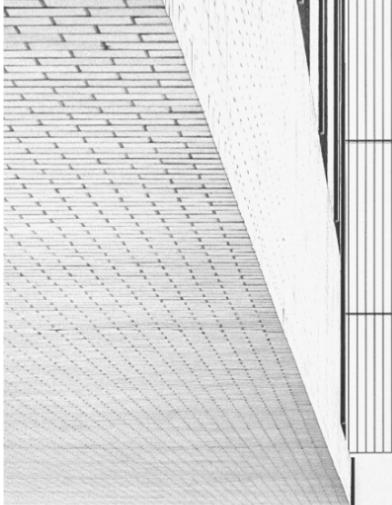
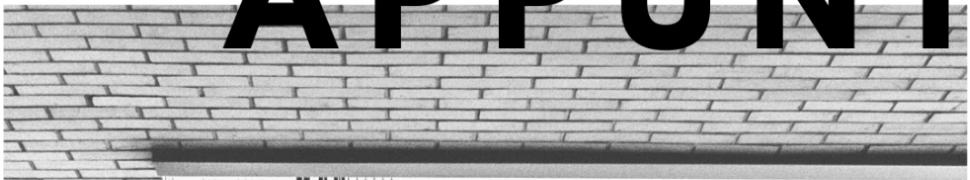
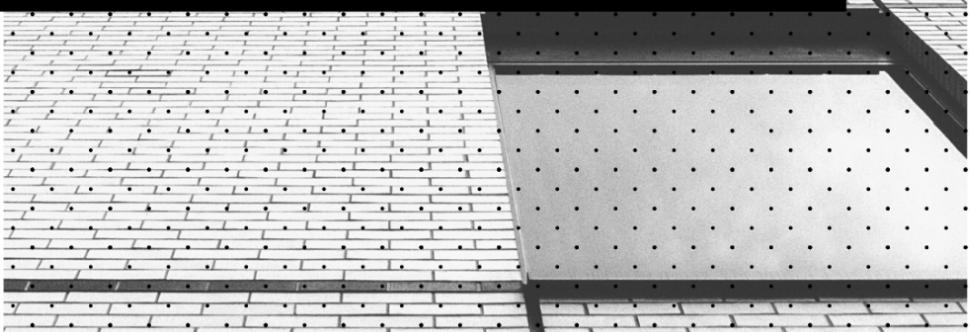


APPUNTI



PROGRAMMAZIONE PARALLELA E HPC

C.d.L. Informatica



SCRITTO DA GABRIEL BENOLLI

Università degli studi di Parma

Introduzione	5
Motivazioni dell'HPC (High Performance Computing)	5
Architetture parallele	5
Cluster UniPr	6
Programmazione a memoria condivisa	6
Programmazione a memoria distribuita	7
Programmazione GPU	8
Hardware performance	8
CPU	9
Memoria	9
RoofLine Analysis	10
Storage	11
NAS (Network Attached Storage)	11
SAN (Storage Area Network)	12
Clustered File system : GPFS	12
High Speed Networks	13
Benchmarks	13
Sistemi per il calcolo ad alte prestazioni	14
Tassonomia di FLYNN	14
SISD - Single Instruction Single Data	15
SIMD - Single Instruction Multiple Data	15
Processori Vettoriali	15
Istruzioni Vettoriali	16
MIMD - Multiple Instruction Multiple Data	16
Limiti della tassonomia di Flynn	17
Architettura della memoria	17
Sistemi a memoria condivisa	17
Sistemi a memoria distribuita	18
Sistemi Ibridi	19
Progettazione di programmi paralleli	19
Parallel computing	19
Decomposizione di dominio	20
Decomposizione funzionale	20
Comunicazione tra i Task	21
Comunicazioni collettive	21
Costo della comunicazione	22
Sincronizzazione	22
Bilanciamento del carico	23
Ottimizzazioni: Granularità	23
Scalabilità	24
Tempi di Overhead	25
Legge di Amdahl	25

Programma Parallello	25
Programmazione delle istruzioni SIMD	26
Programmazione parallela MIMD	27
SPMD - Single Program Multiple Data	28
Master-Slave	28
Fork / Join	28
Sistemi ibridi	29
Programmazione a memoria condivisa con OpenMP	29
Fork - Join Model	29
Direttive OpenMP	30
Direttiva Parallel	30
Direttiva For	31
Direttiva Sections	32
Direttiva Critical	33
Direttiva Single	33
Direttiva Barrier	34
Funzione per il calcolo dei tempi	34
MPI - Message Passing Interface	35
mpirun	36
Il modello MPMD è comunque possibile	37
Esecuzione di mpirun via SLURM	37
mpicc	37
Comunicazione Punto-Punto	39
Operazioni collettive	40
Funzioni in MPI	40
Communicators	40
MPI_Send	41
MPI_Recv	41
MPI_Sendrecv	42
Comunicazioni Collettive	42
MPI_Barrier	43
MPI_Bcast	43
MPI_Scatter	43
MPI_Gather	44
MPI_Reduction	44
MPI + OMP - Programmazione IBRIDA	44
CUDA C/C++	44
Flusso di elaborazione semplice	45
Hello World con codice device	45
Somma di interi in CUDA	46
SM e architettura GPU	50
Condivisione dati tra Threads	51

Data Race	52
Strategie di ottimizzazione delle performance	55
Accesso coalesced alla memoria globale	55
Calcolo prodotto matrici	56
IDEA NATIVE:	56
IDEA SHARED MEMORY:	57
Propagazione calore CUDA	59

Introduzione

Motivazioni dell'HPC (High Performance Computing)

Problemi di notevole impatto in diversi ambiti che possono essere risolti computazionalmente ma richiedono notevoli risorse di calcolo e storage per dare una soluzione in un tempo ragionevole.

La legge di Moore (1965) ipotizza che le prestazioni dei microprocessori raddoppiano ogni 2 anni.

I limiti della legge di Moore si sono raggiunti negli ultimi anni (attorno al 2005) a causa dei limiti fisici imposti per la riduzione delle dimensione dei transistor. La frequenza di lavoro si è stabilizzata attorno a 2-3 Ghz. La conseguenza è stata l'introduzione della tecnologia multicore all'interno dello stesso processore.

Le prestazioni complessive continuano a crescere esponenzialmente ma in modo distribuito su più core o attraverso altre tecniche hardware. Per sfruttare le prestazioni dell'hardware occorre programmare in modo parallelo le applicazioni.

È possibile parallelizzare il flusso di esecuzione di un algoritmo a diversi livelli hardware:

- all'interno di un core (vettorizzazione, pipeline, superscalare, hyperthreading),
- all'intero di un nodo (multi-core, multi-socket),
- utilizzando acceleratori (e.g. GPU),
- tra più nodi di un cluster.

Il parallelismo dell'hardware cresce ed evolve continuamente consentendo di avere nel tempo una **crescita esponenziale delle prestazioni** ma al contempo la complessità nella programmazione dei diversi livelli di parallelismo porta ad una **crescente difficoltà nello sfruttamento delle risorse disponibili**.

Architetture parallele

Nei sistemi di calcolo moderni **l'accelerazione (speed-up)** si ottiene distribuendo il carico computazionale su di una architettura hardware in grado di sfruttare le performance dei diversi livelli di parallelismo su:

- Diversi thread o processi su un singolo nodo (memoria condivisa)
- Diversi processi su più nodi interconnessi (memoria distribuita)
- Kernel di calcolo eseguiti su acceleratori (GPU)

Dal punto di vista della programmazione esistono diverse metodologie di progettazione per scomporre il carico computazionale in **Task** che verranno poi distribuiti sulle diverse unità di processamento.

La scomposizione dovrà tenere conto dell'**organizzazione dei dati**, delle **componenti funzionali** dell'algoritmo e dei **livelli di parallelismo messi a disposizione dal sistema di calcolo**.

Con il termine **SpeedUp** si intende la misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato:

$$\text{SpeedUp} = \frac{T_{\text{parallelo}}}{T_{\text{seriale}}}$$

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

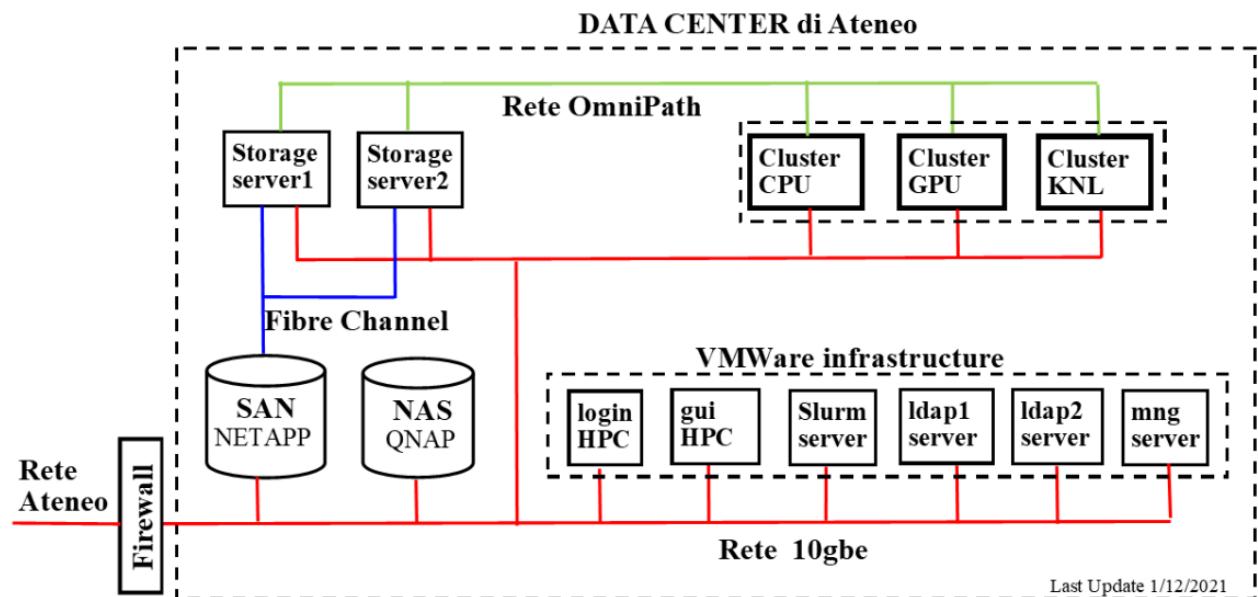
Le limitazioni della scalabilità sono dovute a **Overhead** introdotti dalla parallelizzazione, come ad esempio la comunicazione e il sincronismo tra i Task.

TDP (Thermal Design Power) si riferisce al consumo di energia sotto il massimo carico teorico. La sua unità di misura è il watt e il suo valore viene dichiarato dai produttori.

PUE (Power Usage Effectiveness) è il rapporto tra la potenza totale (PT) assorbita dal data center e quella usata dai soli apparati IT (PIT) ovvero: $\text{PUE} = \text{PT} / \text{PIT}$.

PUE rappresenta una misura di quanto efficiente sia un centro di calcolo, o data center, nell'usare l'energia elettrica che lo alimenta

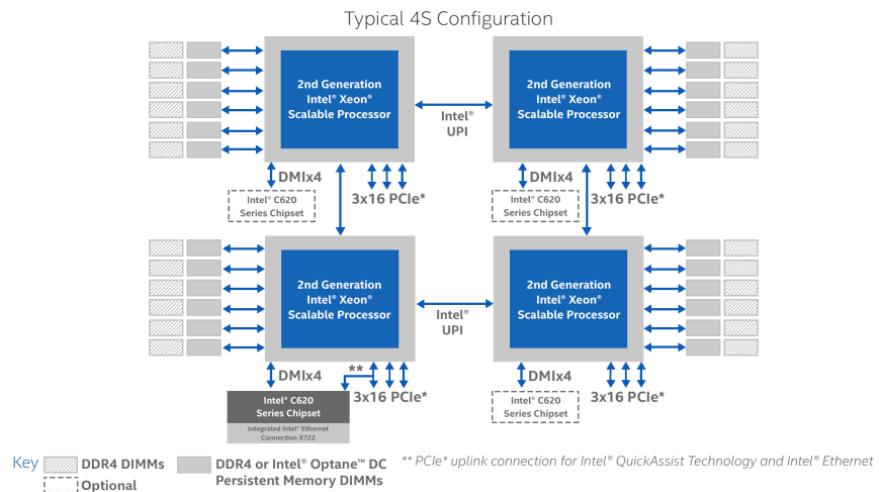
Cluster UniPr



Programmazione a memoria condivisa

Un'architettura a memoria condivisa è realizzata con sistemi **SMP** (Symmetric Multi Processor) in cui un pool di processori omogenei operano in modo indipendente ma condividono lo spazio di indirizzamento della memoria RAM.

I task del programma possono essere assegnati a thread in esecuzione su differenti unità di processamento. La comunicazione tra i task avviene mediante l'accesso sincronizzato a variabili condivise.



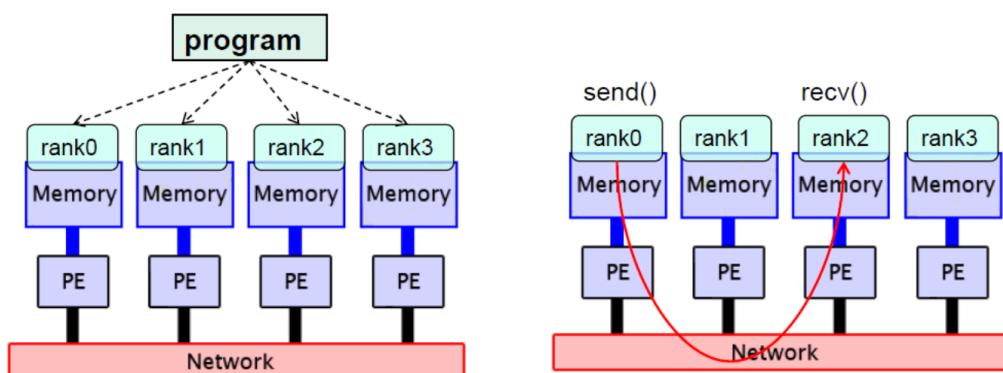
Programmazione a memoria distribuita

Un'architettura a memoria distribuita è costituita da un cluster di computer interconnessi da una rete di comunicazione.

I task (processi) sono eseguiti sulle CPU del cluster e hanno il proprio spazio di indirizzamento.

La comunicazione richiede una specifica libreria in grado di implementare diversi modelli di comunicazione come punto–punto (tra coppie di task) o globali (tra tutti i task) utilizzando percorsi fisici diversi in base alla localizzazione dei task.

MPI (Message Passing Interface) è la specifica per lo sviluppo di librerie standard di comunicazione.



Programmazione GPU

La GPU è nata come coprocessore per il rendering di immagini su un dispositivo di visualizzazione, ma vista la sua programmabilità dal 2005 si è iniziato ad utilizzarla come coprocessore della CPU. Sono quindi nate le GP-GPU (General Purpose GPU) sprovviste di uscita video.

La programmazione consiste nel costruire un kernel di calcolo che verrà tipicamente inviato assieme ai dati dall'host alla GPU, la quale elabora i dati e al termine invia i risultati all'host.

Hardware performance

La performance complessiva di un computer (singolo o cluster) è la quantità di lavoro che può essere svolto nell'unità di tempo e dipende da:

- **Hardware**

- CPU e GPU
- Memoria
- Storage
- Network

- **Software**

- Algoritmi
- Hardware exploitation = capacità di sfruttare al meglio le risorse disponibili
- Ottimizzazioni (es. compilatori)

Theoretical Peak Performance (TPP) è una stima della performance di un componente hardware (CPU, memoria, rete, storage) in base alle caratteristiche architetturali.

