Motivazioni dell'HPC – Problemi "grand challenge"

L'HPC è volto a risolvere problemi numerici di notevole impatto in diversi ambiti risolvibili computazionalmente ma che richiedono numerose risorse di calcolo e storage per dare una soluzione in un tempo ragionevole, sfruttando le risorse disponibili, evitando colli di bottiglia e trasformando i problemi in algoritmi paralleli.
In una prima fase di analisi hardware, algoritmo e applicazioni devono coordinarsi per garantire buone prestazioni.

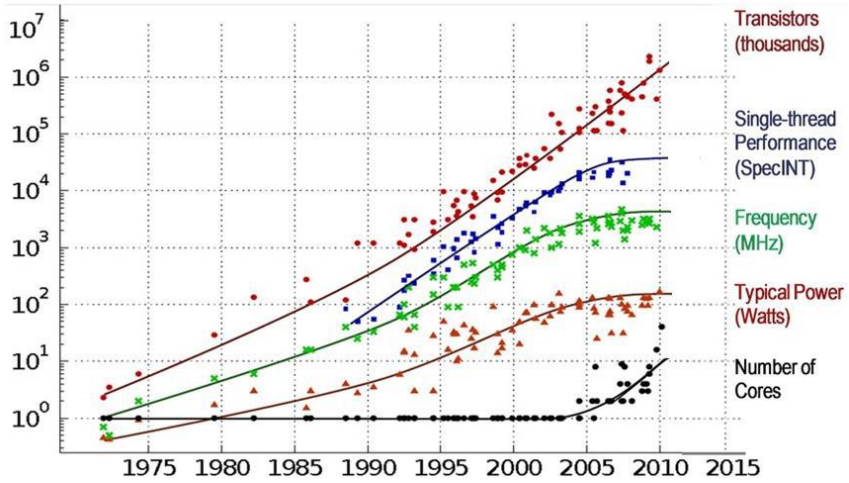
Dopo l'analisi del problema si ha una fase di implementazione attraverso lo sfruttamento di 3 soluzioni Hardware

- Memoria condivisa
  Diverse unità che condividono lo stesso spazio di indirizzamento, implementata con strumenti come thread e OpenMP
- Memoria distribuita
  Processi indipendenti sulla stessa macchina o su macchine diverse con spazi di indirizzamento indipendenti, implementata con strumenti come mpi in cui si ha un'interfaccia di comunicazione basata sullo scambio di messaggi
- GPU
  L'accelerazione più importante nelle prestazioni è stata introdotta da questa soluzione, che ha un modello di programmazione specifico implementato con la libreria CUDA

Queste librerie possono essere utilizzate in diversi ambienti di programmazione ma il loro ambiente nativo è rappresentato da linguaggi a basso livello come il C.
Esempi di applicazioni sono l'ambito scientifico, la difesa e la crittografia (fattorizzazione chiavi RSA con la factory challenge), data science ecc.
I problemi Grand Challenge non sono eseguibili su macchine tradizionali, perché il tempo di calcolo è talmente elevato che richiederebbe mesi, e in ambiti come la climatologia il risultato di calcolo deve essere immediato con una analisi evolutiva nel tempo di diversi parametri.

Hardware performance – La legge di Moore i suoi limiti

La legge di Moore (1965) ipotizza che le prestazioni dei microprocessori raddoppino ogni 18 mesi.



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 dimensioni dei transistor. La frequenza di lavoro (curva verde) 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.

Livelli di parallelismo

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 (multicore, multisocket), utilizzando acceleratori (e.g. GPU), tra più nodi di un cluster. Negli ultimi anni queste tecnologie stanno avendo una rapida evoluzione.

Efficiency trand and Efficiency Gap

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 hardware disponibili.

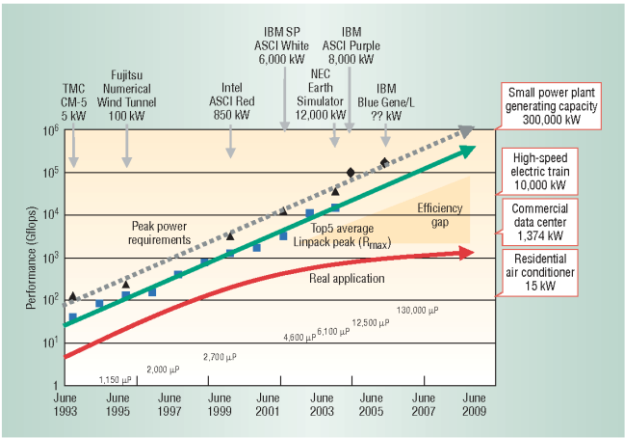


Figure 1. Supercomputer performance and power trends. Scientific applications' growing computational demands have led to exponential increases in system peak performance, power consumption, and complexity requirements.

