UNIVERSITÀ DI PARMA
DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE ED INFORMATICHE
Corso di Laurea in Informatica

Progettazione di programmi paralleli

Programmazione parallela e HPC - a.a. 2023/2024

Roberto Alfieri

Programmazione Parallela e HPC: sommario

PARTE 1 - INTRODUZIONE

PARTE 2 – SISTEMI PER IL CALCOLO AD ALTE PRESTAZIONI

PARTE 3 – PERFORMANCE DELL'HARDWARE

PARTE 4 – PROGETTAZIONE DI PROGRAMMI PARALLELI

PARTE 5 – PROGRAMMAZIONE A MEMORIA CONDIVISA CON OPENMP

PARTE 6 – PROGRAMMAZIONE A MEMORIA DISTRIBUITA COM MPI

PARTE 7 – PROGRAMMAZIONE GPU CON CUDA

Parallel computing

Parallel computing è la tecnica di programmazione necessaria per scomporre il carico computazionale in **Task** che dovranno essere distribuiti ed eseguiti sui diversi livelli di parallelismo dei sistemi HPC.

La decomposizione può essere a livello di domini o di funzioni:

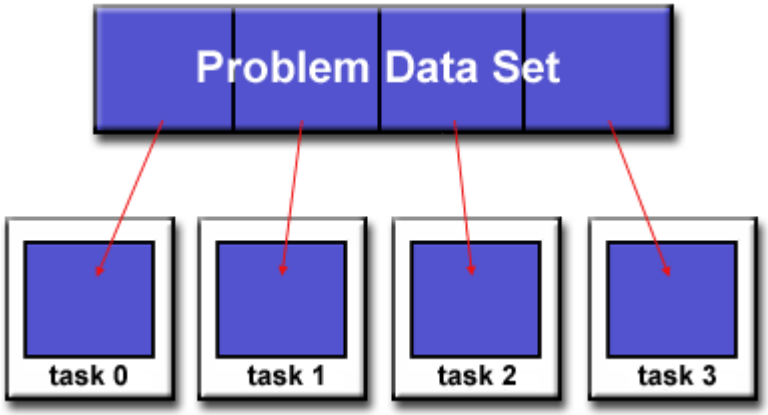
1) **Decomposizione di dominio (data parallel)**
Si applica se i dati sono organizzati in strutture regolari (tipicamente array e matrici)
I dati vengono suddivisi in strutture più piccole di uguale dimensione e assegnati a diverse unità computazionali.

2) **Decomposizione funzionale (task parallel)**
Distribuzione delle funzioni tra diversi task: che eseguono operazioni distinte

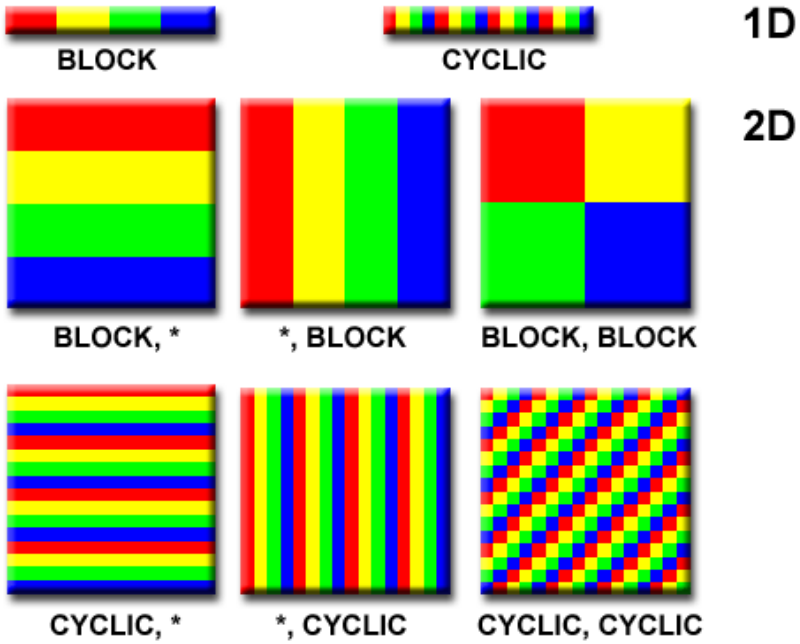
Riferimenti: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Decomposizione di dominio

Si utilizza quando è necessario elaborare un data set di grandi dimensioni con dati strutturati che vengono suddivisi in sottodomini di dimensioni ridotte.
Ogni sottodominio viene elaborato da un task specifico che svolge le stesse operazioni degli altri task ma su dati diversi.



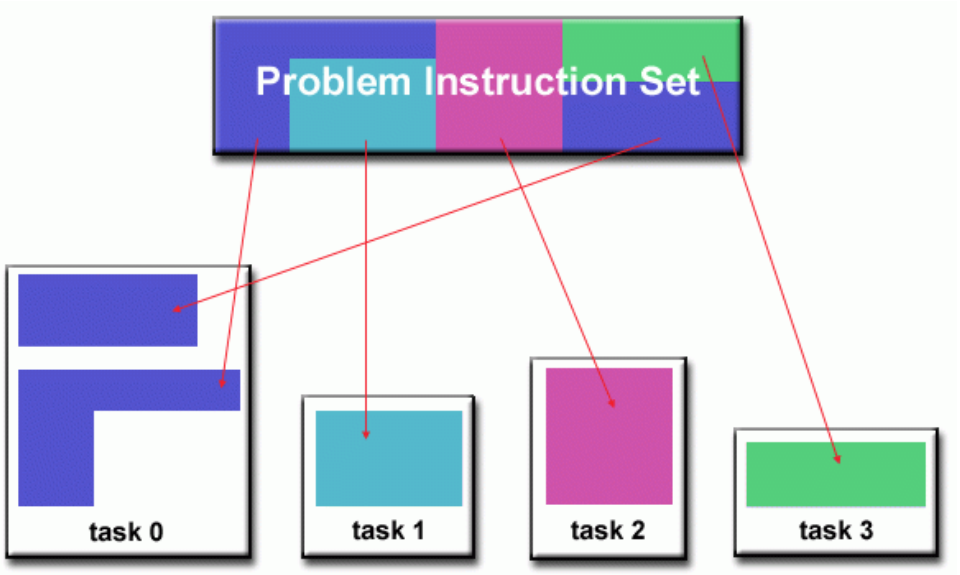
Il dominio dei dati ha una propria dimensione (1D, 2D, ecc) e la decomposizione può avvenire in diversi modi (vedi figura).



Decomposizione funzionale

Insieme di elaborazioni differenti ed indipendenti
Decompongo il problema in base al lavoro che deve essere svolto, individuando i diversi task che possono essere elaborati concorrentemente ed eseguiti su diversi processi.