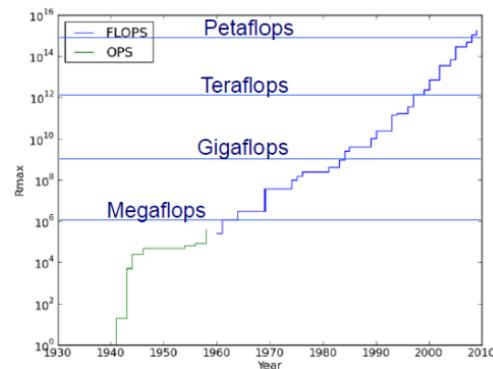
Sustained Performance (throughput): Prestazioni effettive misurate, di un componente hardware o di un sistema di calcolo tramite l'esecuzione di specifici programmi (Benchmark)

CPU

La performance di una CPU è data dalla quantità di istruzioni svolte nell'unità di tempo.
Si misurava in MIPS (milioni di istruzioni eseguire per secondo) ora è più frequente L'utilizzo di **MFLOPS** o **GFLOPS** (operazioni in virgola mobile per secondo)

Occorre specificare il tipo di dato:

- Singola Precisione (SP, 32 bit)
- Doppia Precisione (DP, 64 bit)
- Mezza Precisione (HP, 16 bit)



FLOPS = Clock x FLOPs/cycle

es.

Intel Xeon E5-6140

- Numero core: 18
- Frequ di base: 2.3 GHz (3.7 turbo mode)
- MA (Fused Multiply-Add) : 2 flops
- SIMD AVX-512: 16 flops s.p. - 8 flops d.p.
- TDP(Thermal Design Power): 140 W

Calcolo FLOPS:

$2.3 \times 2 \times 8 \text{ GFlops} = 36 \text{ Gflops (base)}$

$3.7 \times 2 \times 8 \text{ GFlops} = 59 \text{ Gflops (max)}$

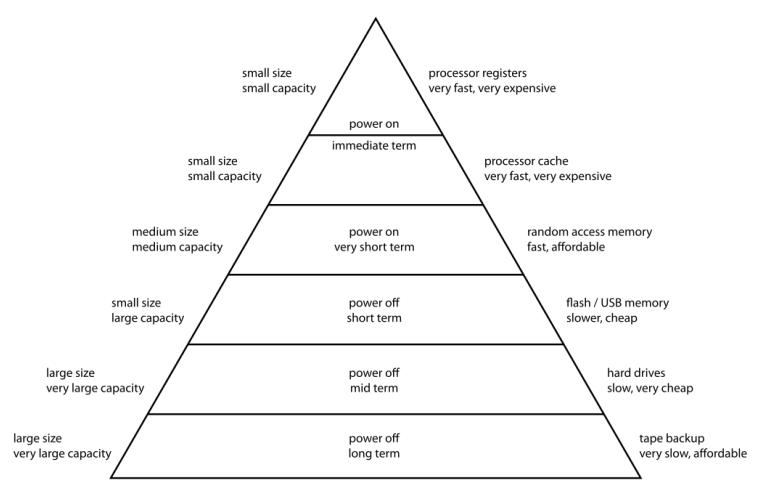
Memoria

La gerarchia della memoria determina **tempi di accesso differenti** a seconda della localizzazione del dato. La memoria può diventare un **collo di bottiglia** nelle prestazioni se non è in grado di fornire dati con il ritmo richiesto dal processore

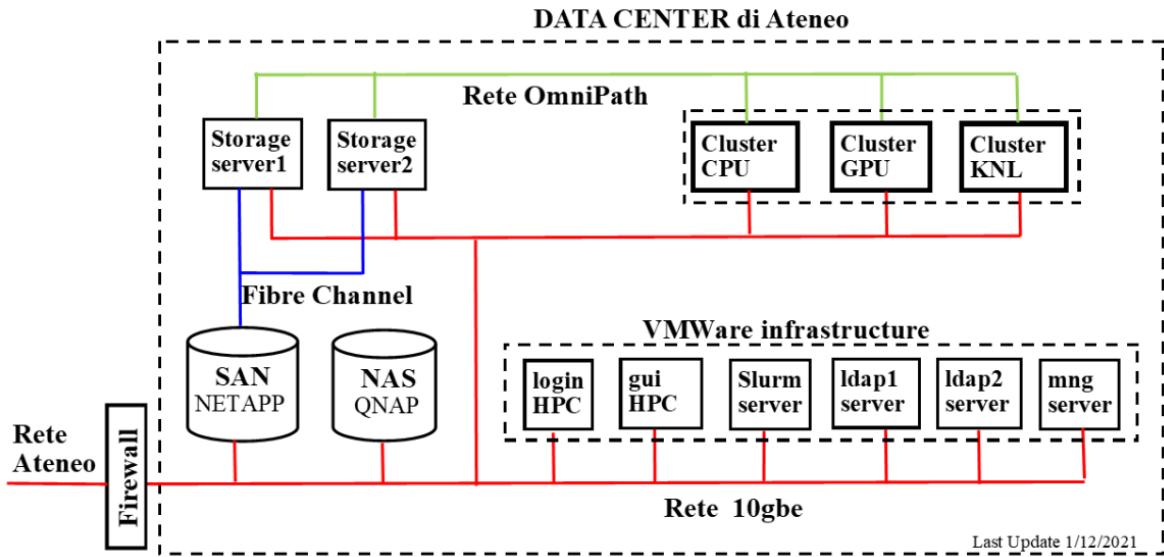
es. CPU Xeon E5-6140

- Cache L1 32+32KB (per core)
- Cache L2 1 MB (per core)
- Cache L3 24,75 MB (per socket)
- RAM DDR 384 GB (per node)

Computer Memory Hierarchy



CPU e GPU del cluster HPC.UniPr.it



Nodi CPU:

22 DualBDW:	11 Tflops
1 QuadBDW	1 TFlops
8 QuadSKL:	22 Tflops
4 KNL	6 Tflops

tot: 40 Tflops d.p.

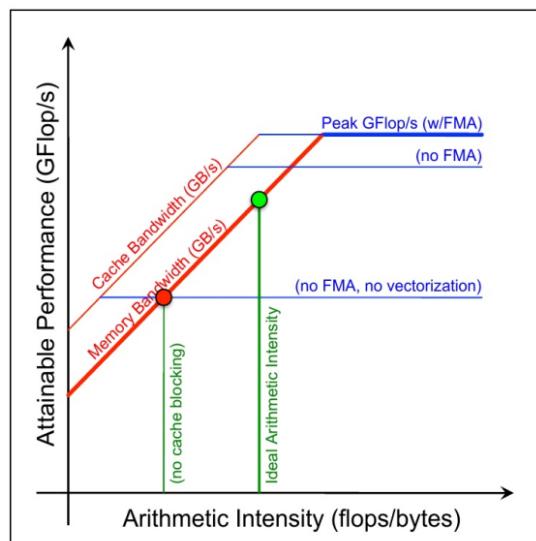
Schede GPU:

14 NVIDIA P100:	66 TFlops
2 NVIDIA V100:	14 Tflops
4 NVIDIA A100:	39 Tflops

tot: 119 Tflops d.p.

RoofLine Analysis

Il modello Roofline consente in maniera intuitiva di stimare le performance di un kernel computazionale mostrando graficamente le limitazioni inerenti CPU/GPU e memoria.



Permette di convergere aspetti riguardanti la località dei dati, la banda, e paradigmi di parallelizzazione in un'unica raffigurazione.

$$\text{Intensità Aritmetica} = \frac{\text{Total FLOPs}}{\text{Total Bytes}}$$

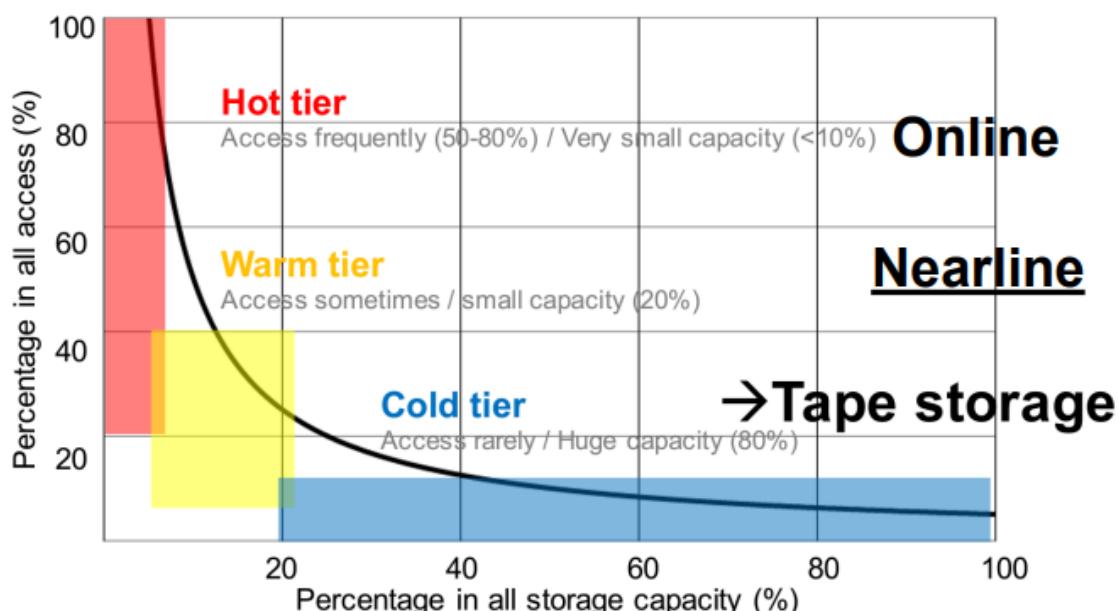
L'intensità aritmetica rappresenta il rapporto tra il lavoro e il traffico di memoria e indica il numero di operazioni effettuate per byte di traffico di memoria.

Storage

Tecnologie per i dispositivi di storage:

Tecnologie dischi esempi	Capacità TB	Prestazioni MB/s (tipico)	Costo K€
SSD Seagate (1)	7.6	600/800	2.7
HDD Seagate (1)	20	280	0.6
TAPE Cartridge HPE LTO8 (2)	30	3.6TB/hour	0.1

Tipi di storage:



L'archiviazione **online** è immediatamente disponibile per l'I/O.

Lo storage **Nearline** non è immediatamente disponibile, ma può essere reso online rapidamente senza l'intervento umano.

L'archiviazione **offline** non è immediatamente disponibile e richiede un intervento umano per diventare online.

NAS (Network Attached Storage)

Un Network Attached Storage (NAS) è un dispositivo collegato alla rete la cui funzione è quella di consentire agli utenti di accedere e condividere una memoria di massa, in pratica costituita da uno o più dischi rigidi, all'interno della propria rete o dall'esterno.

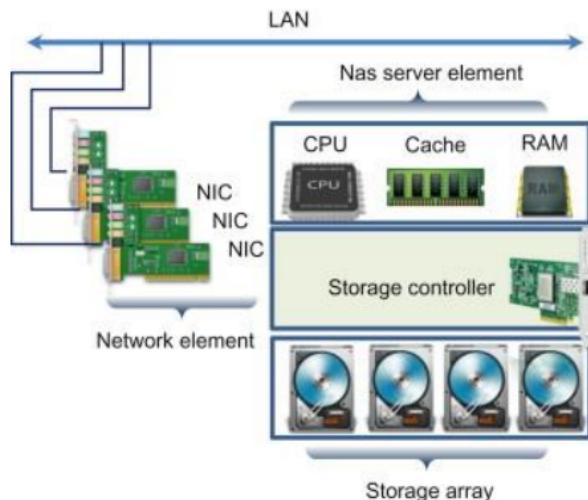
L'accesso ai file avviene tramite specifici protocolli di rete come NFS, SMB e iSCSI.

Vantaggi:

- Condivisione dati
- Gestione centralizzata
- Basso costo

Svantaggi:

- Basse prestazioni
- Risorse limitate



SAN (Storage Area Network)

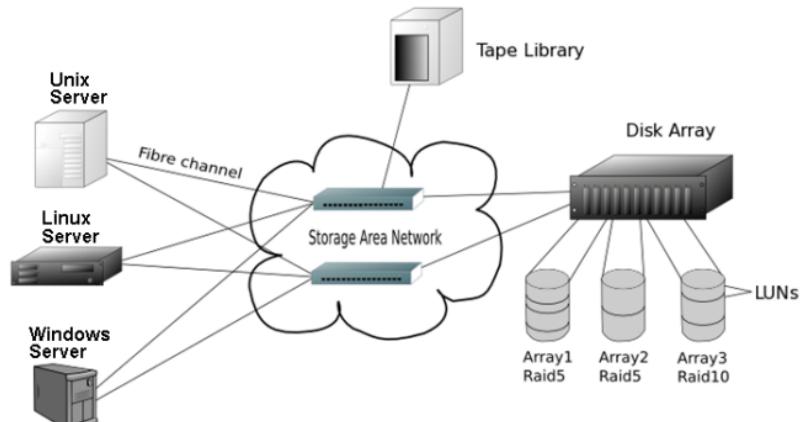
Una SAN è una rete ad alta velocità di trasmissione costituita esclusivamente da dispositivi di memorizzazione di massa, in alcuni casi anche di tipi e tecnologie differenti. Il suo scopo è quello di rendere tali risorse di immagazzinamento (storage) disponibili per qualsiasi computer connesso ad essa.

Vantaggi:

- Condivisione dati
- Gestione centralizzata
- Alte prestazioni
- Scalabilità
- Ridondanza

Svantaggi:

- Alto costo



Clustered File system : GPFS

GPFS (General Parallel File System) è un software di file system cluster ad alte prestazioni sviluppato da IBM.

È costituito da una collezione di dischi su cui GPFS memorizza dati e metadati.

Caratteristiche principali:

- **Failover:** In caso di server failure viene servito da un altro NSD server (DNS Server per cluster e SAN).
- **Shared disk:** tutti i dischi vengono utilizzati contemporaneamente da tutti i nodi
- **Byte range locking:** accesso concomitante di più utenti allo stesso file

- **Stripe**: il singolo file viene suddiviso in blocchi che vengono collocati su tutti i dischi del file system in operazioni di I/O parallele
- **Tiering**: permette di definire gerarchie di storage

Nel cluster HPC.UniPr.it lo storage è:

/hpc/home	20 TB	SAN GPFS HDD
/hpc/group	50 TB	SAN GPFS HDD
/hpc/archive	176 TB	SAN GPFS HDD nearline
/hpc/scatch	46 TB	SAN GPFS HDD + SSD

High Speed Networks

	Bitrate (Gb/s)	Bandwidth (GB/s)	Latency (microsec.)	Costo scheda (K€)
GbEthernet (tcp/ip)	1	0.1	47	0.03
10GbEthernet (tcp/ip)	10	0.9	13	0.1
Intel OmniPath	100	12	1	1
Infiniband Mellanox EDR	100	12	1	1

Benchmarks

Con il termine benchmark si intende un insieme di test software volti a fornire una misura delle prestazioni reali (sustained performance) di un computer per quanto riguarda diverse operazioni.

SPEC (Standard Performance Evaluation Corporation) è una organizzazione no-profit che produce e mantiene performance benchmark per computers.

Il Benchmark più recente per le CPU è SPEC CPU2017

I Benchmark **LINPACK** sono utilizzati per misurare le prestazioni dei computer nelle operazioni in virgola mobile. LINPACK è una libreria software sviluppata per eseguire operazioni di algebra lineare.

Nell'HPC viene utilizzato l' High Performance Linpack, una versione portabile del Benchmark LINPACK che viene utilizzato per stilare la classifica TOP500.

Comandi utili

Il comando `time` ritorna i tempi di esecuzione di un programma.

```
> time sleep 1
real 0m1.003s      # tempo reale di esecuzione (wall clock time)
user 0m0.000s      # tempo di utilizzo della CPU nello stato User
sys 0m0.003s       # tempo di utilizzo della CPU nello stato Kernel
```

`gprof` è il profiler del progetto GNU.

Un profiler consente di determinare quali parti del programma consumano più tempo.

Per utilizzarlo occorre compilare con l'opzione `-pg`.