TOP500

Top500 è un sito in cui vengono elencati i 500 calcolatori più potenti del mondo. Attualmente in prima posizione c'è il supercomputer Fugaku dal Giappone. Notiamo che i calcolatori più potenti sono quelli che utilizzano GPU.

GREEN500

Dal 2012 viene stilata anche la classifica in base all'efficienza energetica ottenuta riordinando la top500.

Consumo energetico

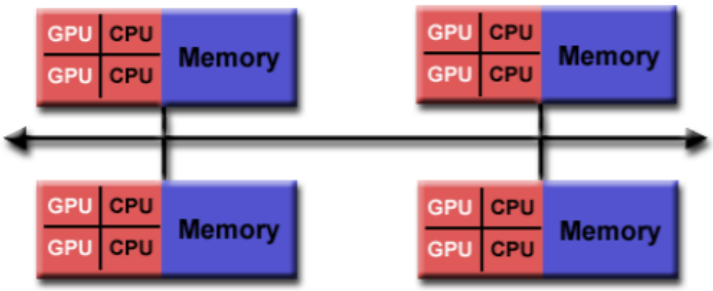
Il consumo energetico dei sistemi HPC cresce di pari passo, esponenzialmente, con la potenza di calcolo.

- 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)
  Rapporto tra la potenza totale (PT) assorbita dal data center e quella usata dai soli apparati IT (PIT) ovvero PUE = PT / PIT
  La quota aggiuntiva di consumo energetico è principalmente dovuta al raffrescamento.
  PUE rappresenta una misura di quanto efficiente sia un centro di calcolo, o data center, nell'usare l'energia elettrica che lo alimenta. È considerato efficiente se inferiore al 1,8.

Architetture parallele

Nei sistemi di calcolo moderni l'accelerazione (speedup) 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 da thread o processi eseguiti sui core della macchina)
- Diversi processi su più nodi interconnessi (memoria distribuita)
- Kernel di calcolo eseguiti su acceleratori (GPU) o coprocessori



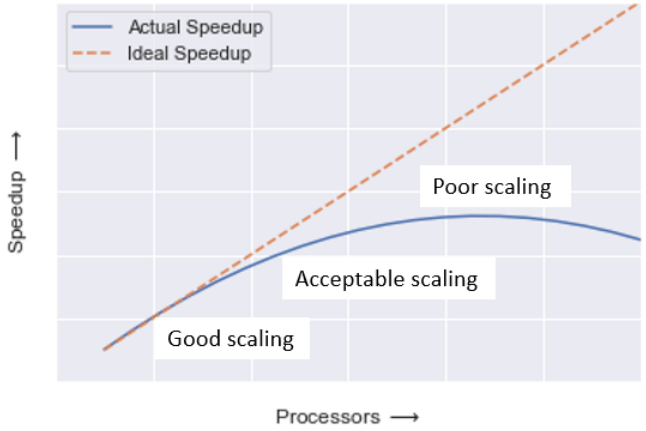
Un datacenter HPC e un datacenter generico hanno poche differenze dal punto di vista esterno, la differenza sta nella rete, in quanto negli HPC il problema è distribuito su diversi nodi di calcolo e l'integrazione dei processi deve essere più stretta necessitando di una rete più veloce, e nell'accelerazione del calcolo migliorata negli ultimi anni principalmente con le GPU.

Dal punto di vista software nel calcolo parallelo servono applicazioni distribuite e ad alte prestazioni. Si ha una scalabilità di problemi, da quelli con bassi requisiti di prestazioni a quelli che necessitano numerosi nodi di calcolo.
Per collegarsi al cluster si passa attraverso un firewall per poi arrivare su una macchina di log-in, con un gestore di code slurm. Ci sono dei nodi con soltanto core e nodi con anche la GPU per accelerare. Non si usano direttamente i server di calcolo ma si interagisce soltanto con la macchina di log-in con la mediazione di uno slurm che fa da job manager. I job vengono sottomessi allo slurm specificando le necessità e il programma da eseguire. Slurm gestisce le risorse disponibili e le assegna dinamicamente attraverso un sistema di code. Al termine dell'esecuzione l'output, l'error e i file generati dall'esecuzione tornano sulla macchina di log-in, attraverso una modalità batch (accodamento). La comunicazione tra i vari nodi deve essere ad altre prestazioni, quindi tecnologie di rete come ethernet non sono ottimali a causa dell'alta latenza, al contrario delle più performanti come `omnipath` (Intel). Ci sono poi dischi organizzati in due categorie di storage NAS e SAN, la prima è un server che utilizza protocolli come nfs per consentire l'accesso ai dischi locali organizzati in raid, mentre il san è il più utilizzato e complesso. Per l'interazione tra i dischi e i server è utilizzato un Fibre Channel.