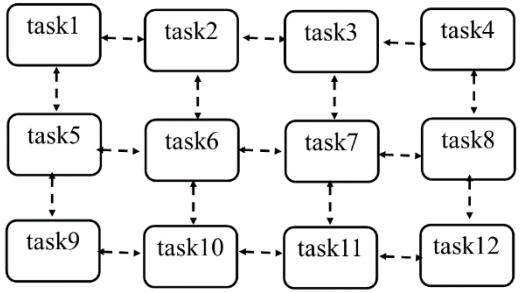
PRO: scalabile con il numero di elaborazioni indipendenti
CONTRO: vantaggiosa solo per elaborazioni sufficientemente complesse



Comunicazione nelle applicazioni parallele

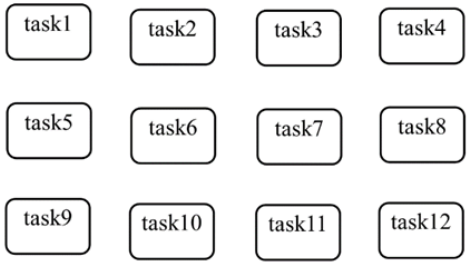
La comunicazione tra i task riguarda principalmente i modelli di parallelismo a memoria distribuita.

Se la dinamica dell'applicazione richiede iterazioni con scambio dati ad ogni iterazione l'applicazione parallela è detta **«tightly coupled»**
Esempio: la diffusione del calore nello spazio è parallelizzabile a decomposizione di dominio ma il nuovo valore di temperatura di un sottodominio dipende dalla temperatura dei sottodomini adiacenti



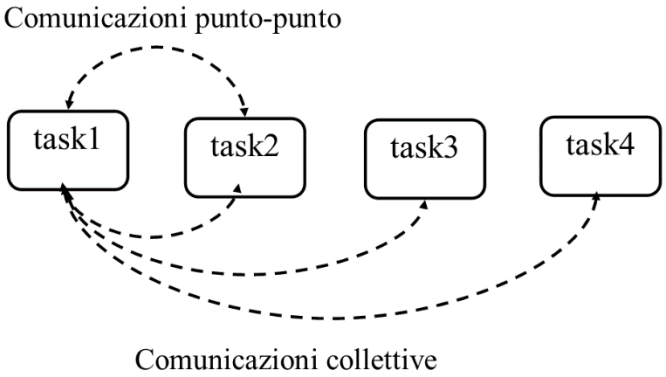
Applicazioni parallele che comunicano raramente sono dette **«loosely coupled».**
Ad esempio comunicazioni all'inizio dell'esecuzione per distribuire i compiti e alla fine per raccogliere i risultati.

Se non abbiamo comunicazione tra i task il problema è detto «**embarassingly parallel»**
Ad esempio lo scambio del colore dei pixel di una immagine tra bianco e nero è parallelizzabile a decomposizione di dominio ma non è richiesta comunicazione tra i task

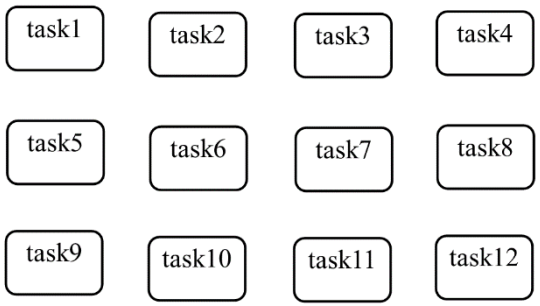


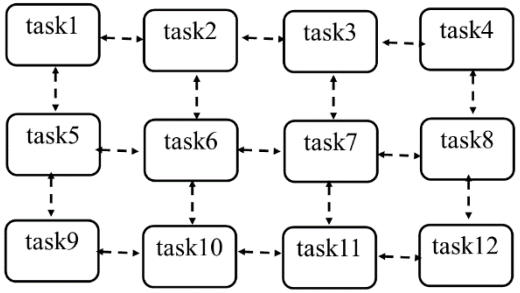
Comunicazione tra i Task

La comunicazione tra i task riguarda principalmente i modelli di parallelismo a memoria distribuita e può essere:
- punto-punto, tra due task
- collettiva, che coinvolge tutti i task



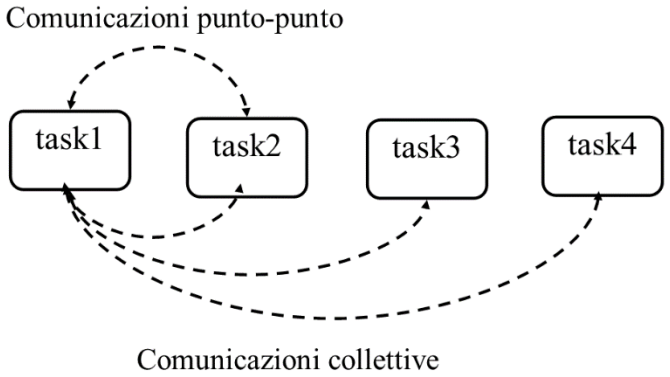
La necessità di comunicazione tra i task dipende dal tipo di problema.
Esempi:
- lo scambio del colore dei pixel di una immagine tra bianco e nero è parallelizzabile a decomposizione di dominio ma non è richiesta comunicazione tra i task
- La diffusione del calore nello spazio è parallelizzabile a decomposizione di dominio ma il nuovo valore di temperatura di un sottodominio dipende dalla temperatura dei sottodomini adiacenti.



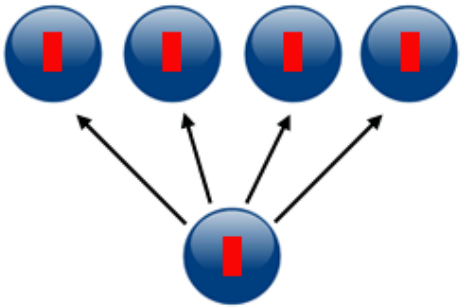


Comunicazioni punto-punto e collettive

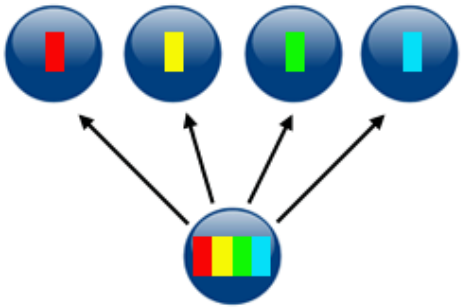
La comunicazione può essere:
- punto-punto, tra due task
- collettiva, che coinvolge tutti i task



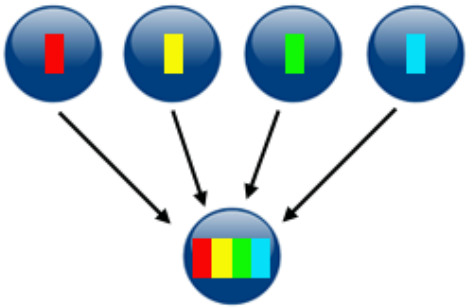
Le comunicazioni collettive possono avere diverse varianti:



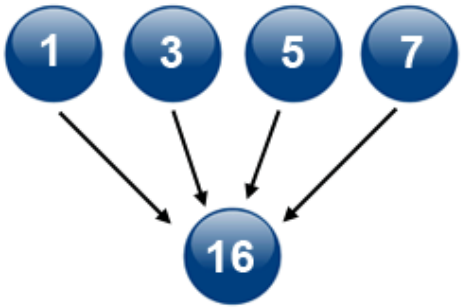
broadcast



scatter



gather



reduction

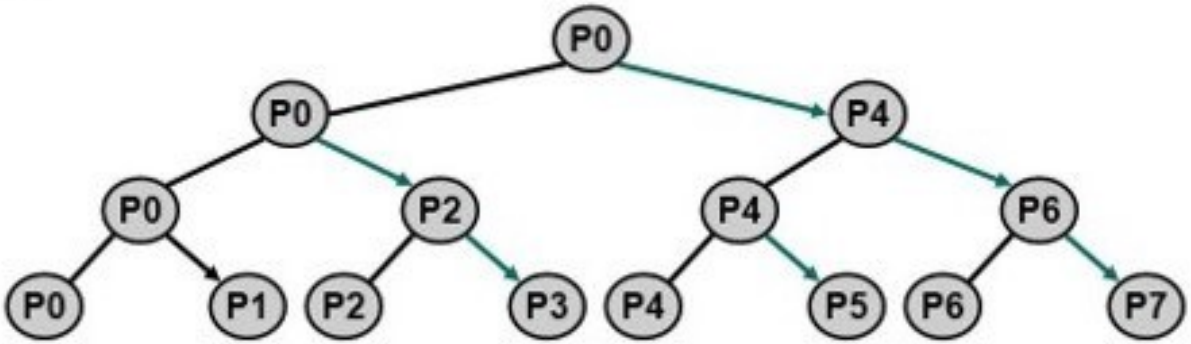
Costo della Comunicazione

Ogni messaggio inviato richiede un tempo che influisce sulle prestazioni, soprattutto per le applicazioni tightly coupled.

Pe comunicazioni punto-punto: Tmess= Tlat+ M/Band
Tlat è la latenza (secondi), M numero byte del messaggio, Band è la Bandwidth (Byte/s)

Comunicazioni **coll**ettive:
Tcoll= Tmess x (P-1) (P è il numero di Task)

Per le comunicazioni collettive se utilizziamo la strategia divide-et-impera (vedi figura) possiamo migliorare in costo della comunicazione in O(lg(P))



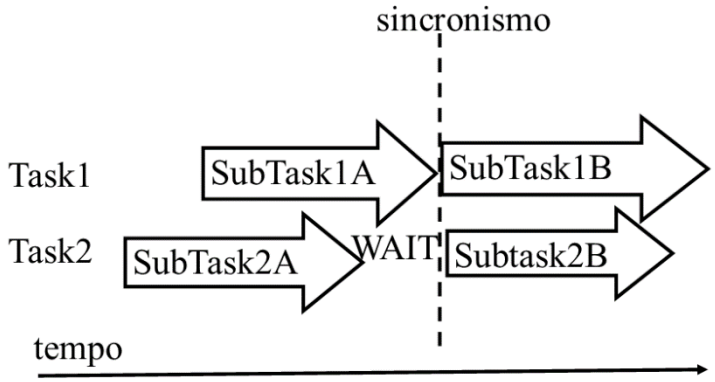
If we have ***p*** processes,
this procedure allows us to distribute the input data In $log_2(p)$ stages,
rather than p-1 stages, which, if p is large, is a huge savings.

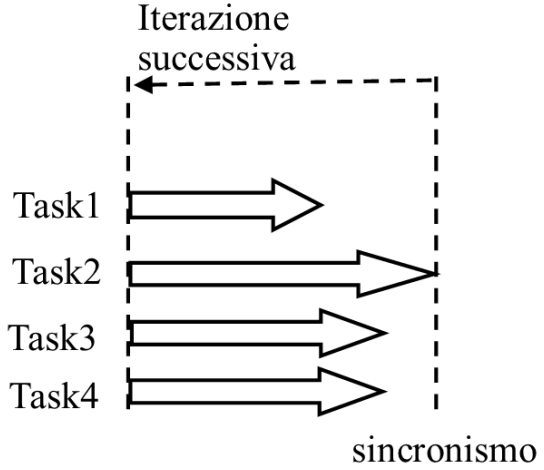
Sincronizzazione

I task possono interagire tra loro anche per dipendenze tra sezioni di codice o dati.
La necessità della sincronizzazione riguarda sia i modelli a memoria distribuita che a memoria condivisa e può riguardare 2 (o più) task oppure tutti i task del dominio.

Esempi:

- il Task 2 può procedere solo quando il task 1 ha raggiunto un determinato obiettivo
- In un programma tightly coupled tutti i task iterano la stessa funzione su dati diversi e possono procedere all'iterazione N+1 solo quando tutti hanno completato l'iterazione N



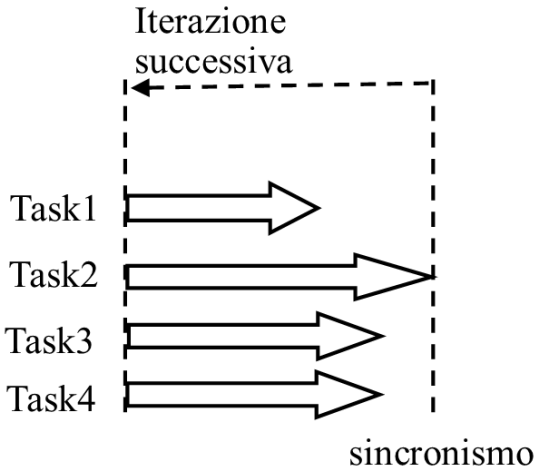


- Accesso esclusivo: l'accesso di un task ad un dato o ad una sezione di codice può avvenire solo in modo esclusivo; un solo task può accedere, gli altri devono attendere.

Strumenti per la Sincronizzazione

Gli strumenti per la sincronizzazione sono
- **Lock e semafori**, nei sistemi a memoria condivisa. Ad esempio il task A si blocca sul Semaforo X in attesa che venga sbloccato.
- **Scambio di messaggi**, nei sistemi a memori distribuita. Ad esempio il task A attende un messaggio dal task B per poter procedere.