Al momento dell'esecuzione viene generato il file `gmon.out` che potrà poi essere analizzato con il comando `gprof`.

La funzione `clock_gettime()` consente di determinare i tempi di esecuzione all'interno di un programma.

pandas è una libreria software scritta per il linguaggio di programmazione Python per la manipolazione e l'analisi dei dati.

matplotlib è una libreria per la creazione di grafici per il linguaggio di programmazione Python progettata per assomigliare a quella di MATLAB.

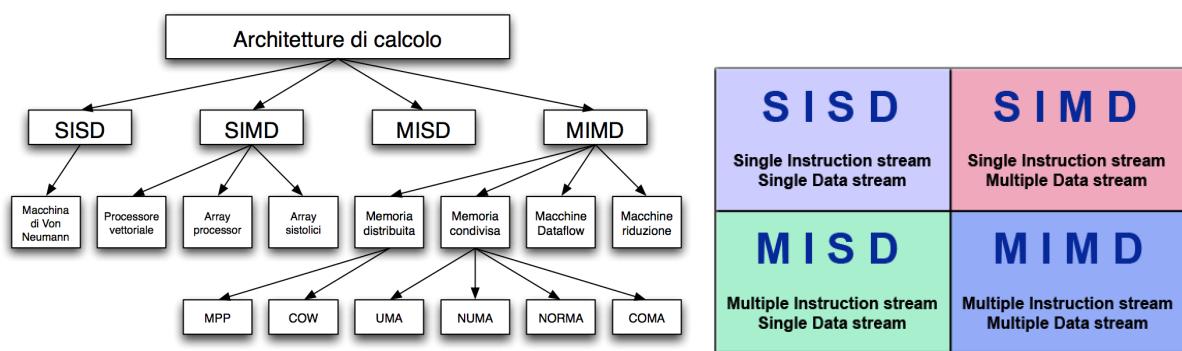
Sistemi per il calcolo ad alte prestazioni

Tassonomia di FLYNN

Ci sono differenti modi per classificare i sistemi per il calcolo parallelo.

Uno dei più usati è stato introdotto da M.J. Flynn nel 1966.

Questa tassonomia è basata su una tabella a 2 dimensioni indipendenti il flusso di **istruzioni** e il flusso di **dati**, che possono essere **single** o **multiple**.

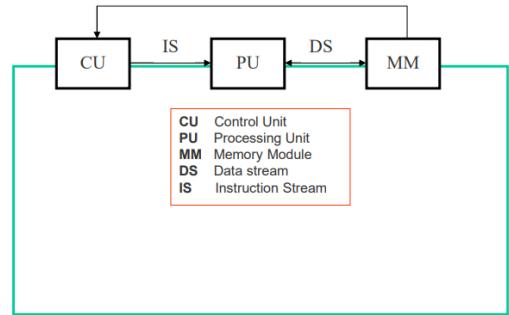


SISD - Single Instruction Single Data

Un solo flusso di istruzioni eseguito dalla CPU

Un solo flusso di dati

Sono i sistemi seriali (classica architettura di Von Neumann)



SIMD - Single Instruction Multiple Data

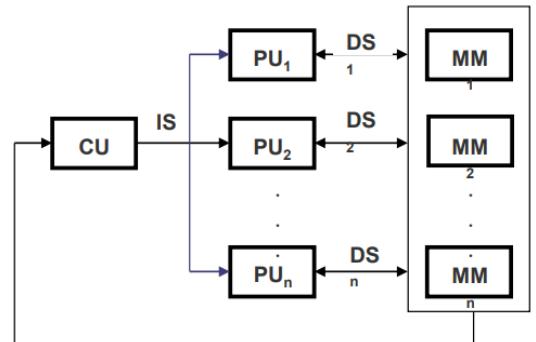
Una singola istruzione opera simultaneamente su più dati.

I sistemi SIMD hanno una sola **unità di controllo**

Principali tipologie SIMD:

- Processori Vettoriali

- Array di elementi di elaborazione che condividono l'unità di controllo
- Le istruzioni sono distribuite in parallelo a tutte le PU (Processing Unit)
- Ogni PU ha la propria memoria
- Necessaria una rete di comunicazione per i dati



- Istruzioni Vettoriali

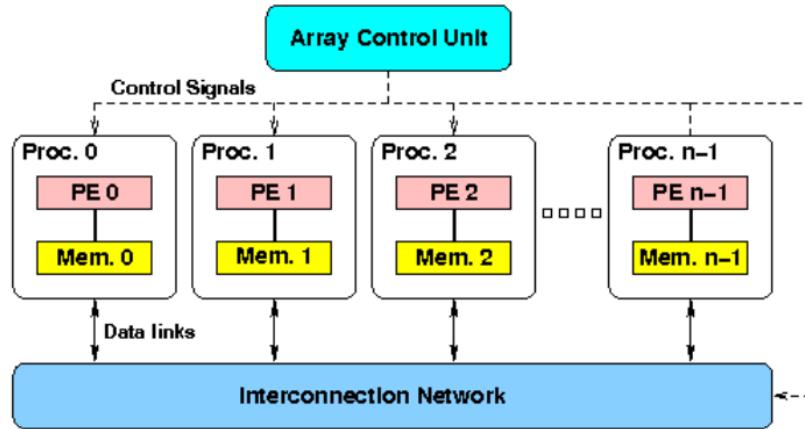
- Parallelismo realizzato all'interno del processore
- La memoria è condivisa

Processori Vettoriali

Un processore vettoriale (array processor) è costituito da più elementi di elaborazione (PU) identici e da una Unità di Controllo(CU).

L'unità di controllo preleva le istruzioni dalla memoria “programma” e distingue tra istruzioni scalari (eseguite direttamente) e istruzioni vettoriali che vengono inviate in parallelo a tutti i Processing Unit (PU)

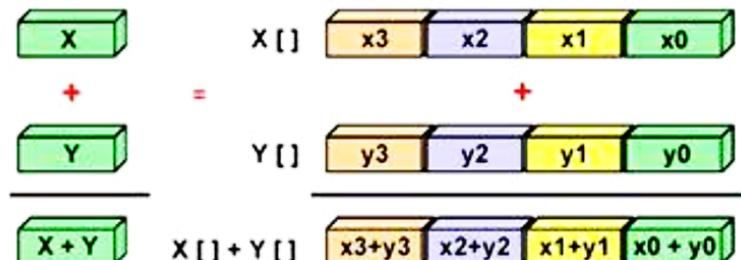
L'unità di controllo(CU) non invia una nuova istruzione finché il processore più lento non ha completato il lavoro. La rete di interconnessione consente lo scambio dei dati tra i processori



Istruzioni Vettoriali

I processori moderni supportano un set di istruzioni vettoriali (o istruzioni SIMD) che si aggiunge al set di istruzioni di istruzioni scalari. Le istruzioni vettoriali specificano una particolare operazione che deve essere eseguita su un determinato insieme di operandi detto vettore (il vettore ha dimensione 128, 256, 512 bit).

Le unità funzionali che eseguono istruzioni vettoriali sfruttano il pipelining per eseguire la stessa operazione su tutte le coppie di operandi.

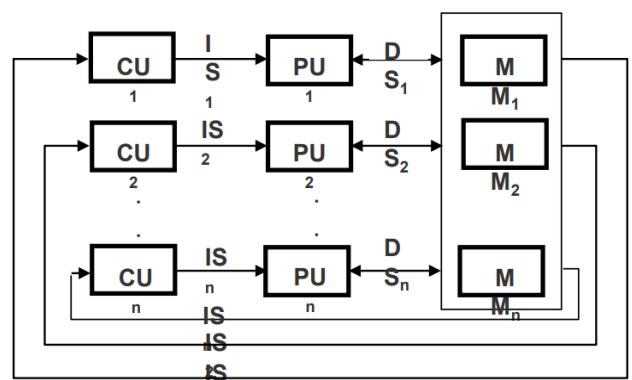


MIMD - Multiple Instruction Multiple Data

Ogni processore può eseguire un differente flusso di istruzioni.

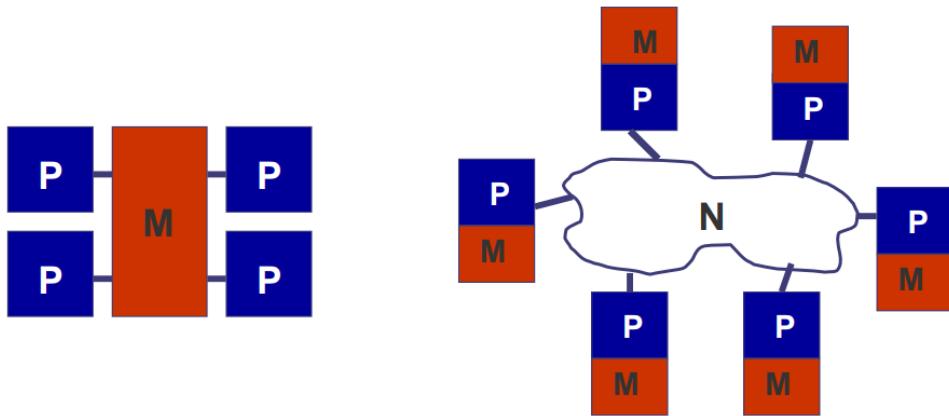
Ogni flusso di istruzioni lavora su un differente flusso di dati.

Molte architetture MIMD includono SIMD come caso particolare. I più moderni sistemi di calcolo parallelo ricadono in questa categoria.



Limiti della tassonomia di Flynn

La classificazione di Flynn non consente di esprimere caratteristiche come la distinzione tra architettura a memoria **distribuita** e architettura a memoria **condivisa**



Architettura della memoria

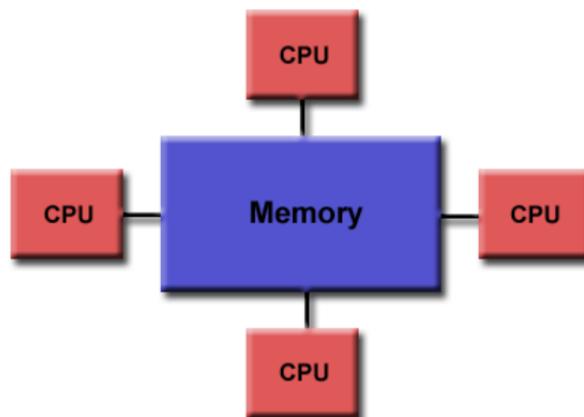
Sistemi a memoria condivisa

Tutti i processori accedono alla memoria come spazio di indirizzamento globale.
Le modifiche alla memoria da parte di una CPU sono visibili da tutti gli altri.

Esistono 2 sotto-categorie principali: **UMA** e **NUMA**

- **Uniform Memory Access (UMA)**

L'accesso alla memoria è uniforme: i processori presentano lo **stesso tempo di accesso** per tutte le parole di memoria. Ogni processore può disporre di una cache locale, le periferiche sono condivise. Negli anni i sistemi a memoria condivisa presentano un numero sempre più crescente di processori, questi multiprocessori sono chiamati **tightly coupled systems** per l'alto grado di condivisione delle risorse



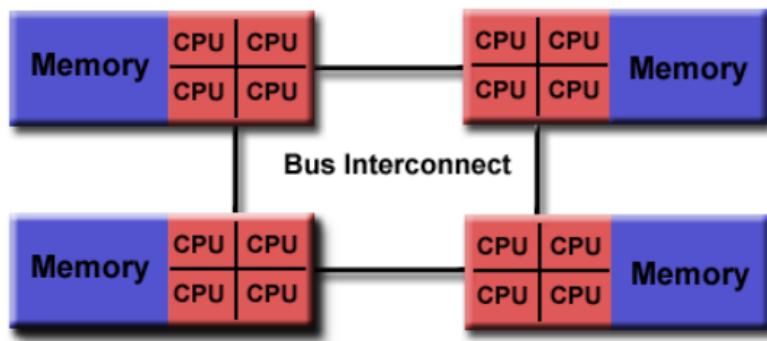
- **Non-Uniform Memory Access (NUMA)**

La memoria è **fisicamente distribuita** fra tutti i processori (ogni processore ha una propria memoria locale). L'insieme delle memorie locali forma uno spazio di indirizzi globale, accessibile da tutti i processori.

Per far sì che ogni processore possa indirizzare la memoria di tutti i processori: un processore ha accesso diretto alla memoria degli altri

Nel processore il tempo di accesso alla memoria non è uniforme:

L'accesso è più **veloce** se il processore accede alla **propria memoria locale**; quando si accede alla **memoria dei processori remoti** si ha un **delay** dovuto alla rete di interconnessione.

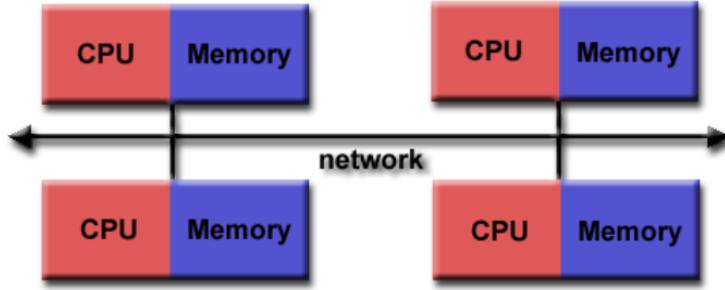


Sistemi a memoria distribuita

Ogni processore possiede una propria memoria locale, che non fa parte dello spazio di indirizzamento degli altri processori. Ogni sistema CPU/Memoria è detto **nodo** e agisce in modo indipendente.

L'infrastruttura di rete per lo scambio di messaggi può essere Ethernet, anche se viene generalmente utilizzata una tecnologia specifica a bassa latenza (e.g. Infiniband).

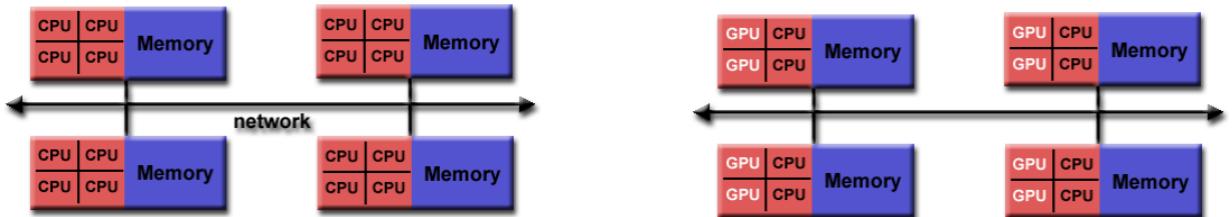
- Vantaggi
 - Il numero di processori e la memoria complessiva **scalano** con il numero di nodi.
 - Costi contenuti
- Svantaggi
 - Il programmatore deve gestire i dettagli della comunicazione tra i nodi.
 - Il tempo per l'accesso alla memoria remota dipende dal tipo di infrastruttura di rete, spesso è molto elevato per i nodi più lontani



Sistemi Ibridi

I principali sistemi paralleli oggi sono ibridi, ovvero sono composti da più nodi a memoria condivisa (UMA o NUMA) interconnessi tra loro da una rete ad alta velocità.