Designing Parallel Programs

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 di

- Organizzazione dei dati
- Componenti funzionali dell'algoritmo



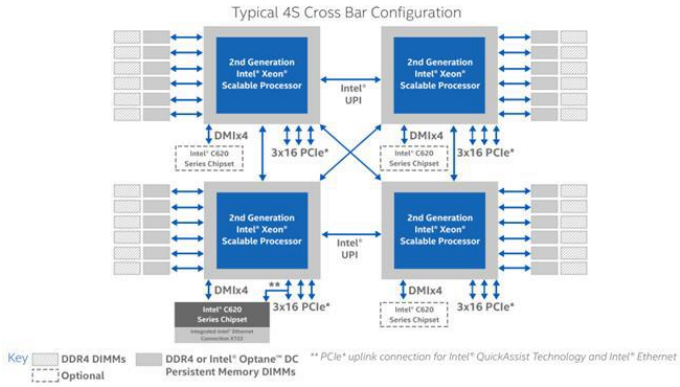
Con il termine Speedup si intende la misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato

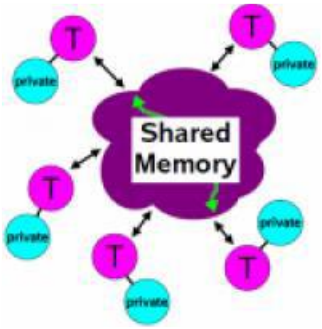
$$SpeedUp = \frac{Tparallelo}{Tseriale}$$

La 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.
Lo speedup ideale con il numero di unità di processamento utilizzabili.
La scalabilità dimostra che è inutile utilizzare troppi processori in quanto comportano un peggioramento delle prestazioni.

Programmazione a memoria condivisa

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



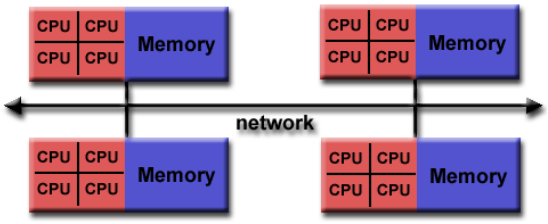


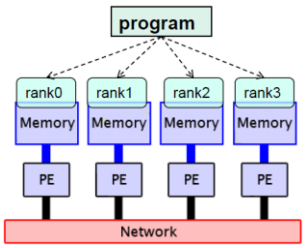
I task del programma possono essere assegnati a thread in esecuzione su differenti unità di processamento. La comunicazione tra i task avviene mediante l'accesso sincronizzato a variabili condivise.

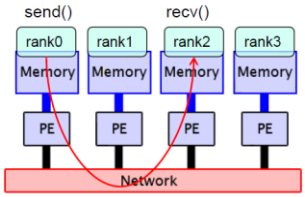
Le librerie che possiamo utilizzare per la programmazione sono Posix Threads e OpenMP (Open Multiprocessing) che è una libreria ideata specificamente per applicazioni parallele a memoria condivisa ed è supportata da vari linguaggi di programmazione come il C/C++ e il Fortran.

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)

Specifica per lo sviluppo di librerie standard di comunicazione. Le implementazioni possono essere opensource (esempio Open MPI) o commerciali (come Intel MPI).

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.
Il costruttore principale di GP-GPU è NVIDIA che fornisce un modello di programmazione CUDA, la libreria e il relativo compilatore **nvcc**.
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.

Laboratorio – Setup cluster HPC

L'accesso è consentito solo dall'ateneo o via VPN. Oltre all'accesso con password si utilizza l'autenticazione con chiavi RSA. La comunicazione tra i diversi processi sui nodi avviene via ssh, per questo si attiva la comunicazione password-less tra i nodi. Essendo la home directory condivisa dai nodi e la macchina di log-in basta inserire la chiave pubblica nel file authorized_keys.