Le **barriere (barrier):** implicano il coinvolgimento di **tutti i task del dominio**, sia a memoria. distribuita che condivisa; chi raggiunge la barriera deve attendere che tutti i task del dominio l'abbiamo raggiunta

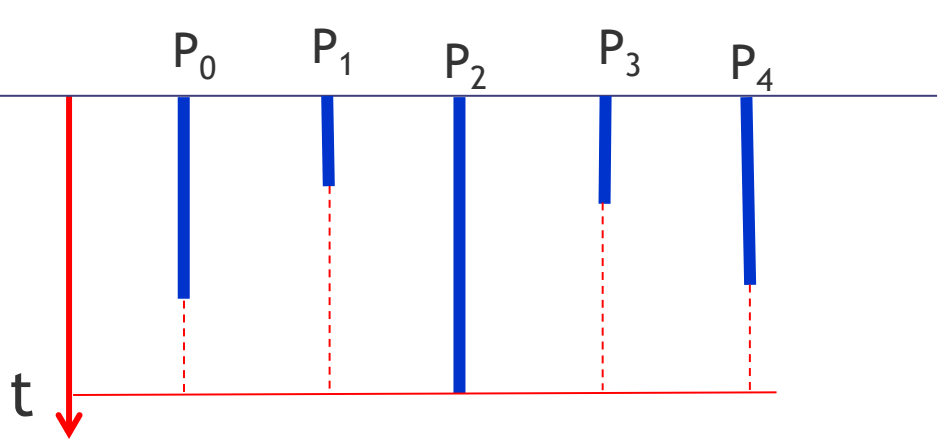


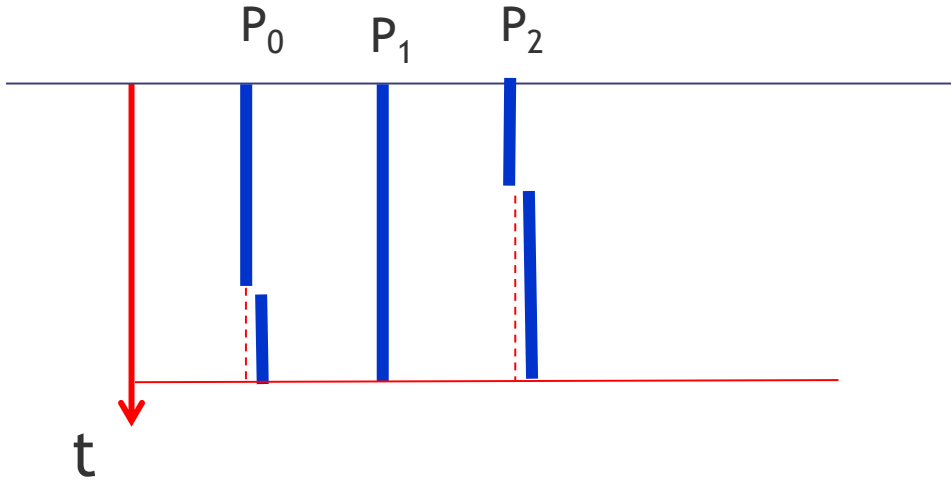
Il tempo di inattività di un task (dovuto a dipendenze o altro) è detto **tempo di Idle**

Il sincronismo se non è sotto controllo può essere causa di un degrado delle performance.

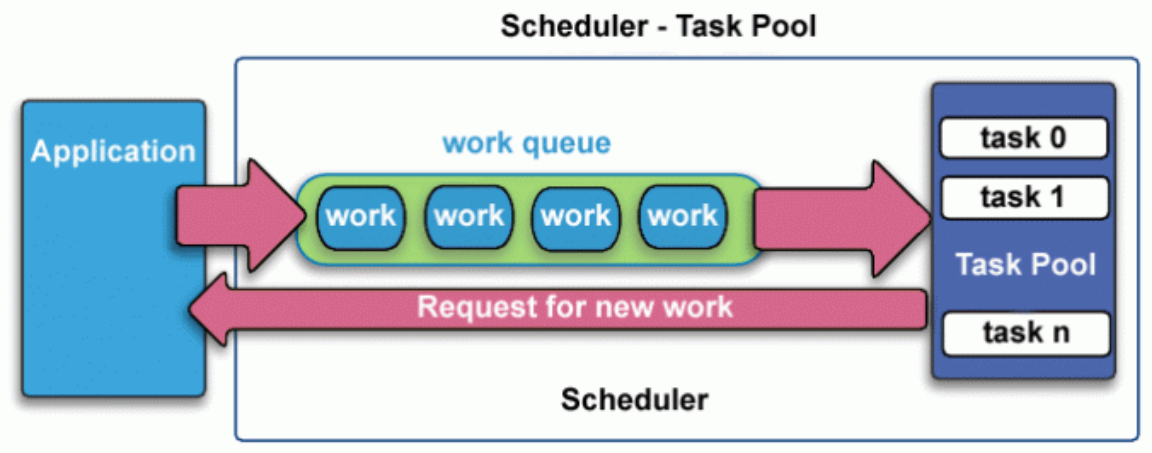
Bilanciamento del carico

Il bilanciamento del carico (load balancing) è una tecnica di progetto con l'obiettivo di di distribuire il lavoro in modo da minimizzare il tempo di Idle dei processi.





Una tecnica comune per bilanciare il carico è il modello master-slave (detto a volte manager-worker), in cui un task (master) ha il compito di suddividere il lavoro in piccoli task e gestire lo scheduling dinamico dei task verso un pool di Slaves

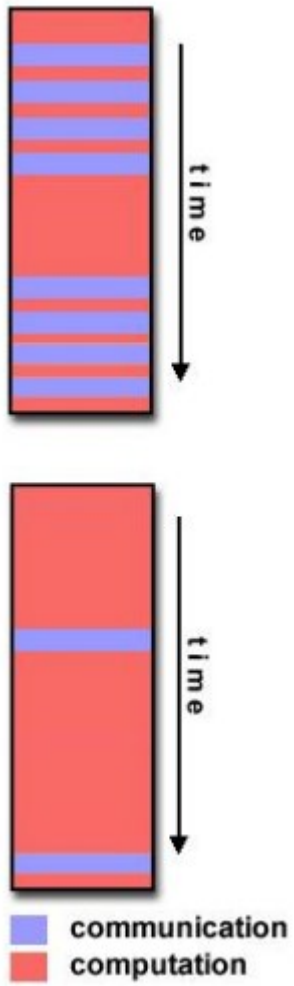


Granularità

La decomposizione del problema può avvenire con diverse granularità:

Parallelismo a grana fine (fine grain)
- Utile per bilanciare il carico e diminuire l'overhead di sincronismo
- Potrebbe portare ad un aumento delle comunicazioni e del conseguente overhead

Parallelismo a grana grossa (coarse grain)
- Migliora il rapporto tra calcolo e le comunicazioni
- Può essere difficile bilanciare il carico



