato da un processo ad un altro (oneroso).

Il SO associa delle risorse a un processo: connessioni di rete, memoria, disco etc.

Processi diversi accedono ad aree di memoria diverse.

Il processo è rappresentato da:

- **codice** (*text*) del programma eseguito
- **Dati:** variabili globali
- **Program Counter**
- Alcuni **registri** di CPU
- **Stack**: parametri, variabili locali a funzioni/procedure

OpenMP – Processi e thread/2

Un “thread” (o processo leggero) è un flusso di esecuzione che vive all'interno di un processo.

Thread= {PC, registri, stack}

Un thread condivide con il processo e con altri thread (multithreading) dati, codice e risorse.

Ecco perché su un singolo nodo l'accesso alla memoria condivisa è più veloce rispetto al caso di più processi!

Il context switch tra i thread è più veloce rispetto a quello tra processi.

Poiché i thread condividono i dati bisogna gestire bene i conflitti!

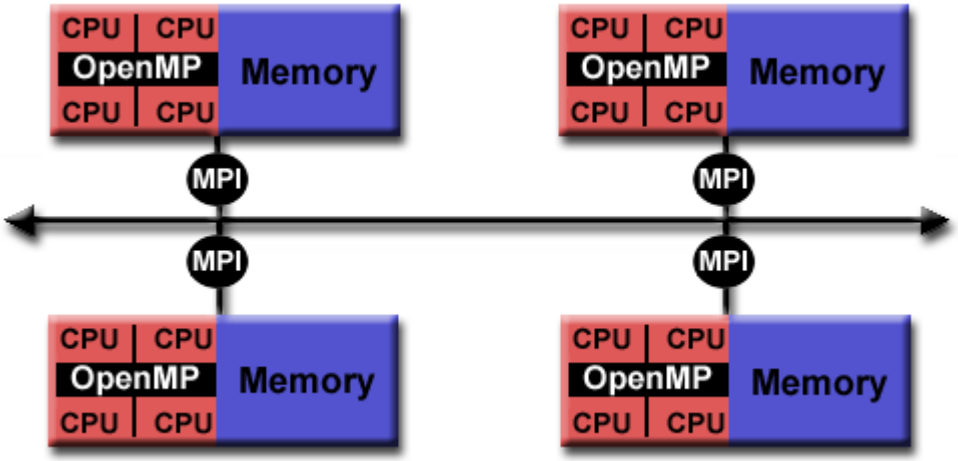
OMP – Tipica struttura del programma

```
#include <omp.h>

main ()  {

int var1, var2, var3;

Serial code
      .
      .
      .

Beginning of parallel section. Fork a team of threads.
Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
  {

  Parallel section executed by all threads
            .
  Other OpenMP directives
            .
  Run-time Library calls
            .
  All threads join master thread and disband

  }

Resume serial code
      .
      .
      .

}
```

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
int nthreads, tid;
/* Fork a team of threads giving them their own copies of
variables */
#pragma omp parallel private(nthreads, tid)
 {
 /* Obtain thread number */
 tid = omp_get_thread_num();
 printf("Hello World from thread = %d\n", tid);
 /* Only master thread does this */
 if (tid == 0)
  {
  nthreads = omp_get_num_threads();
  printf("Number of threads = %d\n", nthreads);
  }
 } /* All threads join master thread and disband */
}
```

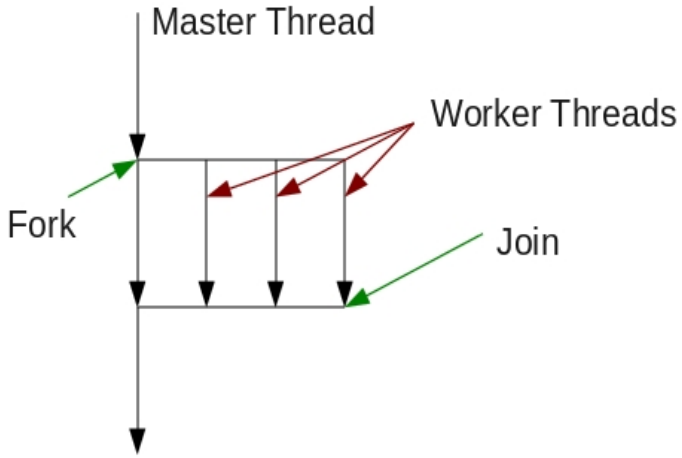
Programmazione Ibrida MPI + OpenMP

Su architetture distribuite in cui ogni nodo di calcolo è dotato di più core che condividono una stessa memoria è possibile implementare una modalità di programmazione ibrida che combini le funzioni di scambio di messaggi tra nodi diversi fornita da MPI a quelle del « Multiprocessing » su memoria condivise fornite da OpenMP



Moltiplicazione di matrici - OpenMP