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 performance
  - CPU e GPU
  - Memoria
  - Storage
  - Network (sistema di calcolo distribuito sul cluster)
- Software
  - Algoritmi
  - Hardware exploitation (capacità di sfruttare le risorse hardware disponibili)
  - Ottimizzazioni (es compilatori ottimizzati)

Theoretical Peak Performance

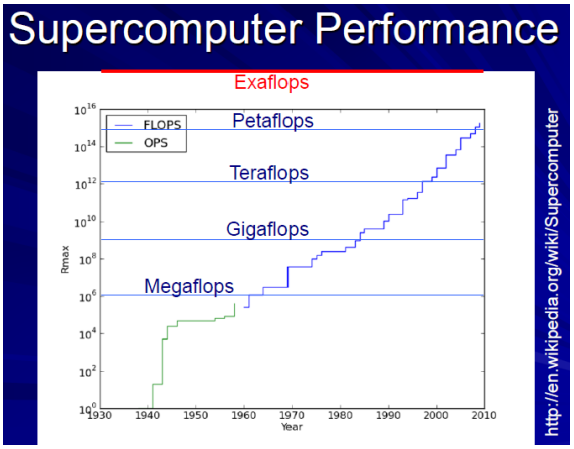
Stima della performance di un componente Hardware (CPU, memoria, rete, storage) in base alle caratteristiche architetturali.

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 perché influenzano il numero di operazioni possibili

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

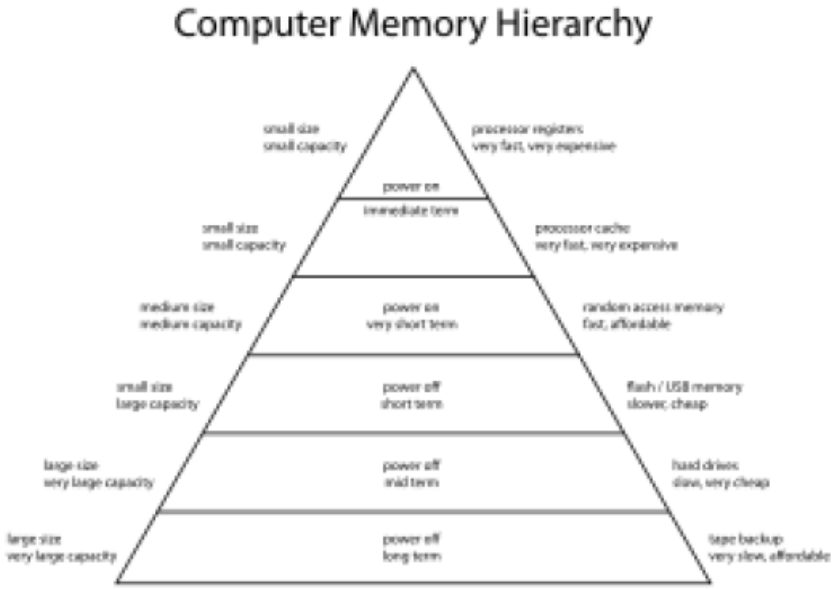
Prestazioni di un core:

$$FLOPS = Clock \cdot \frac{FLOPs}{clycle}$$

Le prestazioni sono rappresentate nel tempo attraverso una scala logaritmica. Negli anni 60 i supercalcolatori erano dell'ordine dei MFLOPS.

Memoria

La gerarchia della memoria determina tempi di accesso differenti a seconda della localizzazione del dato.

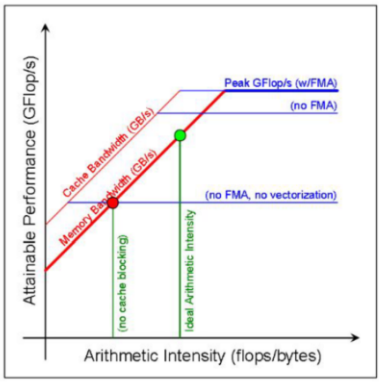


La memoria può diventare un collo di bottiglia nelle prestazioni se il processore non è in grado di fornire dati con il ritmo richiesto dal processore.