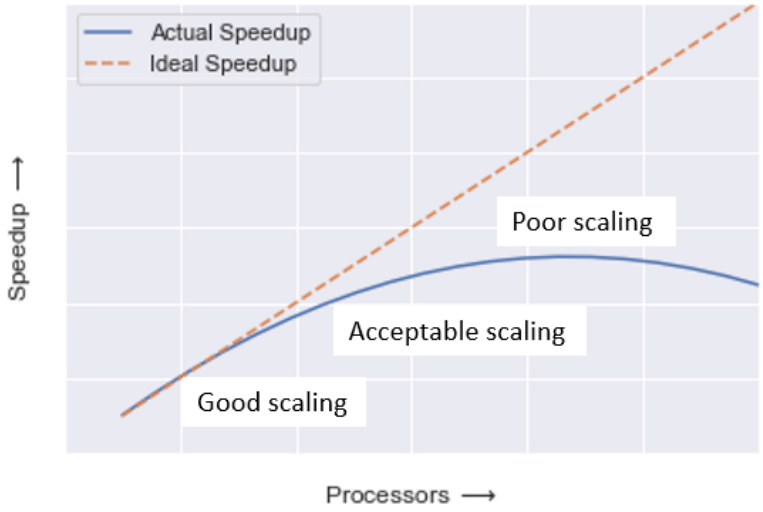
Performance del parallelismo

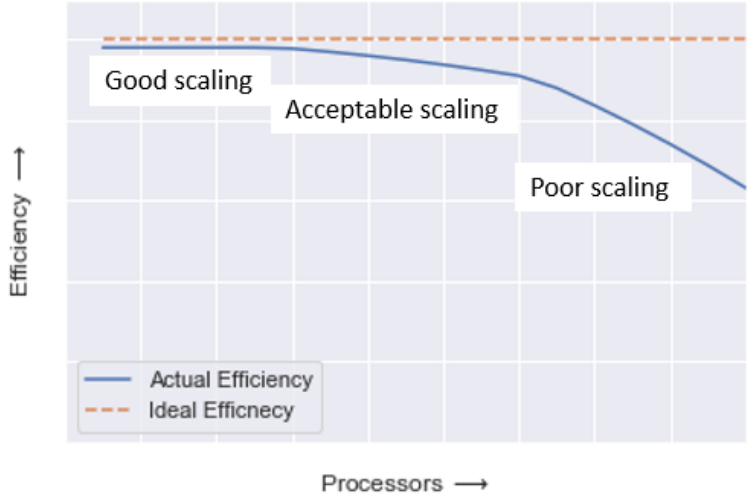
Con il termine **Speedup** si intende la misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato: SpeedUp = Tseriale / Tparallelo, che ne caso ideale coincide con il numero di processori

L'efficienza (**efficiency**) è il rapporto tra Speedup e numero di processori P.

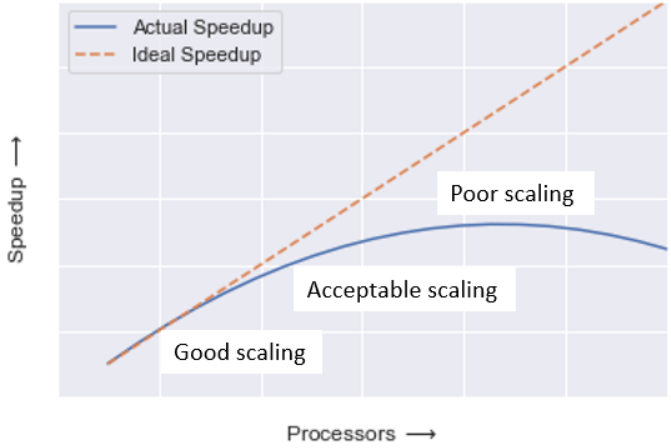
E = Speedup/P nel caso ideale vale 1.

Scalabilità è la capacità del programma parallelo di mantenere una crescita proporzionata dello Speedup al crescere delle unità di processamento.





Scalabilità



Scalabilità è la capacità del programma parallelo di mantenere una crescita proporzionata dello Speedup al crescere delle unità di processamento

Strong Scaling

Misura l'efficienza dell'applicazione al crescere del numero di processori mantenendo fissa la dimensione complessiva del problema.

Weak Scaling

Misura l'efficienza dell'applicazione al crescere del numero di processori mantenendo fissa la dimensione del problema assegnata ad ogni processore.

Overhead

Le limitazioni della scalabilità sono dovute a Overhead introdotti dalla parallelizzazione.

Tempi di Overhead

La parallelizzazione di un programma seriale può introdurre dei tempi di Overhead che incidono sullo Speedup: Speedup = Ts / (Ts/P + Toh)

dove Ts è il tempo seriale, P il numero di Processori e Toh il tempo di overhead.

Overhead principali:

- **Tempo speso per le comunicazioni**
  - -- Tcomm = Σ (Tmess) = Σ (Tlat+ M/Band)
  - -- può essere rilevante per la programmazione a memoria distribuita
- **Tempo di Idle**
  - -- può essere rilevante per programmi sbilanciati
- **Tempo di avvio e chiusura dei task paralleli** (Processi e/o Thread)
  - -- può essere rilevante nella programmazione a memoria condivisa

Legge di Amdahl

La **legge** di **Amdahl** distingue in un programma seriale la porzione parallelizzabile da quella non parallelizzabile Ts=Tp+Tnp e stabilisce un limite massimo allo Speedup.

Samdahl = Tseriale/Tparallelo = (Tnp+Tp) / (Tnp + Tp/P) = 1/(Qnp+Qp/P)

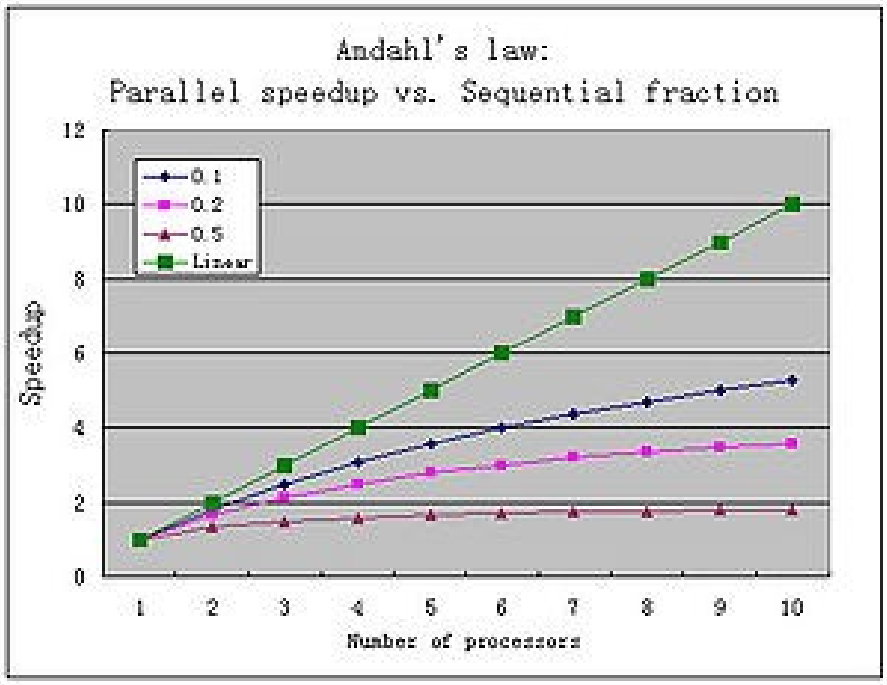
Tp è il tempo parallelizzabile

Tnp è il tempo non parallelizzabile

In termini di quote percentuali:

Qnp= Tnp/(Tnp+Tp) Qp=Tp/(Tnp+Tp)

dove Qnp+Qp =1



Sreal è lo Speedup reale che tiene conto di Amdahl e Overhead:

Sreal = (Tnp+Tp) / (Tnp + Tp/P + Toh)

Programma Parallelo

E' un programma in grado di **suddividere l'algoritmo in diversi task** (processi, threads, ..) distribuiti sulle diverse unità di processamento e coordinati tra loro per realizzare un obiettivo computazionale complessivo.
L'esecuzione di processi di calcolo non sequenziali richiede:

- un calcolatore non sequenziale (in grado di eseguire un numero arbitrario di operazioni contemporaneamente)
- un linguaggio di programmazione che consenta di descrivere formalmente algoritmi non sequenziali (**parallelismo esplicito**) e/o
- un compilatore in grado di parallelizzare automaticamente parti del programma sequenziale (**parallelismo implicito**)

La tassonomia dei calcolatori paralleli prevede diverse architetture hardware (SIMD, MIMD a memoria condivisa o separata, GPU), ciascuna con il proprio modello di programmazione.