Nelle ultime generazioni di sistemi paralleli i nodi a memoria condivisa possono disporre di acceleratori basati su GPU. Questi acceleratori dispongono di un proprio spazio di memoria e comunicano con il nodo attraverso i bus di I/O



Progettazione di programmi paralleli

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 dominio di funzioni:

- **Domain Decomposition** (Data Parallelism)

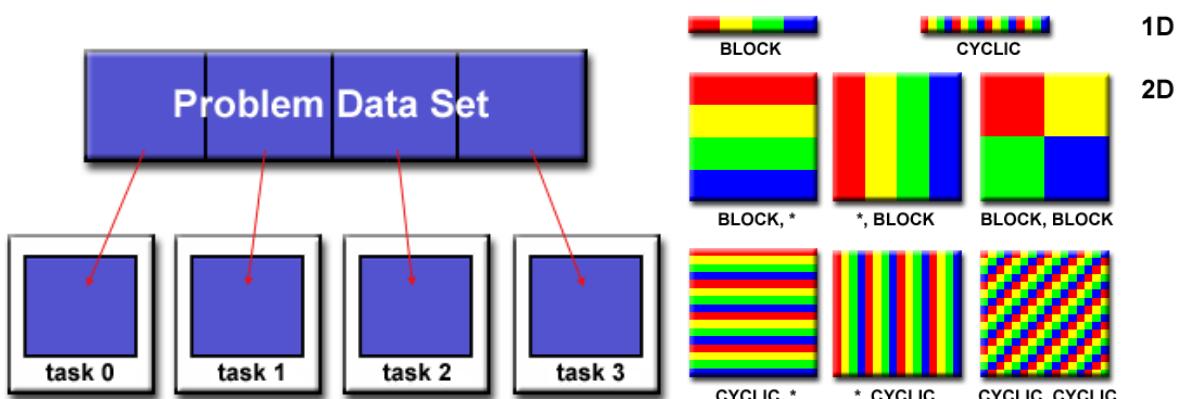
Si applica se i dati sono organizzati in strutture regolari (tipicamente array e matrici). I dati vengono suddivisi in strutture più piccole di dimensione uguale e assegnati a diverse unità computazionali. La stessa task è svolta su strutture di dati più piccole

- **Decomposizione funzionale** (Task Parallelism)

Distribuzione delle funzioni tra più soggetti: non tutti eseguono le stesse operazioni. Il problema viene scomposto in base al lavoro che deve essere svolto. Ogni attività esegue quindi una parte del lavoro complessivo. La scomposizione funzionale si presta bene a problemi che possono essere suddivisi in compiti diversi.

Decomposizione di dominio

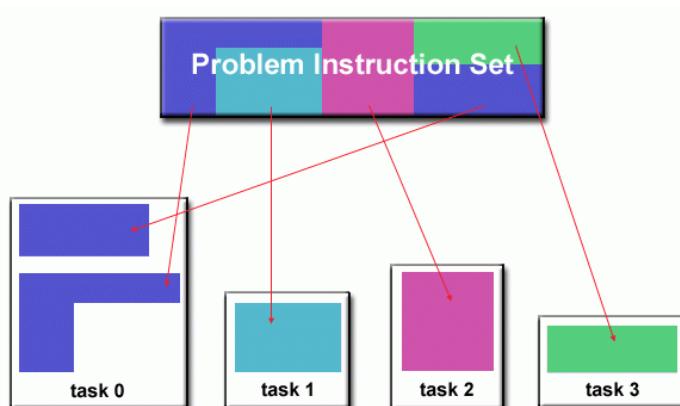
Si utilizza quando è necessario elaborare un data set di grandi dimensioni con dati strutturati che vengono suddivisi in sottodomini di dimensioni ridotte. Ogni sottodominio viene elaborato da un task specifico. Il dominio dei dati ha una propria dimensione (1D, 2D, ...) e la decomposizione può avvenire in diversi modi (vedi figura).



Decomposizione funzionale

Insieme di elaborazioni differenti ed indipendenti
Decompongo il problema in base al lavoro che deve essere svolto
Ogni processo prende in carico una particolare elaborazione

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



Comunicazione tra i Task

Le comunicazioni possono essere

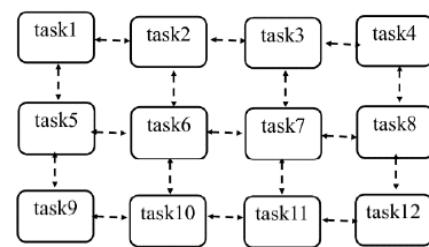
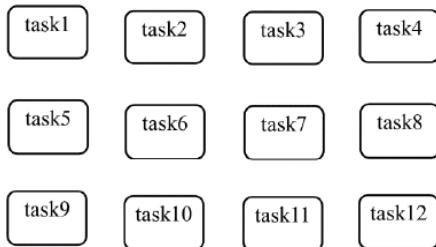
- Punto-Punto
- Collettive



La necessità di comunicazione tra i task dipende dal tipo di problema.

Esempi:

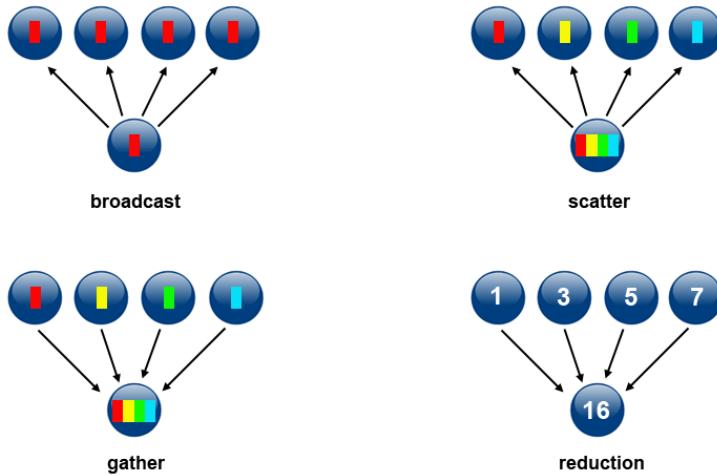
- Lo scambio del colore dei pixel di una immagine tra bianco e nero è parallelizzabile a decomposizione di dominio ma non è richiesta comunicazione tra i task
- La propagazione del calore è parallelizzabile a decomposizione di dominio ma il nuovo valore di temperatura di un sottodominio dipende dalla temperatura dei sottodomini adiacenti.



Se non abbiamo comunicazione tra i task si dice che il problema è « **embarrassingly parallel** »

Comunicazioni collettive

Le comunicazioni collettive possono avere diverse varianti:



Costo della comunicazione

Ogni messaggio inviato richiede un tempo che influisce sulle prestazioni

Per comunicazioni Punto-Punto:

$$T_{messaggio} = T_{latenza} + M_{byte}/Bandwidth$$

Per comunicazioni collettive:

$$T_{collettivo} = T_{messaggio} \times (P - 1) \quad P = \text{numero di Task}$$

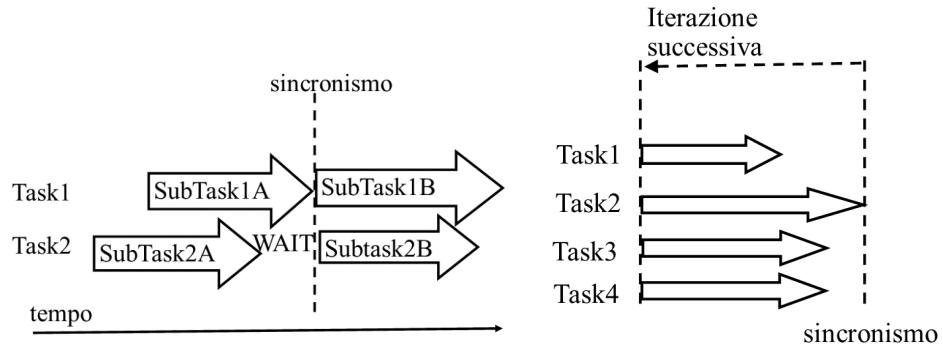
Sincronizzazione

I task possono interagire tra loro anche per dipendenze tra sezioni di codice o dati.

Esempi:

- Il Task 2 può procedere solo quando il Task 1 ha raggiunto un determinato obiettivo.
- In un programma data parallel 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.

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

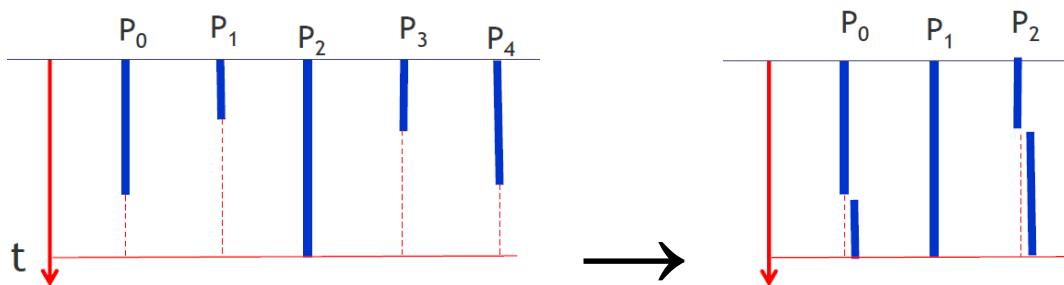


Gli strumenti per la sincronizzazione sono:

- **Le barriere:** implicano il coinvolgimento di tutti i task
- **Lock e semafori:** possono coinvolgere qualsiasi numero di task

Bilanciamento del carico

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

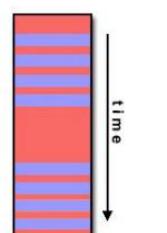


Una tecnica comune per bilanciare il carico è il modello **master-slave** (detto a volte manager worker), in cui un task master ha il compito di suddividere il lavoro in piccoli task e gestire lo scheduling dinamico dei task verso un pool di slaves

Ottimizzazioni: Granularità

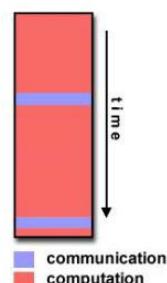
Parallelismo a grana fine (fine grain)

- Pochi calcoli tra le comunicazioni
- Può essere facile bilanciare il carico
- Overhead di comunicazioni



Parallelismo a grana grossa (coarse grain)

- Molti calcoli tra le comunicazioni
- Può essere difficile bilanciare il carico
- Probabile aumento delle performance



Con il termine **SpeedUp** si intende la misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato:

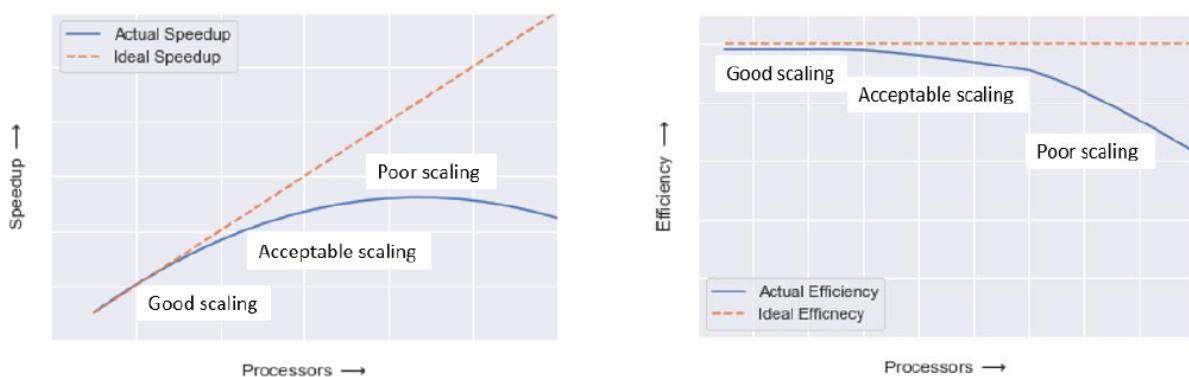
$$SpeedUp = \frac{T_{seriale}}{T_{parallelo}}$$

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

$$E = \frac{SpeedUp}{P}$$

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**. (es. mantengo le 100 iterazioni aumentando il numero dei thread)

Weak Scaling

Misura l'efficienza dell'applicazione al crescere del numero di processori mantenendo fissa la **dimensione del problema assegnata ad ogni processore**. (es. aumento il numero di thread e le 100 iterazioni diventano 100*N.thread che sto usando)

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:

$$SpeedUp = \frac{T_{seriale}}{\frac{T_{seriale}}{P} + T_{overhead}}$$

Overhead principali:

- Tempo speso per le comunicazioni
 - $T_{comm} = \sum T_{messaggio} = \sum (T_{latenza} + M_{byte}/Bandwidth)$
- Tempo di avvio e chiusura dei task paralleli
- Tempo di sincronismo

Legge di Amdahl

La legge di Amdahl distingue in un programma seriale la porzione parallelizzabile da quella non parallelizzabile $T_{Seriale} = T_{Parallelo} + T_{NonParallelo}$ e stabilisce un limite al massimo allo Speedup.

$$S_{Amdahl} = \frac{T_{seriale}}{T_{parallelo}} = \frac{T_{NonParallelo} + T_{parallelo}}{T_{NonParallelo} + \frac{T_{parallelo}}{P}}$$

Definiamo S_{real} con lo Speedup reale che tiene conto dei Amdahl e Overhead:

$$S_{real} = \frac{T_{NonParallelo} + T_{parallelo}}{T_{NonParallelo} + \frac{T_{parallelo}}{P} + T_{Overhead}}$$

Programma Parallelo

È 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**)
- Un compilatore in grado di parallelizzare automaticamente parti del programma sequenziale (**parallelismo implicito**)

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

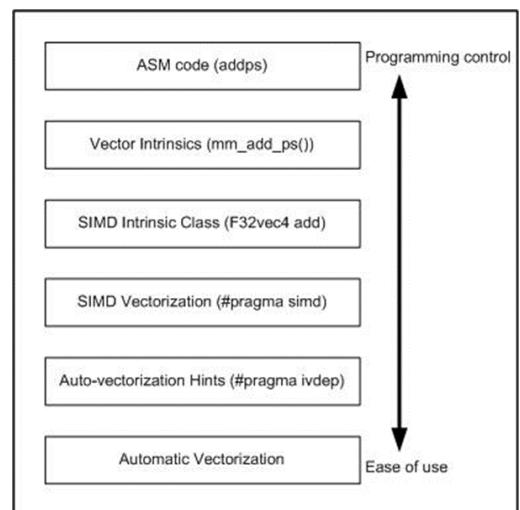
- **Automatico:**
Il compilatore analizza il sorgente ed individua possibili parallelizzazioni.
I loop (for, while,...) sono i costrutti più adatti per il parallelismo automatico
- **Direttive per il compilatore:**
Attraverso direttive il programmatore puo' dare indicazioni sulle parti di codice da parallelizzare e come farlo. Le direttive vengono ignorate da un compilatore che non riconosce le direttive
- **Esplicita:**
Occorre individuare manualmente i task e programmare esplicitamente l'interazione tra i task.

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
I compilatori possono individuare e vettorizzare parti di codice in modo autonomo
- **Guidata dal programmatore**, ad esempio tramite direttive “`#pragma`”
le quali il programmatore può aiutare il compilatore a vettorizzare correttamente
- Esplicita attraverso le “**intrinsics**”, particolari costrutti che il compilatore riconosce e mappa direttamente su codice assembly.



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)

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:

SyncV-IPC

- Creare sezioni di memoria condivisa
- Sincronizzazione con semafori

- Thread:

Posix thread (Pthreads)

- Comunicazione a memoria condivisa tra più thread
- Sincronizzazione con semafori

- 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

Nel paradigma message passing i processi comunicano scambiandosi messaggi

- Primitive message passing di base:

- **send** (parameter list)
- **receive** (parameter list)

- MPI (Message Passing Interface)

- La libreria MPI è uno standard “de-facto” per il message passing.

SPMD - Single Program Multiple Data

SPMD (Single Program Multiple Data) è un modello di programmazione in cui tutti i task eseguono la stessa copia del programma simultaneamente, elaborando dati diversi. Il flusso delle istruzioni eseguite è indipendente, quindi task diversi possono eseguire funzioni diverse dello stesso programma.

SPMD è il modello tipico di un programma a memoria condivisa openMP:
Un unico programma con diversi thread che lavorano su diversi dati.

`mpirun` è il comando MPI che mette in esecuzione N istanze dello stesso programma.

> `mpirun -np 4 a.out`

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.

es.

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

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.