Performance della memoria - RoofLine Analysis

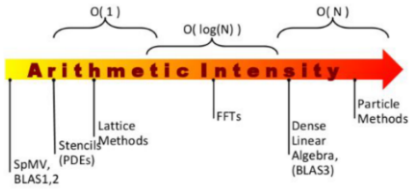
Il modello Roofline consente in maniera intuitiva di stimare le performance di un kernel computazionale mostrando graficamente le limitazioni inerenti CPU/GPU e memoria. Ogni kernel di calcolo ha la sua arithmetic intensity, ossia una metrica che definisce qual è il rapporto tra le operazioni floating point da eseguire rispetto ai byte da elaborare e dipende dall'applicazione. La macchina può avere diverse prestazioni (riga blu), senza usare istruzioni FMA o vettori abbiamo prestazioni peggiori, utilizzandoli entrambi sfruttiamo al massimo l'hardware messo a disposizione.
La curva del Memory Bandwidth varia a seconda della velocità della memoria nel fornire i dati.

Lavorando sempre in memoria senza usare la cache scendiamo nella curva. Applicazioni con arithmetic intensity bassa (pochi flop e molti dati) presentano un limite di performance dovuto alla memoria. Viceversa sfruttando la cash (in cima alla gerarchia) la comunicazione ha prestazioni più elevate.



$$\text{Attainable FLOPs/sec} = \min \begin{cases} \text{Peak FLOPs/sec}, \\ \text{Peak Memory Bandwidth} \times \text{Arithmetic Intensity} \end{cases}$$

$$\text{Arithmetic Intensity} = \frac{\text{Total FLOPs}}{\text{Total Bytes}}$$



Deslippe et al., "Guiding Optimization Using the Roofline Model," tutorial presentation at IXPUG2016, Argonne, IL, Sept. 21, 2016.

Storage

Tecnologie per i dispositivi di storage

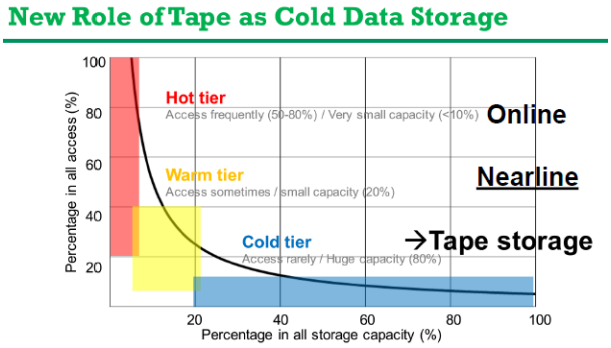
Tecnologie dischi esempi	Capacità TB	Prestazioni MB/s (tipico)	Costo K€
SSD Seagate (1)	7.6	600/800	2.7
HDD Seagate (1)	20	280	0.6
TAPE Cartridge HPE LTO8 (2)	30	3.6TB/hour	0.1

Valori tipici, costi indicativi

Le applicazioni che fanno accesso continuativo al disco necessitano di un disco veloce al contrario di quelle che effettuano calcoli direttamente in memoria. Le tecnologie utilizzabili sono di tre tipi

- SSD, poca capacità ma alte prestazioni e costo alto
- HDD, capacità media, calo nelle prestazioni e nel prezzo
  - TAPE Cartridge, una cartuccia che messa in un lettore permette l'accesso in lettura e scrittura, occorre di una macchina che gestisca le cartucce lette in

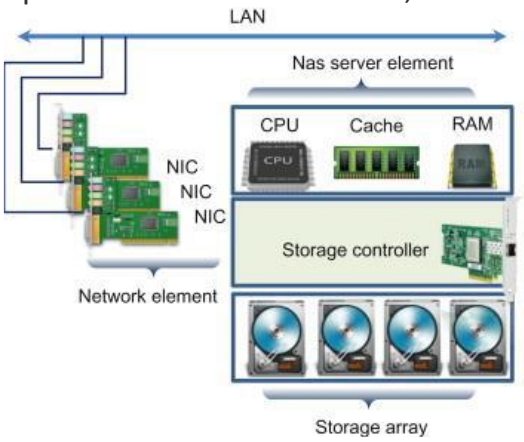
modo sequenziale. Serve per l'archiviazione.



In un sistema di calcolo risultano utili per l'80% Cold tier, ovvero dati utilizzati raramente, per un 10% si hanno dati con un accesso continuativo (hot tier) e warm tier con dati online ma accesso saltuario.

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 i SCSI.