Riferimenti: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Parallelismo automatico, guidato e manuale

Partendo dal programma seriale ci sono diversi modi per arrivare al programma parallelo:

- **Automatico** : il compilatore analizza il sorgente ed individua possibili parallelizzazioni.
  I loop (for, while) sono costrutti più adatti per il parallelismo automatico.

- **Direttive per il compilatore**: attraverso direttive il programmatore può dare indicazioni sulle parti di codice da parallelizzare e come farlo.
  Le direttive vengono ignorate da un compilatore che non riconosce le direttive.
  Esempio: openMP

- **Esplicita:** occorre individuare manualmente i task e programmare esplicitamente l'interazione tra i task (esempio MPI)

Esempio di parallelismo automatico, guidato o esplicito:

Programmazione delle istruzioni SIMD

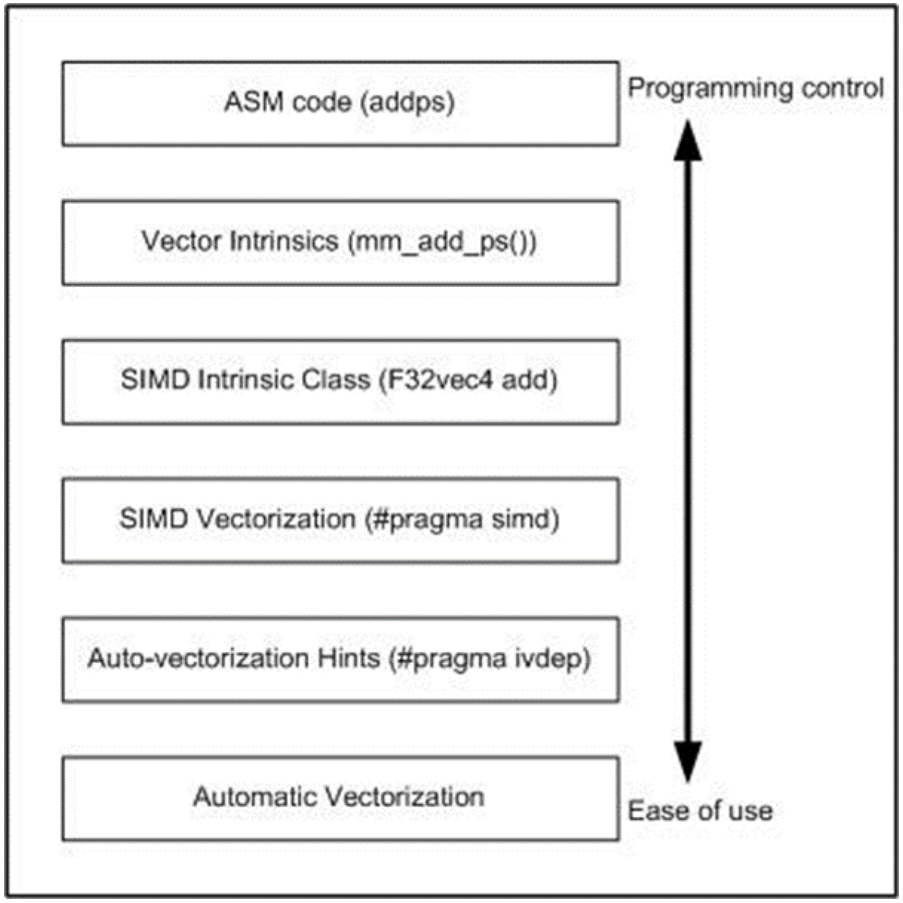
La vettorizzazione è una tecnica che consente di effettuare in parallelo la stessa operazione su tutti gli elementi di un vettore. Viene realizzata mediante architetture SIMD (Single-Instruction Multiple-Data).

La programmazione vettoriale può essere:

- **Automatica** da parte del compilatore.

- Guidata dal programmatore, ad esempio
- tramite direttive “#pragma”

- Esplicita attraverso le **“intrinsics”.**
In questo caso le istruzioni vettoriali possono essere usate nei programmi come se fossero chiamate a funzioni mediante gli Intrinsics, cioè particolari costrutti che il compilatore riconosce e mappa direttamente su codice assembly.



Approfondimenti https://www.moreno.marzolla.name/teaching/HPC/SIMD.pdf

Esempio di vettorizzazione automatica: Autovettorizzazione

I compilatori possono individuare e vettorizzare parti di codice.

Ad esempio con gcc compiler

```
#include <stdio.h>
#define n 1024

int main () {
  int i, a[n], b[n], c[n];

  for(i=0; i<n; i++) { a[i] = i; b[i] = i*i; }
  for(i=0; i<n; i++) c[i] = a[i]+b[i];

  printf("%d\n", c[1023]);
}
```

gcc -O3 -march=native –ffast-math -c prog.c

Riferimenti

Esempio di vettorizzazione guidata : direttive openMP

OpenMP è un API per la creazione di applicazioni parallele su sistemi a memoria condivisa mediante l'inserimento di direttive #pragma nel programma sorgente.