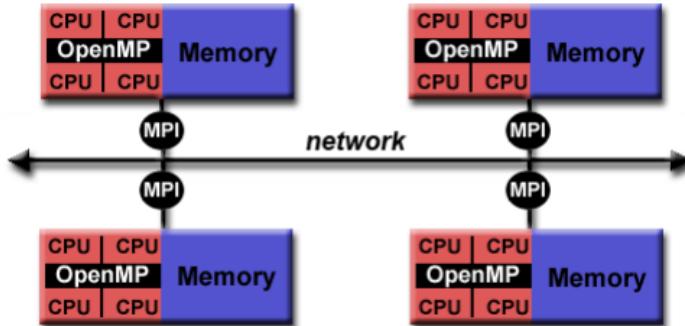
- **Join All:** Quando il task più lento ha raggiunto la join il master continua l'esecuzione mentre gli altri terminano.
- **Join any:** Il primo task che raggiunge la join sblocca il master. Gli altri terminano dopo aver raggiunto la join.
- **Join none:** il master thread attiva la fork e continua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

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

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

Sistemi ibridi

La programmazione delle architetture ibride avviene combinando il modello message passing (MPI) con il modello a thread (OpenMP).



Programmazione a memoria condivisa con OpenMP

OpenMP(**Open Multi-Processing**) è un'API (Application Program Interface) che può essere utilizzata per dirigere in modo esplicito il **parallelismo della memoria condivisa multithread**, è composto da 3 componenti principali:

- Direttive del compilatore
- Routine della libreria di routine
- Variabili d'ambiente

OpenMP è progettato per macchine a memoria condivisa multiprocessore/core. L'architettura sottostante può essere una memoria condivisa UMA o NUMA.

I programmi OpenMP realizzano il parallelismo esclusivamente attraverso l'uso di thread. Un thread di esecuzione è l'unità di elaborazione più piccola che può essere pianificata da un sistema operativo. I thread esistono all'interno delle risorse di un singolo processo. Senza il processo, cessano di esistere.

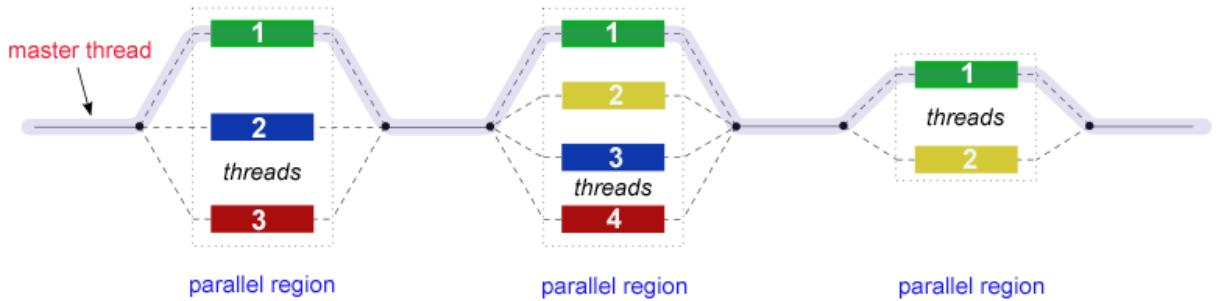
In genere, il numero di thread corrisponde al numero di processori/core della macchina. Tuttavia, l'uso effettivo dei thread dipende dall'applicazione.

OpenMP è un modello di programmazione esplicito (non automatico), che offre al programmatore il pieno controllo sulla parallelizzazione.

La parallelizzazione può essere semplice come prendere un programma seriale e inserire le direttive del compilatore o complesso come l'inserimento di subroutine per impostare più livelli di parallelismo, blocchi e persino blocchi nidificati.

Fork - Join Model

OpenMP utilizza il modello fork-join dell'esecuzione parallela:



Tutti i programmi OpenMP iniziano come un unico processo: il **thread principale**. Il thread principale viene eseguito in sequenza finché non viene rilevato il primo costrutto di **regione parallela**.

FORK : il thread principale crea quindi un insieme di *thread* paralleli .

Le istruzioni nel programma che sono racchiuse dal costrutto regione parallela vengono quindi eseguite in parallelo tra i vari thread dell'insieme.

JOIN : quando i thread del team completano le istruzioni nel costrutto della regione parallela, si sincronizzano e terminano, lasciando solo il thread principale.

Il numero di regioni parallele e i thread che le compongono sono arbitrari.

Direttive OpenMP

Formato delle direttive:

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

Direttiva Parallel

```
#pragma omp parallel default(shared) private(a,b)
```

Esempio HelloWorld

```
#include <omp.h>

main() {
    int nthreads, tid;
    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid)
```

```

{

/* Obtain and print thread id */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);

/* Only master thread does this, master thread always have id = 0 */
if (tid == 0)
{
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
}

} /* All threads join master thread and terminate */

}

```

Direttiva For

```

#include <omp.h>
#define CHUNKSIZE 100
#define N      1000

main()
{
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
{
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];

    } /* end of parallel section */
}

```

schedule: Descrive come le iterazioni del ciclo sono suddivise tra i thread del team. La pianificazione predefinita dipende dall'implementazione. Per una discussione su come un tipo di pianificazione può essere più ottimale di altri

static: Le iterazioni del ciclo sono divise in pezzi di dimensioni chunk e quindi assegnate staticamente ai thread. Se chunk non è specificato, le iterazioni sono equamente (se possibile) divise in modo contiguo tra i thread.

dynamic: Le iterazioni del ciclo sono divise in pezzi di dimensioni chunk e pianificate dinamicamente tra i thread; quando un thread termina un blocco, ne viene assegnato un altro dinamicamente. La dimensione predefinita del blocco è 1.

nowait: Se specificato, i thread non si sincronizzano alla fine del ciclo parallelo.

Direttiva Sections

La direttiva **sections** è un costrutto di condivisione del lavoro non iterativo. Specifica che le sezioni di codice indicate devono essere suddivise tra i thread del team.

Le direttive **section** indipendenti sono nidificate all'interno di una direttiva **sections**. Ogni sezione viene eseguita una volta da un thread nel team. Sezioni diverse possono essere eseguite da thread diversi. È possibile che un thread esegua più di una sezione se è abbastanza veloce e l'implementazione lo consente.

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

int main(int argc, char* argv[])
{
    int i;
    omp_set_num_threads(3);
    #pragma omp parallel private(i)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                for(i = 0; i < 5; i++)
                    printf("Th %d: %d del blocco 1\n",omp_get_thread_num(),i);
            }
            #pragma omp section
            {
                for(i = 0; i < 5; i++)
                    printf("Th %d: %d del blocco 2\n",omp_get_thread_num(),i);
            }
            #pragma omp section
            {
                for(i = 0; i < 5; i++)
                    printf("Th %d: %d del blocco 3\n",omp_get_thread_num(),i);
                sleep(4);
            }
        } // end sections
```

```

//#pragma omp barrier
    printf("%d ha finito\n", omp_get_thread_num());

} // end parallel
}

```

Direttiva Critical

Sezione critica, un solo thread alla volta esegue quella determinata istruzione

```

#include <omp.h>
#include <stdio.h>

int main() {
int t, i, j=0;
#pragma omp parallel private(t, i) shared(j)
{
    t = omp_get_thread_num();
    printf("running %d\n", t);
    for (i = 0; i < 1000000; i++)
    {
#pragma omp critical
        j++; /* race! */
    }
    printf("ran %d\n", t);
}
printf("%d\n", j);
}

```

Direttiva Single

La direttiva single permette di far eseguire il codice ad un solo thread, il primo che arriva. Possiamo anche specificare che sia il thread master ad eseguire quel determinato codice, possiamo specificarlo usando la direttiva #pragma omp master

```

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define SIZE 1000000
#define NUMTHR 16

double array[NUMTHR][SIZE];

int main(int argc, char *argv[])
{

```

```

int i=0,j=0;

#pragma omp parallel shared(array) private(j,i) num_threads(NUMTHR)
{

    j=omp_get_thread_num();
    srand48((unsigned)time(NULL)); // inizializzazione del seme per drand48()

    printf("Esecuzione del thread %d: inizio calcolo \n", j);
    for (i=1; i<SIZE; i++) array[j][i] = drand48() * drand48() * drand48();
    printf("Esecuzione del thread %d: fine calcolo \n", j);

    #pragma omp single
    { printf("THR SINGLE: entra sleep 4 \n"); sleep(4); printf("THR SINGLE
      esce sleep 4 \n"); }

    #pragma omp barrier

    printf("%d ha finito\n", j);

}
}

```

Direttiva Barrier

La direttiva **barrier** sincronizza tutti i thread nel team.

Quando viene raggiunta una direttiva BARRIER, un thread attende a quel punto finché tutti gli altri thread non hanno raggiunto quella barriera. Tutti i thread riprendono quindi l'esecuzione in parallelo del codice che segue la barriera.

```
#pragma omp barrier
```

Funzione per il calcolo dei tempi

```

#include <omp.h>
#include <iostream>
#include <unistd.h>

using namespace std;

int main() {

double t1,t2;

```

```

cout << "Start timer" << endl;
t1=omp_get_wtime();

// Do something long
sleep(2);

cout << "Stop timer" << endl;
t2=omp_get_wtime();

cout << "time: " << t2-t1 << endl;
}

```

`omp_set_num_threads(NT)` → specificare di creare NT thread per parallelizzare

`omp_get_thread_num()` → indica quale thread sta eseguendo

`omp_get_num_threads()` → indica quanti thread ci sono attualmente

es. `printf ("tid:%d/%d \n",omp_get_thread_num(),
omp_get_num_threads());`

`omp_get_wtime();` → ritorna il tempo di orologio corrente

`omp_get_max_threads` → numero massimo di thread in una regione parallela

`omp_get_num_procs` → ritorna un intero con il numero di processori sul nodo

MPI - Message Passing Interface

Il Message Passing Interface (MPI) è un protocollo di comunicazione per computer. È di fatto lo standard per la comunicazione tra nodi appartenenti a un cluster di computer che eseguono un programma parallelo multi-nodo. MPI rispetto alle precedenti librerie utilizzate per il passaggio di parametri tra nodi, ha il vantaggio di essere molto portabile (MPI è stata implementata per moltissime architetture parallele) e veloce (MPI viene ottimizzato per ogni architettura).

Lo standard MPI definisce la sintassi e la semantica delle routine di libreria utili a un'ampia gamma di utenti che scrivono programmi di passaggio messaggi

L'interfaccia di passaggio messaggi (MPI) è un mezzo standardizzato per lo scambio di messaggi tra più computer che eseguono un programma parallelo nella memoria distribuita.

mpirun e **mpicc** sono i comandi principali per l'utilizzo del modello a memoria distribuita con MPI nei sistemi HPC.

Sul cluster HPC sono installate le seguenti implementazioni di MPI-1 MPI-3 MPI-4 in openMPI e Intel-MPI

MPI	Compiler	module load
openmpi (1.10.7)	gcc5 (5.4.0)	gnu openmpi
openmpi3 (3.1.4)	gcc7 (7.3.0)	gnu7 openmpi3
openmpi4 (4.0.2)	gcc7 (7.3.0)	gnu7 openmpi4
openmpi3 (3.1.4)	gcc8 (8.3.0)	gnu8 openmpi3
intelmpi3.1 (2019.5.281)	intel-compiler (2019.5)	intel impi
openmpi3 (3.1.4)	intel-compiler (2019.5)	intel openmpi3

mpirun

mpirun è il comando che si occupa della creazione e dell'esecuzione dei processi. La gestione dei processi sui diversi nodi avviene tramite demoni (Orted in openMPI) che comunicano tra loro via ssh

mpirun realizza il modello **SPMD (Single Program Multiple Data)**: lo stesso programma viene eseguito su istanze multiple (task).

Ogni task ha la propria memoria e i propri dati. Ogni task è identificato da un numero intero denominato **rank** nel range [0, N-1].

Il numero di task e la loro distribuzione sui nodi è definita attraverso le opzioni di mpirun. Le opzioni non sono standardizzate e possono variare tra le diverse implementazioni e versioni. mpirun può essere integrato in un eventuale Job Manager, come Slurm, che può prendere il controllo dell'esecuzione.

Opzioni principali di mpirun:

- -np -n (openmpi e intel) : numero di processi da attivare
- -ppn (intel) -npernode (openmpi) : processi per nodo
- -host (intel e openmpi) : lista dei nodi su cui attivare i processi
- -hostfile (openmpi) -machine (intel) : file con l'elenco degli host e il numero di slot per host

Esempio comandi:

```
> mpirun -np 6 hostname      # esegue 6 istanze di hostname all'interno di 6
                                processi eseguiti sullo stesso host da cui e' stato
                                eseguito il comando

> mpirun --host wn53,wn54 hostname      # descrive i nodi su cui attivare i processi

> mpirun --host wn53,wn54 -np 2 hostname    # esegue 2 istanze di hostname
```

```
> mpirun --host wn53,wn54 -npernode 1 -np 2 hostname # esegue due istanze  
di  
                                              hostname, ma ne viene  
                                              eseguita uno per nodo
```



```
> mpirun --host wn53,wn54 -ppn 2 hostname # esegue 2 istanze per nodo di  
                                              hostname, due sul nodo wn53 e due  
                                              sul nodo wn54, 4 istanze in totale
```

Il modello MPMD è comunque possibile

Esempio:

```
> mpirun -np 4 hostname : -np 2 date
```

Esegue 4 istanze di hostname e 2 istanze di date. quindi è possibile avere più programmi in esecuzione su diversi nodi.

Esecuzione di mpirun via SLURM

Grazie all'integrazione tra slurm e mpirun, le risorse selezionate da Slurm vengono automaticamente comunicate a mpirun tramite dei wrapper che adattano la sintassi di Slurm alle diverse implementazioni MPI.

Possiamo specificare il numero dei nodi, i processi per cpu ad esempio, direttamente tramite le variabili d'ambiente o le direttive ad inizio file di slurm. Il comando mpirun viene lanciato senza alcuna specifica mentre dovrà specificare ad inizio file slurm con le variabili d'ambiente, mpirun controllerà automaticamente il contesto hardware su cui lavorare.

L'elenco delle risorse assegnate può essere visualizzato consultando le variabili di ambiente.
Le variabili utili sono:

\$SLURM_JOB_NODELIST (elenco dei nodi assegnati da slurm)
\$SLURM_JOB_CPUS_PER_NODE (numero di cpu assegnate per nodo)

[QUI](#) è possibile consultare l'elenco delle opzioni SLURM

mpicc

MPI fornisce una libreria per lo scambio di messaggi tra i processi.

Per utilizzare le primitive della libreria si utilizza un compilatore specifico che è un wrapper del compilatore di base.

Linguaggio	Default Compiler	MPI wrapper
Gnu C	gcc	mpicc
Gnu C++	g++	mpicxx o mpic++
fortran	gfortran	mpifc
Intel C	icc	mpiicc
Intel C++	icpc	mpiicpc

La comunicazione a memoria distribuita necessita di una rete di comunicazione per lo scambio di informazioni. Ogni processore possiede una propria memoria locale, ogni memoria è separata e possiede uno spazio degli indirizzi indipendente.

Le operazioni di lettura e scrittura sono locali per questo non ci sono problemi di coerenza nelle cache. Per garantire l'accesso ai dati remoti ad un task il programmatore deve gestire esplicitamente la comunicazione tra task

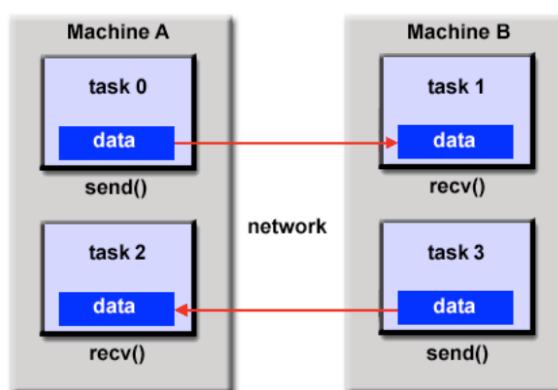
Il modello di programmazione corrispondente è chiamato **message passing** e specifica che ogni task può accedere direttamente solo alla propria memoria locale e deve comunicare con gli altri task per poter accedere alla loro memoria locale.

MPI è una specifica per un'interfaccia di libreria, non un'implementazione, ci sono multiple implementazioni di MPI.

MPI non è un linguaggio e tutte le operazioni MPI sono espresse come funzioni, routine, o metodi per C, C++, Fortran

Message passing with MPI

La cooperazione tra processi è basata su comunicazioni esplicite un processo sander e un processo receiver si scambiano messaggi.



Ogni processo è un'istanza del sottoprogramma in esecuzione, di solito, lo stesso sottoprogramma viene eseguito su set di dati diversi

SPMD (single program, multiple data)

Ogni processo è identificato da un numero intero, chiamato **rank**, che va da 0 a n-1, dove n (dimensione) è il numero totale di processi.

I messaggi sono composti in due parti:

- Intestazione:
 - source: rank of the sender
 - destination: rank of the receiver
 - tag: ID of the message (from 0 to MPI_TAG_UB)
 - communicator: context of the communication
- Body
 - type: MPI_datatype
 - length: number of elements
 - buffer: array of elements

I tipi di dato sono i seguenti:

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Comunicazione Punto-Punto

È il tipo di Comunicazione che coinvolge solo due processi un sender e un receiver. La comunicazione può essere sincrona o asincrona.

I **buffer** sono implementati in modo differente da ogni libreria MPI

Gli spazi di indirizzamento gestiti dai programmati, per l'allocazione delle variabili, sono chiamati buffer dell'applicazione. I buffer possono essere presenti in entrambi i lati sia sander sia receiver, sono di dimensione limitata e spesso non possono essere controllati dal programmatore.

Alcune operazioni possono causare il blocco del chiamante, le operazioni di non bloccanti consentono al processo di continuare, subito dopo la chiamata. Il processo può fare **test** o **wait** del processo remoto subito dopo la chiamata non bloccante.

Operazioni collettive

- Barrier (for synchronization)
- Data Movement (collective communications)
 - Broadcast
 - Scatter
 - Gather
- Reduction (collective computations)
 - Minimum, Maximum
 - Sum
 - Logical OR, AND, etc.
 - User-defined

Funzioni in MPI

MPI_ è uno spazio dei nomi riservato per funzioni e le routine MPI.

Dopo il prefisso, solo la prima lettera è in maiuscolo (es MPI_Send())

Tutte le funzioni MPI restituiscono un intero.

Le costanti sono vengono scritte in tutte in maiuscolo.

Nella header va incluso: #include <mpi.h>

Communicators

Un comunicatore è un insieme di processi che possono comunicare tra loro.

- ha un nome
- ha una dimensione (numero di processi)
- ogni processo può essere univocamente identificato(rank)
- i processi sono uguali

Due processi possono comunicare se appartengono allo stesso comunicatore

Il comunicatore predefinito è MPI_COMM_WORLD, che include n processi (di rank= 0,..,n-1).

`MPI_Comm_rank(MPI_Comm comm, int *rank)` → restituisce il rank del processo

`MPI_Comm_size(MPI_Comm comm, int *size)` → restituisce la dimensione del comunicatore

`int MPI_Init(int *argc, char ***argv)` → la prima routine che deve essere chiamata,

apre la sezione MPI e crea il comunicatore di default MPI_COMM_WORLD

```
int MPI_Finalize() → indica la fine della sezione MPI
```

MPI_Send

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

- **buf** → puntatore al messaggio da inviare (buffer dell'applicazione)
- **count** → numero di elementi nel messaggio
- **datatype** → tipo degli elementi nel messaggio
- **dest** → rank del destinatario
- **tag** → numero intero non negativo il cui scopo è lasciato all'utente
- **comm** → comunicatore

MPI_Recv

```
int MPI_Recv(void *buf,  
            int count,  
            MPI_Datatype datatype,  
            int source,  
            int tag,  
            MPI_Comm comm,  
            MPI_Status *status)
```

- **buf** → puntatore all'array in cui deve essere archiviato il messaggio ricevuto
- **count** → numero di elementi nel messaggio
- **datatype** → tipo degli elementi nel messaggio
- **source** → è il rank del mittente
- **tag** → solo i messaggi con il tag specificato vengono considerati per la ricezione
- **comm** → comunicatore
- **status** → contiene informazioni sull'intestazione del messaggio da ricevere

Una comunicazione è **locally completed** su un processo se quest'ultimo ha completato la sua parte di operazioni relative alla comunicazione (l'ultima è lo svuotamento del buffer di uscita). Per quanto riguarda l'esecuzione, il completamento locale significa che il processo può eseguire le istruzioni che seguono SEND o RECV.

Una comunicazione è **globally completed** se tutti i processi coinvolti hanno completato la loro operazioni relative alla comunicazione.

Una comunicazione è **globally completed** se e solo se è **locally completed** su tutti i processi coinvolti.

Quando le operazioni sono completate?

Synchronous send → completato quando il buffer dell'applicazione può essere riutilizzato e la ricezione del messaggio è stata completata

Buffered send → completato quando il messaggio è stato completamente copiato nel buffer di trasferimento

Standard send → completato quando il buffer dell'applicazione può essere riutilizzato

Receive → completato quando il messaggio è arrivato

MPI_Sendrecv

```
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype s_dtype, int dest,
int stag, void *dbuf, int dcount, MPI_Datatype d_type, int src, int dtag,
MPI_Comm comm, MPI_Status *status)
```

Esegue un'operazione di invio e ricezione bloccante, invia e riceve utilizzano lo stesso comunicatore, ma è possibile usare tag diversi. Il buffer di invio e il buffer di ricezione devono essere disgiunti e possono avere lunghezze e tipi di dati diversi.

Possiamo lasciare non specificato il sender nella funzione MPI_Recv sostituendolo con MPI_ANY_SOURCE.

Possiamo usare una wild card anche per il campo tag utilizzando MPI_ANY_TAG

Elenco di primitive bloccanti e non bloccanti:

Blocking:

- Synchronous send → MPI_Ssend
- Buffered send → MPI_Bsend
- Standard send → MPI_Send
- Ready send → MPI_Rsend
- Receive → MPI_Recv

Nonblocking:

- Synchronous non blocking send → MPI_Issend
- Buffered non blocking send → MPI_Ibsend
- Standard non blocking send → MPI_Isend
- Ready non blocking send → MPI_Irsend
- Non blocking receive → MPI_Irecv

Le primitive bloccanti terminano quando il messaggio è stato inviato e il buffer può essere riutilizzato (MPI_Send) e il receiver ha iniziato a ricevere il messaggio(MPI_Ssend)

Comunicazioni Collettive

Sono comunicazioni che coinvolgono più processi, evitano che il programmatore debba implementare la funzione usando numerose comunicazioni punto-punto.

Ci sono 3 classi: all-to-one, one-to-all and all-to-all

MPI_BARRIER

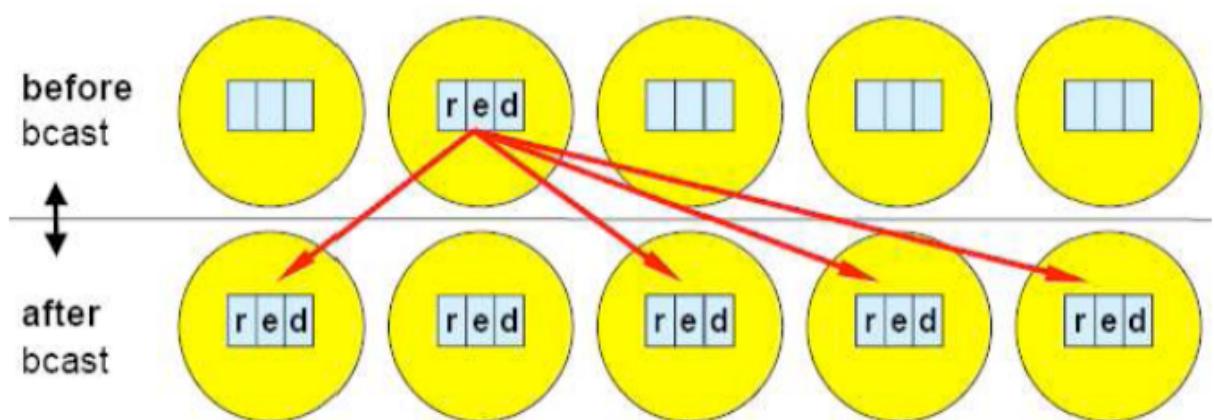
int MPI_BARRIER(MPI_Comm comm)

I task devono fermarsi in attesa che tutti i task abbiano raggiunto il punto in cui è stata inserita un MPI_BARRIER.

MPI_BCAST

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

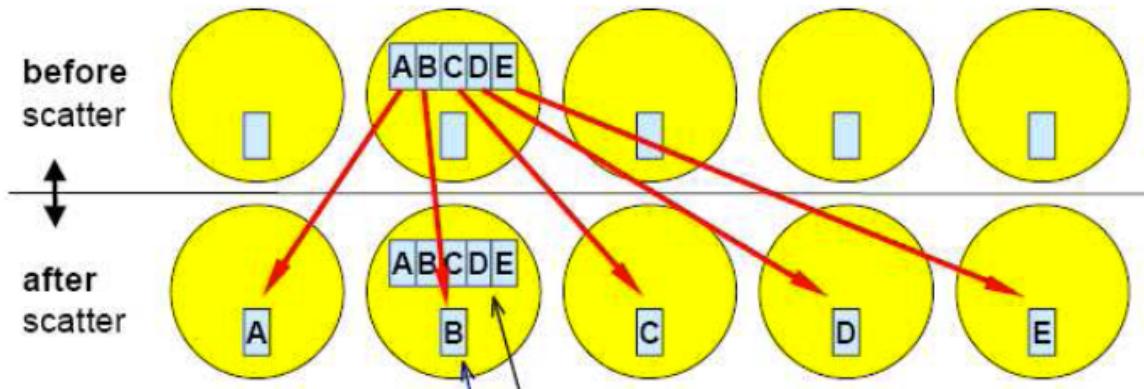
Replica il contenuto del buffer di root su tutti gli altri processi all'interno del comunicatore



MPI_SCATTER

int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

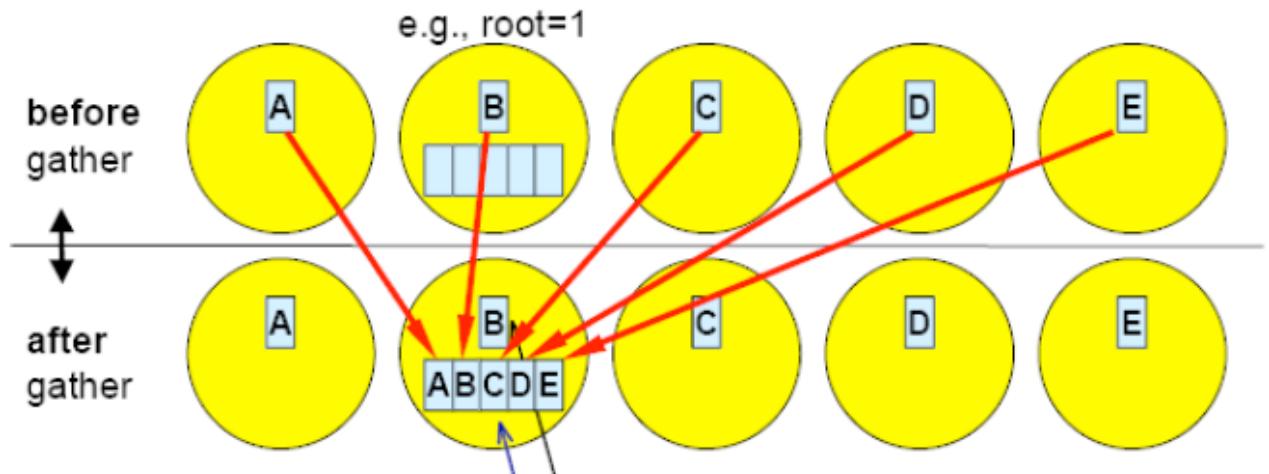
Suddivide il dato da inviare in n parti (con n = numero task) e invia la iesima parte ad ogni task



MPI_Gather

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Operazione opposta a MPI_Scatter, ogni processo, invia il contenuto del send buffer al processo root, il processo root riceve i dati e li ordina in base al rank del sender.



MPI_Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Esegue una computazione sui dati distribuiti tra i processi all'interno di un gruppo di processi, le operazioni possono essere: +, *, min, max, logic operation(and, or, xor).

Tutte le primitive collettive possono essere non bloccanti, esiste la variante della primitiva con la I davanti così da separare l'esecuzione dal test e sincronizzazione.

MPI + OMP - Programmazione IBRIDA

Nella programmazione ibrida (MPI + openMP) viene generalmente attivato un solo task MPI per nodo (o per socket) il quale si occupa della comunicazione con gli altri task, mentre il calcolo all'interno del nodo (o del socket) viene parallelizzato con openMP.

CUDA C/C++

L'architettura della GPU ci consente di eseguire la stessa operazione su diversi dati (SIMD), con un forte livello di parallelismo. Possiamo fare molte cose ma non MIMD.

CUDA è un framework di Nvidia per la programmazione GPU.

Ci fornisce un insieme di estensioni per gestire la programmazione eterogenea e delle API semplici per gestire i device e la memoria.

Terminologia fondamentale:

HOST → La CPU e la sua memoria

DEVICE → La GPU e la sua memoria

L'architettura di solito è di tipo Master-Slave dove il device dipende dall'host.

Flusso di elaborazione semplice

Un normale flusso di elaborazione su GPU è composto da 3 passaggi fondamentali:

1. Copiare i dati di input dalla memoria della CPU alla memoria della GPU
2. Caricare il programma GPU ed eseguirlo sulla GPU
3. Copia i risultati dalla memoria della GPU alla memoria della CPU

La memoria della CPU è gestita in maniera completamente differente rispetto alla memoria GPU, in quella della CPU si predilige la velocità, avere il dato il più velocemente possibile per poterlo elaborare. Sulla GPU, devo consegnare il dato a tutti i core presenti contemporaneamente.

La lettura e scrittura sulla memoria della GPU è molto lenta e costosa, quindi si tende a limitare il numero di scambi di dati tra CPU e GPU.

Il compilatore Nvidia è **nvcc**.

nvcc separa il codice sorgente in codice dispositivo e codice host, il codice dispositivo viene processato dal compilatore nvidia mentre il codice host viene processato dal compilatore standard (es. gcc)

Hello World con codice device

```

__global__ void mykernel(void) {
}

int main(void) {

    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}

```

Il CUDA, la parola chiave `global` indica che la funzione verrà eseguita sulla GPU ma la chiamata verrà fatta dalla CPU nel codice host.

`mykernel<<<1,1>>>();` è la chiamata dal codice host al codice device, questo è tutto il necessario per eseguire una funzione sulla GPU.

Somma di interi in CUDA

```

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main(void) {

    int a, b, c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;

    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);
}

```

```

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}

```

Dovremo usare dei puntatori per le variabili, la funzione **add()** verrà eseguita sul device, quindi **a,b, c** dovranno puntare alla memoria del device.

Le memoria host e device sono entità separate, i puntatori device puntano alla memoria GPU mentre i puntatori host puntano alla memoria della CPU.

Mentre eseguo codice GPU, non posso allocare memoria, a,b, c devono essere allocati nella memoria GPU, questo viene fatto dalla CPU tramite funzioni CUDA.

```
cudaMalloc(), cudaFree(), cudaMemcpy()
```

L'elaborazione GPU riguarda il parallelismo massivo quindi invece di eseguire la funzione add() una volta la eseguiremo N volte modificando la chiamata della funzione nel codice host.