Vantaggi

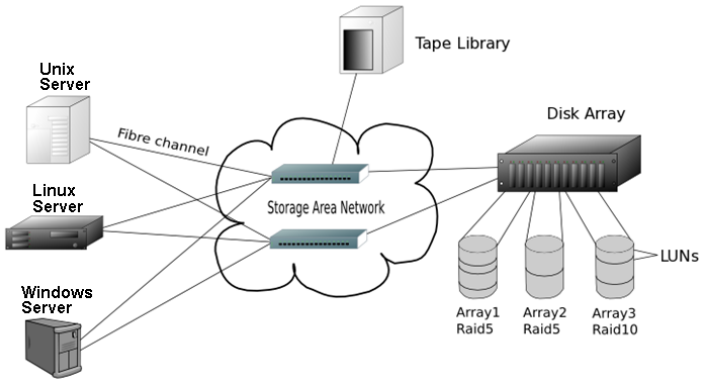
- Condivisione dati
- Gestione centralizzata
- Basso costo

Svantaggi

- Basse prestazioni
- Risorse limitate

SAN – Storage Area Network

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



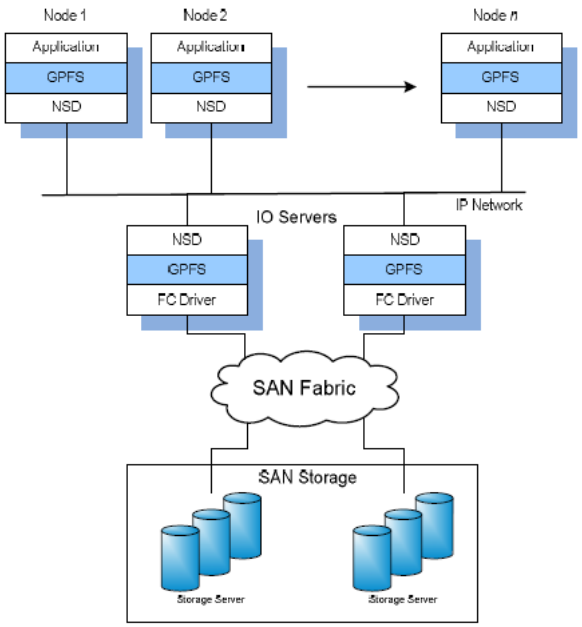
Vantaggi

- Condivisione dati
- Gestione centralizzata
- Basso costo
- Alte prestazioni

Svantaggi

- Scalabilità
- Ridondanza

GPFS – Clustered File system



GPFS è un parallel network shared file system di proprietà dell' IBM in grado di gestire storage server e distribuzione dei dati al client.
E' costituito da una collezione di dischi su cui GPFS memorizza dati e metadati.

L'architettura prevede fino a 8 NSD server. Un solo NSD server è attivo in un dato istante.
Caratteristiche principali:

- Failover
  In caso di server failure viene servito da un altro NSD server.
- 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 con le diverse tecnologie, portando verso l'alto gli hot tier

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

valori tipici, costi indicativi

Necessità banda elevata e latenza bassa, ma problema di scalabilità.

Sustained performance – 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)
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.