La versione 4.0 di openMP introduce nuove direttive #pragma attraverso le quali il programmatore può aiutare il compilatore a vettorizzare correttamente:

In questo esempio la direttiva dice al compilatore che il loop può essere vettorizzato e “consiglia” che i puntatori p e q vengano allineati in blocchi di 32 byte.

```
int foo (int *p, int *q) {
  int i, r = 0;
  #pragma omp simd reduction(+:r) aligned(p,q:32)
  for (i = 0; i < 1024; i++) {
    p[i] = q[i] * 2;
    r += p[i];
  }
  return r;
}
```

Esempio di vettorizzazione esplicita: Intrinsics

Le «Intrisics» sono funzioni che si mappano direttamente su una o più istruzioni assembler vettoriali (SIMD).
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#
Esempio di uso degli intrinsics, assumendo di avere un processore equipaggiato con estensione SSE4.1:

```
#include <stdio.h>
#include <smmintrin.h>
void sum(int a[4], int b[4], int c[4]) {
    __m128i ap, bp, cp;       // variabili di tipo __m128i vengono mappate su registri SSE a 128 bit
    ap = *(__m128i*)a;        // copia i 16 byte all'indirizzo a in una variabile SSE
    bp = *(__m128i*)b;        // copia i 16 byte all'indirizzo b in una variabile SSE
    cp = _mm_add_epi32(ap,bp); // calcola la somma vettoriale di ap e bp e scrive il vett. risultante in cp
    *(__m128i*)c = cp;        // copia i 16 byte della variabile SSE cp all'indirizzo b
}
int main() {
    int a[4] = { 1, 2, 3, 4 };
    int b[4] = { 10, 20, 30, 40 };
    int c[4];
    sum(a, b, c);
    printf("%d %d %d %d\n", c[0], c[1], c[2], c[3]);
    return 0;
}

$ gcc sum.c -O2 -msse4.1 -o sum
```

Programmazione parallela MIMD

I processori di un calcolatore parallelo comunicano tra loro secondo 2 schemi di comunicazione:
Shared memory: I processori comunicano accedendo a variabili condivise
Message-passing: I processori comunicano scambiandosi messaggi

Questi schemi identificano altrettanti paradigmi di programmazione parallela:

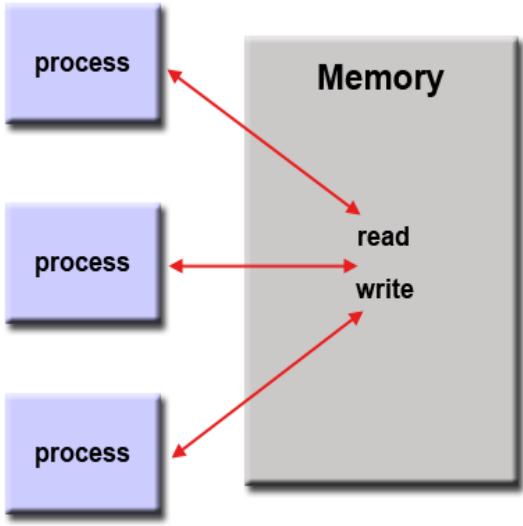
- **paradigma a memoria condivisa** o ad ambiente globale (Shared memory)
  i processi interagiscono esclusivamente operando su risorse comuni

- paradigma a memoria locale o ad ambiente locale (Message passing)
non esistono risorse comuni, i processi gestiscono solo informazioni locali e l'unica modalità di interazione è costituita dallo **scambio di messaggi (message passing)**

Memoria condivisa

Nel paradigma “shared memory” i task (processi/thread) comunicano accedendo a variabili e strutture dati condivise.
L'accesso condiviso richiede anche strumenti per la sincronizzazione delle operazioni.

PROCESSI:
SysV-IPC
- Creare sezioni di memoria condivisa (shmget() shmctl() ....)
- Sincronizzazione con semafori (semget() semctl() ...)



THREAD:
Posix thread (Pthreads)
- Comunicazione a memoria condivisa tra più thread
- Sincronizzazione con semafori (pthread_mutex_*())
- https://hpc-tutorials.llnl.gov/posix/

OpenMP:
- La sezione di codice che si intende eseguire in parallelo viene marcata attraverso una apposita direttiva che causa la creazione dei thread prima della esecuzione
- https://hpc-tutorials.llnl.gov/openmp/

Fork / Join

La fork eseguita dal task master genera dinamicamente uno o più nuovi task che eseguono un nuovo flusso di controllo parallelamente al master.

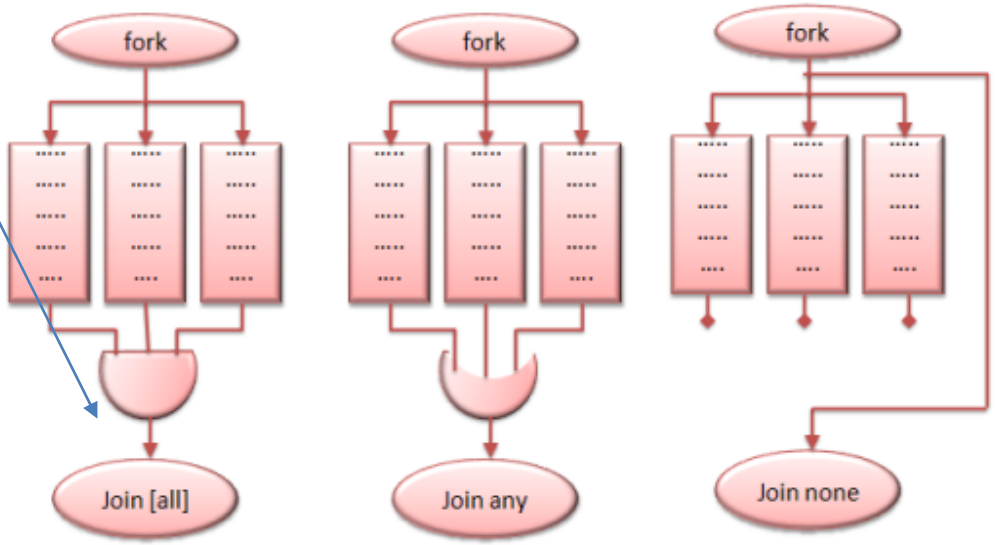
La Join viene eseguita da tutti (o parte) dei task concorrenti.

- Join All: Quando il task più lento ha raggiunto la join il master continua l'esecuzione mentre gli altri terminano.
- Join any: Il primo task che raggiunge la join sblocca il master. Gli altri terminano dopo aver raggiunto la join.
- Join none: il master thread attiva la fork e continua l'esecuzione.

Data la dinamicità dei task è un modello adatto per ambiente **multi-thread.**

OpenMP si basa sul modello **Join All**

L'utilizzo elevato di strutture fork/Join può introdurre **overhead** dovuti a start e stop dei thread.

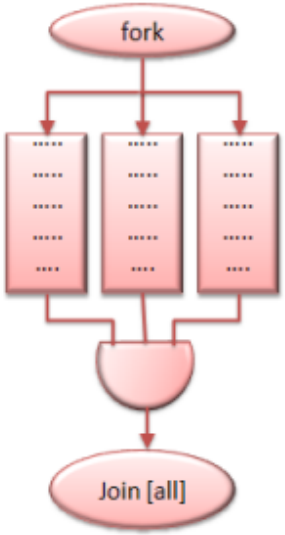


http://www.testbench.in/SV_26_FORK_JOIN.html

Esempio openMP

Una regione parallela si attiva (**fork**) in openMP con la direttiva **omp parallel**
La direttiva **omp for** distribuisce sui diversi thread le iterazioni (indipendenti) di un ciclo for.
Le iterazioni su suddivise in **chunk** di dimensione limitata (default 1) che vengono distribuiti da uno **schedule**r in modo statico (a rotazione) o dinamico.
Al termine del ciclo for si chiude anche la legione parallela (join).