```

add<<< 1, 1 >>> ();
^
add<<< N, 1 >>> ();

```

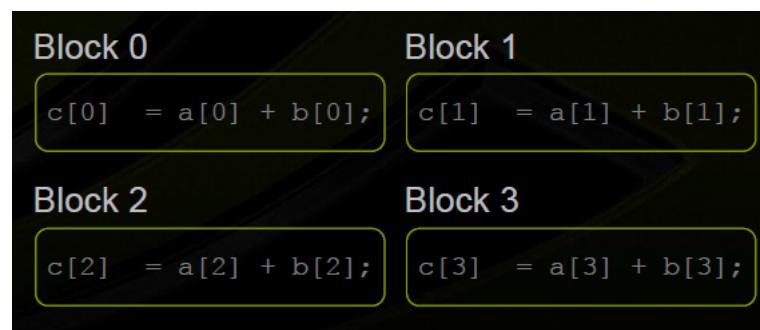
Ogni invocazione parallela della funzione add(), è riferita ad un **blocco di lavoro**, un insieme di blocchi di lavoro è chiamato **griglia**.

Ogni chiamata può fare riferimento al relativo indice di blocco utilizzando **blockIdx.x**

```

__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

```



Quindi il programma parallelo con N blocchi diventa:

```

#define N 512

__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

int main(void) {

    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}

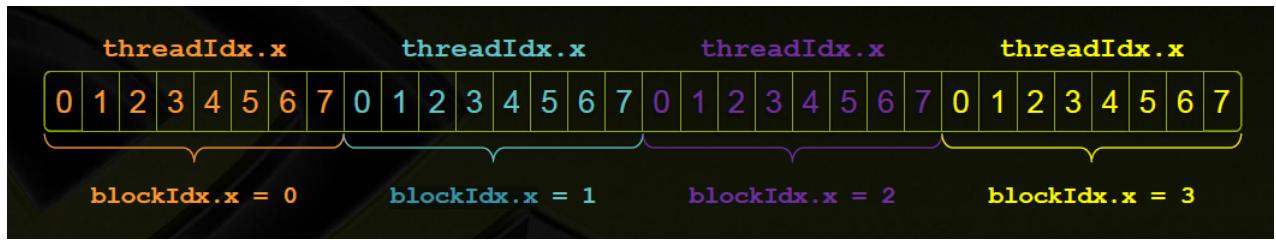
```

Un blocco può essere suddiviso in thread paralleli, usando threadIdx.x come indice. Il secondo parametro della chiamata add() rappresenta il numero di thread per blocco
(add<<<N_blocchi,N_thread>>>)

Specifichiamo inoltre che un blocco può contenere al massimo **1024 thread**.

Per indicizzare correttamente un array usando threadIdx.x e blockIdx.x dobbiamo usare l'indice del thread come offset sommato al numero di blocchi totale per l'indice del blocco considerato.

Esempio:



Se M è uguale al numero di thread per blocco di lavoro allora l'indice per ogni thread è dato da:

```
int index = threadIdx.x + blockIdx.x * M;
```

Il valore di M è contenuto nella variabile built-in **blockDim.x**

Quindi il programma completo con più blocchi e più thread per blocco è il seguente:

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

__global__ void add(int *a, int *b, int *c)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}

int main(void) {

    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```

cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}

```

Un problema tipico è quando abbiamo un array di dati che non è perfettamente divisibile per il numero di thread per blocco. Se ad esempio il nostro array è di 50 elementi, e abbiamo 7 thread per blocco, avremo bisogno di 8 blocchi, i primi 7 pieni mentre servirà solo un thread per l'ottavo blocco. Mettendo un IF, che controlla che l'indice sia minore di N (ovvero il numero degli elementi dell'array) faremo eseguire il codice solo quando saremo all'interno dell'array.

Inoltre possiamo calcolare quanti blocchi saranno necessari per completare l'intero array con il seguente giochetto matematico nella chiamata della funzione

```

__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}

```

e quindi cambiare la chiamata del kernel (una funzione GPU si chiama kernel)

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

SM e architettura GPU

I processori di calcolo (GPU) sono divisi in più **SM (Streaming Multiprocessor)**, a seconda delle caratteristiche della GPU avremo un numero diverso di SM.

Esempio: le Nvidia P100 hanno 56 SM e 3584 Cuda cores, esecuzioni reali che possiamo eseguire in modo parallelo.

Ogni SM è in grado di elaborare un blocco di lavoro, nella P100 posso elaborare 56 blocchi di lavoro in parallelo. Ovviamente poi ho i thread che possono lavorare su una determinata attività ben precisa sui miei dati.

Il massimo numero di thread per blocco è 1024 ma io non ho 1024 ALU per eseguirli parallelamente, quindi anche i thread vengono raggruppati in gruppi di 32, questo gruppo di 32 thread è chiamato **WARP**.

È consigliato avere pochi blocchi e tanti thread per blocco <<1,N>> o tanti blocchi e pochi thread <<N,1>>? Vogliamo avere più blocchi per dare lavoro a più SM e tanti thread per massimizzare il parallelismo.

- Un fattore da considerare è che ogni SM ci dà a disposizione un certo numero di registri, se ogni thread necessita molti registri potremmo terminare quelli disponibili e dover utilizzare la **memoria globale (RAM)** che risulta molto lenta, quindi abbassando il numero di thread per blocco potremmo starci.
- Un altro fattore riguarda il sincronismo dei thread dello stesso blocco, i thread all'interno dello stesso blocco possono essere sincronizzati mentre i blocchi tra loro non possono essere sincronizzati, inoltre è presente una **memoria interna al blocco (memoria condivisa)** accessibile dai thread del blocco.

Condivisione dati tra Threads

La condivisione dei dati fra thread all'interno dello stesso blocco di lavoro è possibile usando una memoria interna al blocco chiamata **shared memory**. Questa memoria è molto veloce, e funziona come una cache gestita dal programmatore.

Si dichiara usando la keyword **__shared__**, allocata per blocco, quindi i dati di questa memoria non sono visibili ai thread appartenenti ad altri blocchi.

Esempio di uso della shared memory:

```
__global__ void stencil_1d(int *in, int *out) {

    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
```

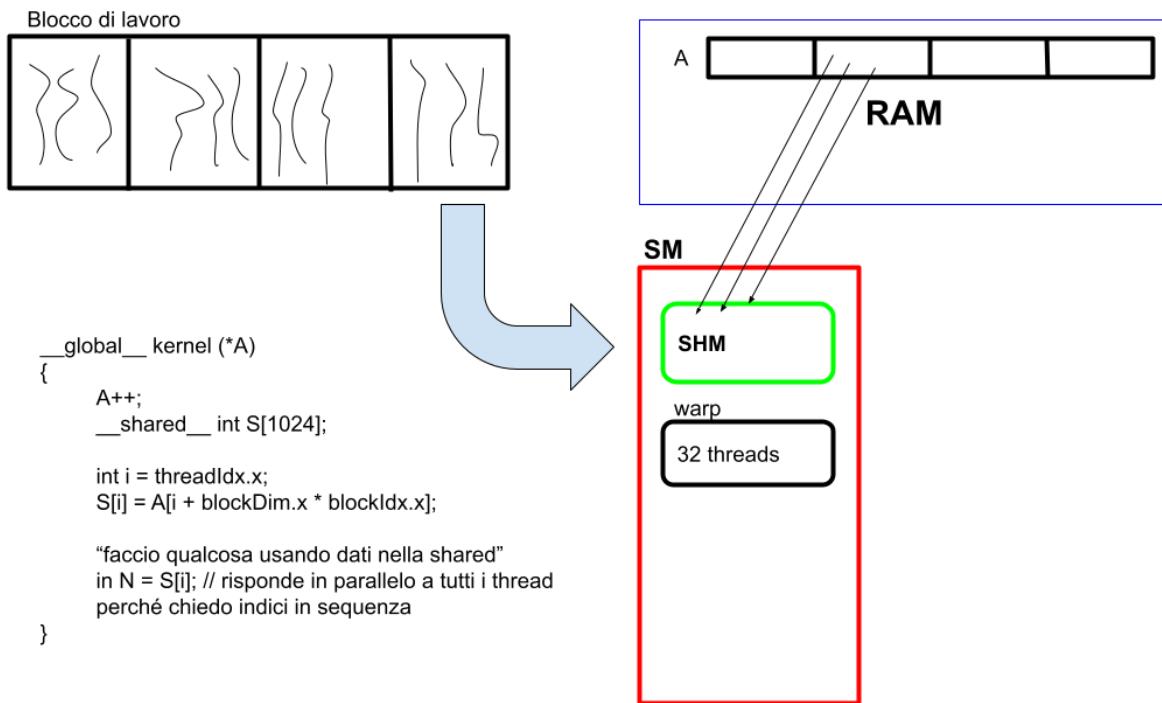
```

    // Store the result
    out[gindex] = result;
}

```

In questo esempio, viene allocato da ogni thread un vettore nella memoria shared su cui successivamente ci vengono copiati i dati presenti sulla memoria ram per poi sommare i dati presenti nelle variabili con indici adiacenti ed essere messi nel vettore di output. Questo consente di non effettuare ripetuti accessi alla memoria RAM ma di usare la memoria shared molto più economica.

Le letture sono tutte parallele, ogni thread copia nella memoria shared un solo dato corrispondente all'indice del suo thread.



Data Race

Facciamo un esempio per capire subito di cosa parliamo:

```

shr[i] = ram[i + i + blockDim.x * blockIdx.x];
A= shr[i];

```

In questo caso si nasconde un conflitto, c'è una dipendenza dei dati. Dato che l'esecuzione è in parallelo tramite i warp, ad esempio ho 1024 threads che eseguono questo codice, ma i warp vengono schedulati, non vengono eseguiti 1024 thread

contemporaneamente quindi è possibile che alcuni warp terminino prima di altri e passino all'istruzione successiva. Ma se l'istruzione successiva dipende da dati che non sono ancora stati copiati abbiamo un errore.

Come possiamo risolvere? Con la **sincronizzazione**.

Imponendo una barriera dopo l'istruzione di copia impediamo che alcuni thread proseguano se tutti i thread non hanno terminato l'istruzione precedente. Quando tutti i thread avranno terminato l'istruzione di copia, allora sarà possibile continuare l'esecuzione del kernel.

Per fare questo viene introdotta la funzione `void __syncthreads();`

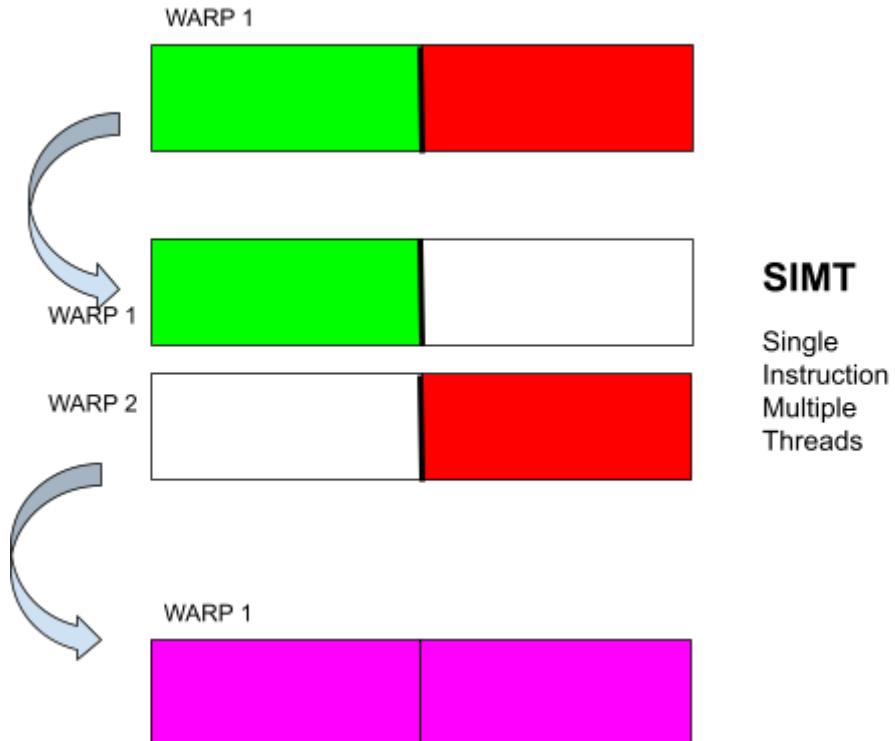
Un altro caso in cui è utile usare questa funzione è quando viene diviso il flusso di esecuzione con un IF. Alcuni thread magari non devono eseguire un determinato codice e quindi proseguono la computazione. Con una barriera possiamo impedire che si perda il sincronismo e quindi riunire il flusso.

```
if(threadIdx.x < 10)
{
    // fai questo
}
// altrimenti continua
_____  
// riuniamo i flussi di esecuzione
void __syncthreads();
```

Noi cerchiamo di fare un unico flusso parallelo ma sulla GPU posso anche differenziare, fare branching del codice, far fare ad ogni thread una cosa diversa una cosa diversa.

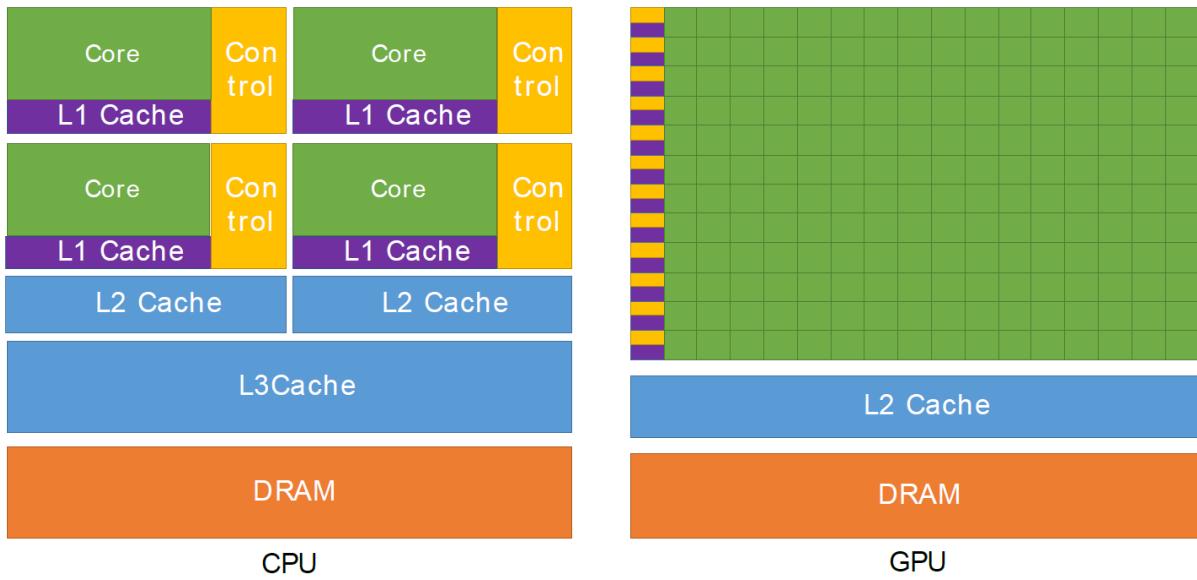
Se tutti fanno la stessa, un'istruzione e me la cavo, se i thread anche dello stesso warp iniziano a fare cose diverse cosa faccio?

Esempio:



Se ho alcuni thread che eseguono codice diverso, viene fatto branching del flusso, il warp verrà diviso in due warp diversi, nel primo verranno eseguiti tutti i thread che eseguono un flusso, mentre nel secondo avrò gli altri che seguono l'altro flusso. In questo modo lo scheduler riesce a parallelizzare al meglio i thread che eseguono lo stesso codice. Poi infine se ho ad esempio una barriera, potrò riunire i flussi in un unico warp e continuare l'esecuzione.

Immagine che illustra le differenze di architettura tra CPU e GPU, i componenti in verde indicano le ALU, in viola nella GPU indicano la shared memory.