Profilazione dei tempi di esecuzione - `time`, `gprof` e `clock_gettime()`

Il comando time ritorna i tempi di esecuzione di un programma.
Esempio:

```
> 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.
È possibile determinare il wall clock time (`CLOCK_REALTIME`) oppure il tempo di utilizzo della CPU (`CLOCK_PROCESS_CPUTIME_ID`)

Analisi e visualizzazione dei dati con Python - pandas e matplotlib

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.
L'utilizzo congiunto di pandas e matplotlib consente di creare semplici script python per la gestione e la visualizzazione dei dati prodotti dall'esecuzione dei programmi di calcolo.

Laboratorio – Ambiente del cluster HPC

Scratch: finalità ad alte prestazioni. Utilizzo del comando time e del profiler `gprof`.

Programmi utilizzati
`cpi.c`
Programma per il calcolo del pigreco utilizzando due diversi metodi di calcolo, con due funzioni, basate sull'integrazione numerica. F1 impiega la metà del tempo rispetto ad F2.

Script in shell che fa assumere ad n diversi valori inseriti su command line, sui quali esegue `cpi.c` ottenendo così uno scaling che mette in relazione l'errore ottenuto rispetto al tempo di calcolo. Si esegue poi un plot con la libreria pandas utilizzando il file csv generato dallo script. Dagli stessi dati IMB è possibile estrarre latenza e banda. L'ambiente software è complesso e viene semplificato da un'organizzazione in moduli. Nello script di esecuzione bisogna anteporre il caricamento del modulo necessario affinché venga reso disponibile per il programma (`module load`).

L’accesso alle risorse di calcolo è mediato da un workload manager, in questo caso slurm. Il gestore di coda ha a disposizione una mappa delle risorse che indica se sono libere o occupate, una volta ricevuta una richiesta la consulta ed eventualmente la assegna, altrimenti il programma resta in attesa fino a quando non si libera la risorsa. È possibile dettagliare il tipo di nodo. Vengono usati i nodi CPU (qualunque tipo di nodo indipendentemente dall’architettura) e GPU (scheda grafica).

È possibile esaminare lo stato di nodi, partizioni e code con il comando `hpc`, in particolare con `hpc-squeue` si esaminano tutti i job accodati dallo slurm e il loro stato (ad esempio R running indica che il programma è in esecuzione, PD pending indica che i programmi sono in attesa).

La priorità dei job in coda è determinata da 3 fattori:

- Timelimit
  I job che richiedono meno tempo hanno priorità maggiore
- Aging
  I job in attesa da lungo tempo hanno priorità maggiore
- Fair share
  Chi ha usato meno risorse negli ultimi 14 giorni ha priorità maggiore

Al crescere del numero dei processori aumenta l’accelerazione del programma idealmente, in realtà aumentando i processori potremmo decelerare producendo overhead. Lo scopo è individuare gli overhead e risolverli.

Ogni job in coda ha un account, un utente può avere più account. Hpc -report fornisce un report di utilizzo del cluster, con partizione utilizzata, numero di job e nodi. Le statistiche sono prodotte in un periodo di due settimane, e classificate in base al nome utente (è possibile però classificarle in base alle partizioni).

Per sottomettere i job è necessario uno shell script in cui in testa vengono inserite le direttive per slurm. Ogni riga dell’intestazione ha il formato