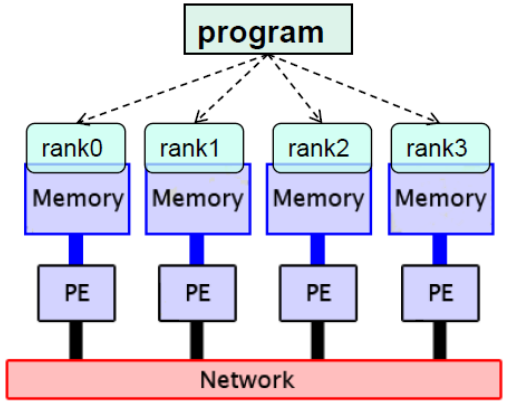
```
..
#pragma omp parallel
#pragma omp for
 for (i = 1; i <= n; i++)
    {

    }
```

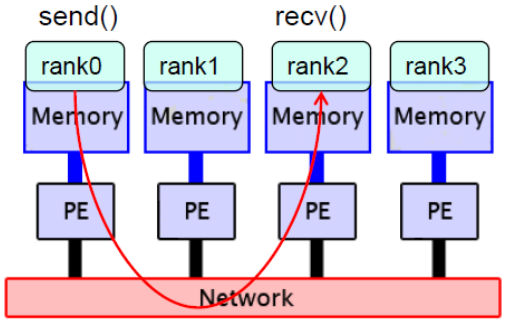


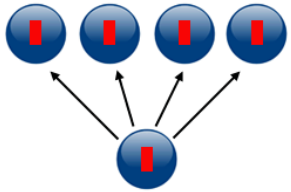
Memoria distribuita

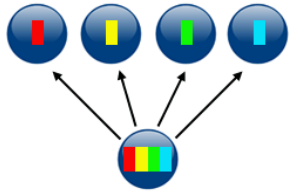
Nel paradigma message passing i processi comunicano scambiandosi messaggi
La libreria **MPI (Message Passing Interface)** https://hpc-tutorials.llnl.gov/mpi/
è uno standard “de-facto” per il message passing
Fornisce primitive per le comunicazioni **punto-punto e collettive.**

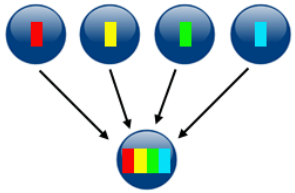
```
..
if (rank=0)
MPI_send(.., 2, ..);
if (rank=2)
MPI_receive(..,0,..);
..

..
MPI_Bcast(…, 0, …);
..
```

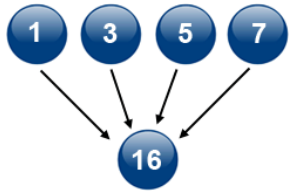





broadcast


scatter


gather



reduction

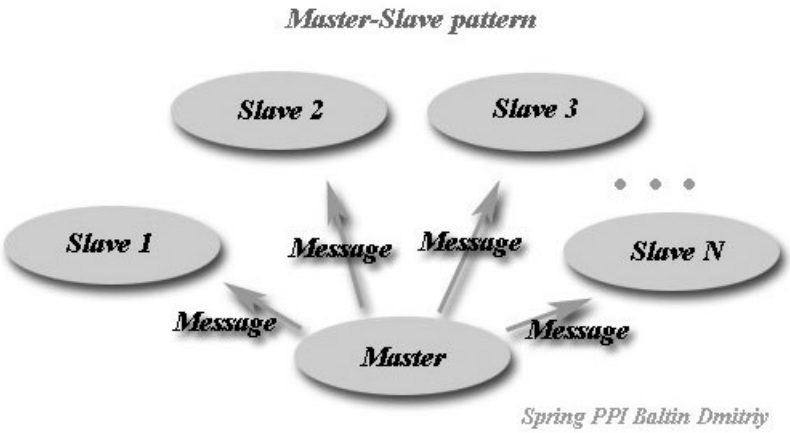
Master-Slave

Un task master controlla il lavoro svolto dagli altri task (slaves).
Utilizzo tipico: load balancing

Un programma Master Slave può essere
facilmente realizzato con il modello SPMD.

Esempio in C:

```
main (int argc, char **argv)
{
   if (process is to become a controller process)
   { Master (/* Arguments /*); }
   else
   { Slave  (/* Arguments /*); }
}
```

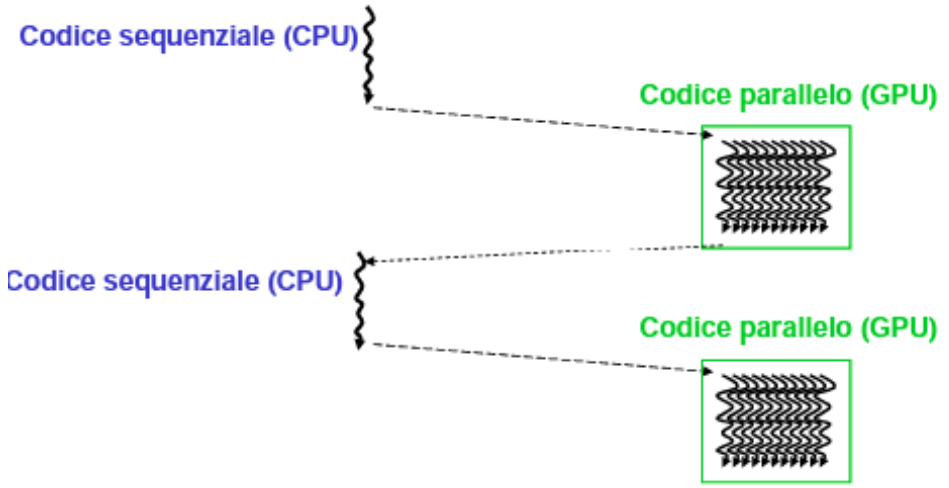


Programmazione GPU con CUDA

Il costruttore di **GPU NVIDIA** ha realizzato diversi modelli di GPU che possono essere utilizzate come acceleratore del calcolo per applicazioni «data parallel» e ha sviluppato il modello di programmazione CUDA, la libreria e il relativo compilatore nvcc

Il codice seriale o poco parallelo viene eseguito sulla CPU.
Il codice massicciamente parallelo di tipo «data parallel» viene scritto con CUDA in un «kernel» e trasferito sulla GPU assieme ai dati da elaborare.
Al termine dell'esecuzione i risultati vengono trasferiti dalla GPU alla CPU.

```
..
cudaMemcpy( .., cudaMemcpyHostToDevice);
myKernel1<<<nblocks,threadsPerBlock>>>(..);
cudaMemcpy( .., cudaMemcpyDeviceToHost);
..
..
cudaMemcpy( .., cudaMemcpyHostToDevice);
myKernel2<<<nblocks,threadsPerBlock>>>(..);
cudaMemcpy( .., cudaMemcpyDeviceToHost);
..
```



Sistemi ibridi

La programmazione delle architetture ibride avviene combinando il modello message passing (MPI) con il modello a thread (OpenMP) a cui si aggiunge la programmazione CUDA se il sistema dispone di GPU.