Quando vogliamo specificare struttura dei blocchi e dei thread, quando facciamo la chiamata al kernel, oltre alla classica che abbiamo visto in una sola dimensione <<N,N>>, è possibile anche specificare fino ad un massimo di 3 dimensioni, se vogliamo che ad esempio un blocco sia organizzato in matrici, usiamo il tipo dim3:

```
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Il calcolo si dice eterogeneo, se una parte del calcolo viene svolto sulla CPU è una parte sulla GPU.

Con il termine **stream** viene inteso un flusso di lavoro, i flussi di lavoro sono indipendenti, e possono essere parallelizzati, se uno stream occupa una parte della potenza di calcolo e un secondo stream occupa la restante parte, possono essere schedulati ed eseguiti in parallelo.

Per calcolare il tempo trascorso, cuda mette a disposizione alcune funzioni comode:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
//do some stuff...
cudaEventRecord(stop, 0);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Strategie di ottimizzazione delle performance

- Massimizzare la parallelizzazione
- Ottimizzazione accessi in memoria shared e globale, usare SHM e non RAM
- Ottimizzazione uso istruzioni e warp, ridurre il branching del codice

Sfruttare le tipologie di memorie disponibili, gli accessi in ram sono molto costosi, la shared memory è molto più veloce e economica. Ci sono anche altri tipi di memoria specifiche come ad esempio la texture memory. La texture memory è un tipo di memoria situata sul dispositivo, una cache ottimizzata per spazi di locazione 2D, il costo per l'accesso nella memoria del device è di una lettura solo se il dato non è presente nella cache. I thread dello stesso warp che leggono indirizzi texture adiacenti avranno una migliore prestazione.

Tabella riepilogativa dei tipi di memoria sulla GPU:

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes ^{††}	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

[†] Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

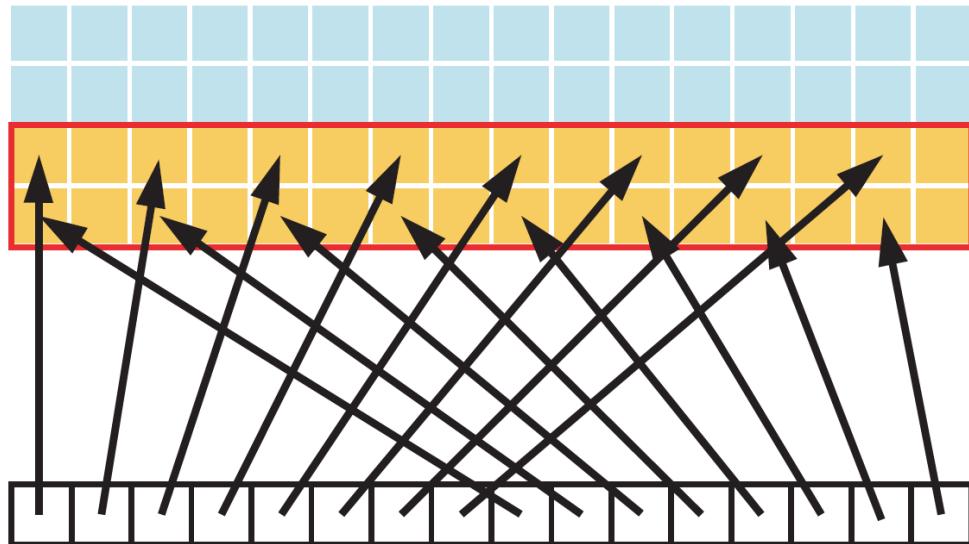
^{††} Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Accesso coalesced alla memoria globale

L'accesso di tipo coalesced è un accesso alla memoria globale in modo organizzato, strutturato, l'organizzazione è a blocchi di 32 byte, quindi una transazione consente di leggere contemporaneamente 32 byte di memoria.

Ad esempio, se i thread di un warp(32 thread) accedono a parole a 4 byte adiacenti (ad esempio valori float adiacenti), serviranno 4 transazioni coalesced a 32 byte per servire quell'accesso alla memoria.

Per questa ragione ad esempio è consigliato non allocare array di struct ma struct di array perché se dovessi accedere ad una variabile delle struct i valori non si troverebbero più adiacenti e quindi i tempi di accesso aumentano drasticamente.



L'occupancy è una metrica per sapere quanto è efficiente l'hardware, questa metrica è data dal rapporto tra il numero di warp attivi per multiprocessore e il numero massimo di warp contemporanei supportati. Se un warp è in stallo o in pausa, per ridurre la latenza e mantenere l'hardware occupato, Cuda esegue altri warp durante l'attesa.

Calcolo prodotto matrici

K = numero righe di A o colonne di B

IDEA NATIVE:

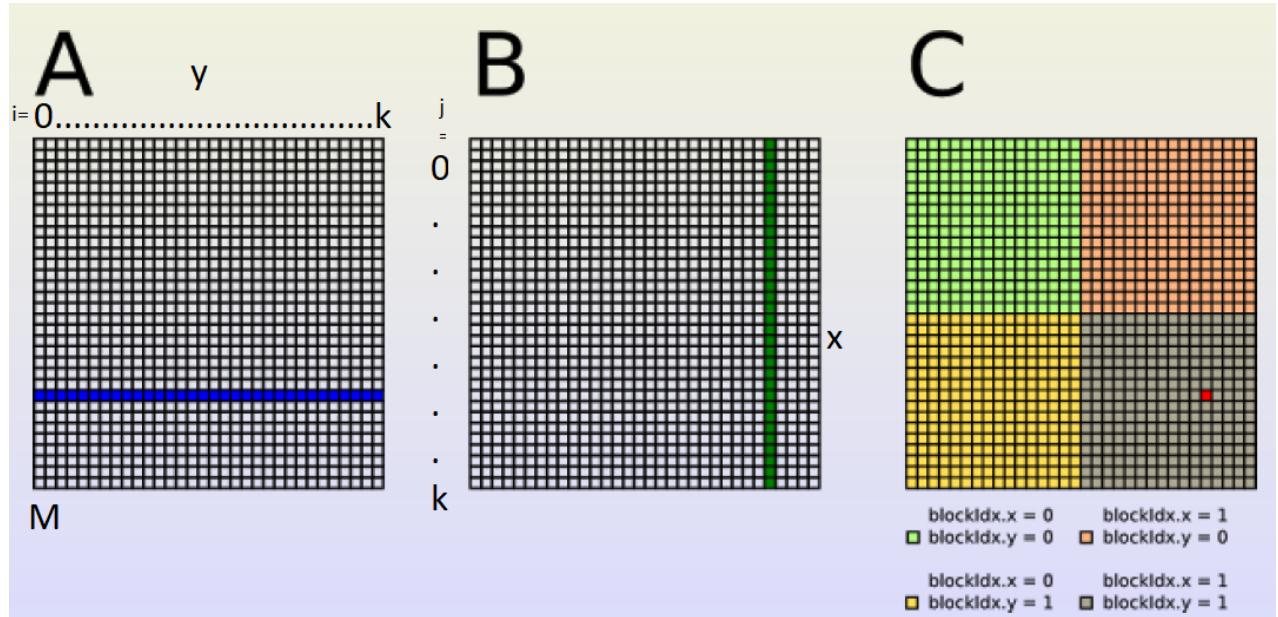
Ogni blocco contiene 32x32 thread, in cui ogni thread calcola il prodotto scalare dei vettori della riga della matrice A con la colonna della matrice B e lo mette nella cella corrispondente della matrice C.

```

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
    accu = 0
    // Row i of matrix C
    i = blockIdx.y * blockDim.y + threadIdx.y
    // Column j of matrix C
    j = blockIdx.x * blockDim.x + threadIdx.x
    for k = 0 to K-1 do
        accu = accu + A_gpu(i,k) * B_gpu(k,j)

        C_gpu(i,j) = accu
}

```



In questo caso, in un blocco, all'interno del for $0 \dots k$, noi nella matrice B scorriamo tutte le righe della colonna j , nei vari thread cambiamo la colonna, ma nella matrice A, in altri thread dello stesso blocco, l'indice i rimane lo stesso per tutta le colonne che i thread scorrono, quindi facciamo moltissimi accessi alla stessa cella leggendo lo stesso identico dato. Questo è abbastanza inutile. Vengono effettuate K letture in ram della stessa cella.

quindi:

Critiche: troppe letture da memoria globale

Soluzione: sfrutto la memoria condivisa del blocco

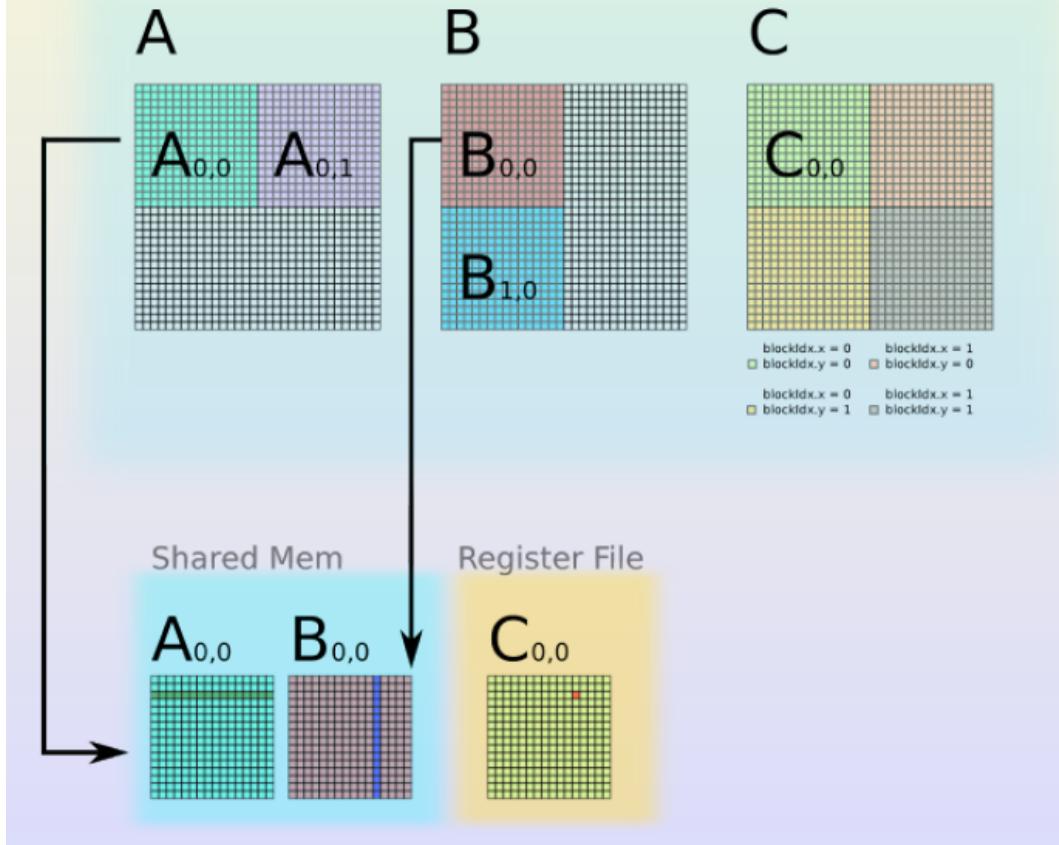
Evito di leggere tante volte gli stessi dati

Organizzo il calcolo parallelo in modo diverso

IDEA SHARED MEMORY:

- Divido le matrici in blocchi (tiles)
- Calcolo ogni blocco di C in piu' fasi
- Ad ogni iterazione, ogni thread copia un blocco di A e uno di B dalla memoria globale in shared
- Ogni thread calcola il prodotto e aggiorna il risultato in un registro
- Al termine delle iterazioni, tutti i thread memorizzano il loro risultato (blocco di C) in memoria globale

Global Memory



```

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
    __shared__ float A_tile(blockDim.y, blockDim.x)
    __shared__ float B_tile(blockDim.x, blockDim.y)
    accu = 0

    for tileIdx = 0 to (K/blockDim.x - 1) do
        //carica blocchi di A e B in shared mem
        i = blockIdx.y * blockDim.y + threadIdx.y
        j = tileIdx * blockDim.x + threadIdx.x
        A_tile(threadIdx.y, threadIdx.x) = A_gpu(i,j)
        B_tile(threadIdx.x, threadIdx.y) = B_gpu(j,i)
        __sync()

        for tileIdx = 0 to (K/blockDim.x - 1) do
            //carica blocchi di A e B in shared mem
            // Prodotto scalare (accumulato)
            for k = 0 to blockDim.x do
                accu = accu + A_tile(threadIdx.y,k) * B_tile(k,threadIdx.x)
            end
            __sync()
}
  
```

```

    end
//scrive il blocco in global
i = blockIdx.y * blockDim.y + threadIdx.y
j = blockIdx.x * blockDim.x + threadIdx.x
C_gpu(i,j) = accu
}

```

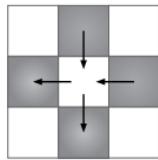
Propagazione calore CUDA

La funzione che ci permette di aggiornare i valori e quindi di propagare il calore in questa simulazione è:

$$T_{NEW} = T_{OLD} + k \cdot (T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4 \cdot T_{OLD})$$

quindi avremo due matrici, una T_{old} da cui leggeremo i dati e T_{new} in cui metteremo i nuovi dati calcolati.

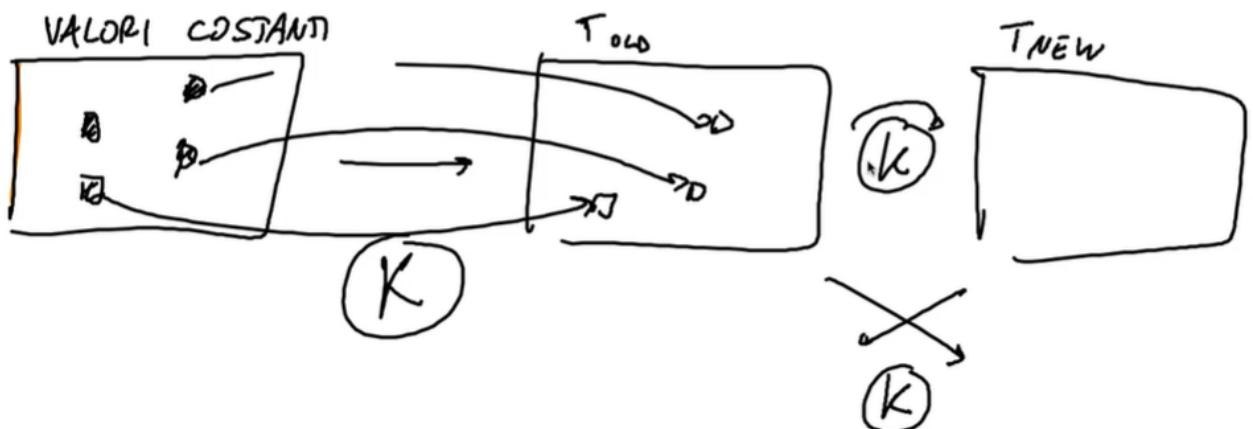
Il nostro stencil quindi è della forma



Alcune celle della nostra matrice dovranno avere dei valori fissi e costanti ad ogni calcolo, saranno le nostre "fonti" di calore.

Un kernel dedicato alla copia dei valori delle celle costanti sulla nostra T_{old} .

Un kernel che effettivamente calcola i nuovi valori di propagazione dalla T_{old} e li memorizza nella T_{new} . Abbiamo bisogno di due kernel per evitare problemi di sincronizzazione, separando in due kernel queste operazioni, possiamo sincronizzare le letture e le scritture, ne richiamo uno, aspetto che esegua poi dal main sincronizzo la fine e poi mando in esecuzione il secondo kernel.



Utilizzando due matrici, evitiamo tutti i problemi legate a letture e scritture, le letture vengono effettuate tutte e solo dalla matrice T_{old} e le scritture avvengono tutte e solo sulla matrice T_{new} . Finita ogni iterazione, T_{old} e T_{new} vengono scambiate così da leggere i nuovi valori e proseguire la propagazione.

```

__global__ void blend_kernel( float *outSrc, const float *inSrc ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;

    outSrc[offset] = inSrc[offset] + SPEED * ( inSrc[top] + inSrc[bottom] +
                                                inSrc[left] + inSrc[right] -
                                                inSrc[offset]*4);
}

```