```
/*** Regione parallela  ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
 {
 tid = omp_get_thread_num();
 if (tid == 0)
  {
  nthreads = omp_get_num_threads();
  printf("Starting matrix multiple example with %d threads\n",nthreads);
  printf("Initializing matrices...\n");
  }
 /*** Initialize matrices ***/
#pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
  for (j=0; j<NCA; j++)
   a[i][j]= i+j;
 ……..
#pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
  {
 //  printf("Thread=%d did row=%d\n",tid,i);
  for(j=0; j<NCB; j++)
   for (k=0; k<NCA; k++)
    c[i][j] += a[i][k] * b[k][j];
  }
 }  /*** Fine della regione parallela***/
```



Un solo processo che fa una « fork »: l'accesso alla memoria condivisa è più veloce rispetto al caso in cui si hanno più processi indipendenti.

Moltiplicazione di matrici - OpenMP

- **Compilazione e lancio del programma su singolo nodo shared memory con 16 core**

icc –o omp_mm_c –openmp –parallel omp_mm.c

export OMP_NUM_THREADS=numero_thread

time ./omp_mm_c

Dimensione matrice N	Tempo seriale	Tempo 4 thread	Tempo 4 core (MPI)	Tempo 8 thread	Tempo 8 core (MPI)	Tempo 16 thread	Tempo 4 core (MPI)
1000	3.04s	0.82s	2.6s	0.61s	2s	0.41s	3s
2000	23s	9s	12s	5s	8s	3s	9s
5000	6m 34s	2m 5s	2m 47s	1m 13s	2m 4s	58s	2m 11s

Sul singolo nodo l'accesso alla memoria condivisa è più veloce nel caso multithreading!

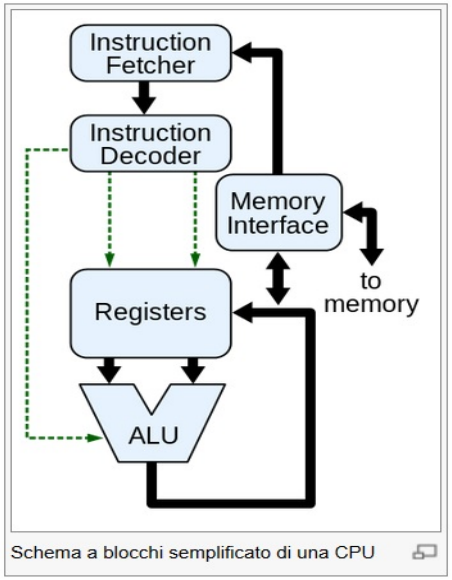
La CPU

CPU (Central Processing Unit) è il microprocessore principale che coordina tutte le unità di elaborazione presenti sul chip.

Esegue le istruzioni di un programma dopo averlo caricato in memoria.

Architettura di Von Neumann: dati e istruzioni risiedono nella stesa memoria.

Architettura di Harvard: dati e istr. in memorie separate.



Schema a blocchi semplificato di una CPU

La CPU contiene:

- CU (Central Unit): legge dalla memoria le istruzioni, i dati per eseguire le istruzioni, esegue l'istruzione e scrive il risultato nella memoria centrale o nei registri.
- ALU (Arithmetic Logic Unit): esegue le op. logiche e aritmetiche
- Registri: piccole memorie interne molto veloci.
  - -Program Counter: indirizzo di memoria della prossima istr. da eseguire
  - -Flag: inieme di bit che segnalano lo stato della CPU e alcune info sul risultato dell'ultima op. eseguita
- FPU (Floating Point Unit): esegue le op. in virgola mobile
- MMU (Memory Management Unit): traduce gli indirizzi di memoria logici in fisici, supporta meccanismi di protezione della memoria e/o la mem. virtuale.