```
#SBATCH   <direttiva>  <opzioni>
```

Al termine dell'esecuzione vengono generati due file che conterranno standard output e error del programma eseguito. Le direttive sono diverse, e il doppio ## le disabilita.

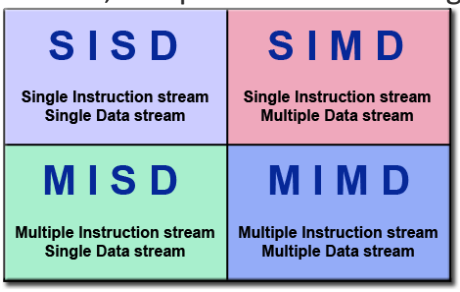
`cp1.bash` utilizza la prima funzione per il calcolo del pigreco producendo come output

- Numero della funzione
- Numero di intervalli utilizzati
- Valore di pigreco calcolato
- Errore prodotto
- Tempo calcolato utilizzando `gettime`.

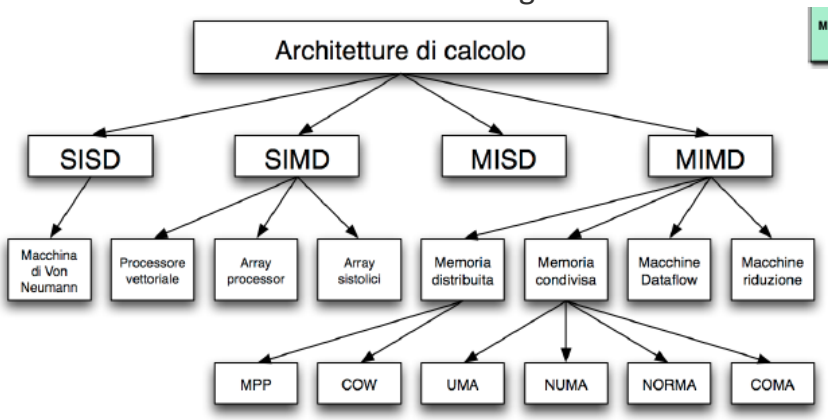
Lo script `cpi1.slurm` possiede le direttive prima elencate per indicare la partizione da utilizzare, dove inserire std input e output, il numero di nodi richiesti, il massimo tempo di calcolo e l'account da utilizzare.
`cpi_jobarray.slurm` è uno script che facilita la parallelizzazione, il gestore di code consente di specificare una sequenza di valori (in questo caso 1 e 2) che indica quante sottomissioni avverranno (una con valore 1 per la prima funzione e una con valore 2 per la seconda). Si calcola poi lo scaling attraverso lo script che genera il file csv.
Con un programma python si genera un plot che per rappresentare graficamente i risultati dello scaling.

Tassonomia di FLYNN

Ci sono differenti modi per classificare i sistemi per il calcolo parallelo. Uno dei più usati è stato introdotto da Flynn nel 1966.
Questa tassonomia è basata su una tabella a due dimensioni indipendenti.
Il flusso di Istruzioni e il flusso di Dati, che possono essere Single o Multiple.

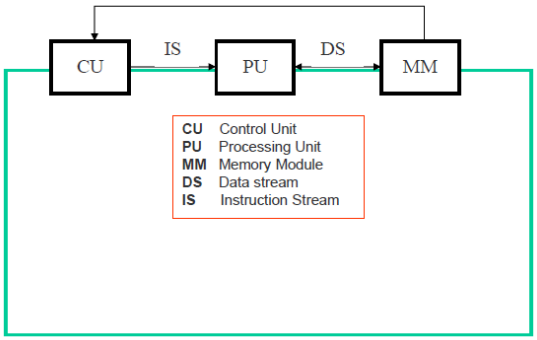


Gli elementi della tabella hanno a loro volta sottocategorie



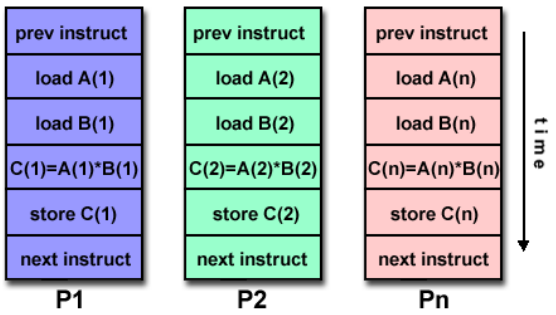
SISD – Single Instruction Single Data

Un solo flusso di istruzioni eseguito dalla CPU e un solo flusso di dati
Sono i sistemi seriali (classica architettura di Von Neumann).



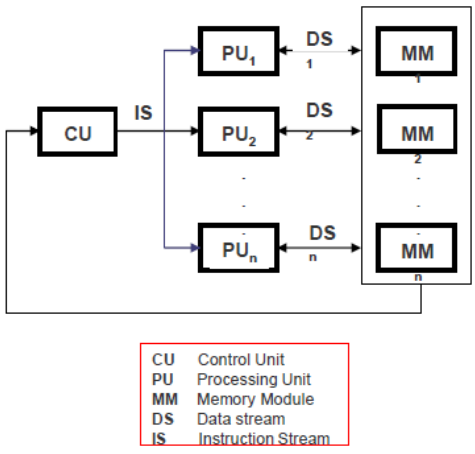
SIMD – Single Instruction Multiple Data

Un solo flusso di istruzioni eseguito dalla CPU.
Più flussi di dati elaborati contemporaneamente con una singola istruzione che opera simultaneamente su più dati.



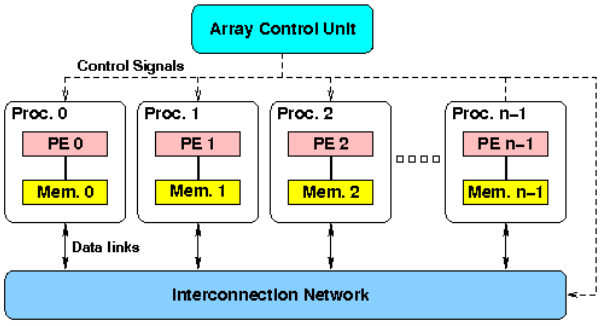
Principali tipologie SIMD

- Processori vettoriali
  - Array di elementi di elaborazione che condividono l'unità di controllo
  - Le istruzioni sono distribuite in parallelo a tutte le PU
  - 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
  - Fattore critico, banda di memoria offerta alle Unità Funzionali



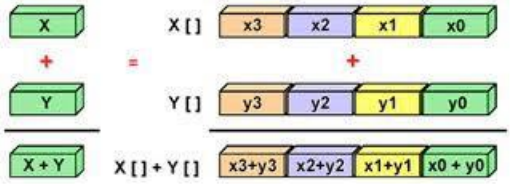
Processori vettoriali

Un processore vettoriale (Array processor) è costituito da più elementi di elaborazione (Processor Element) identici e da una Unità di Controllo.



L'unità di controllo preleva le istruzioni dalla memoria "programma" e distingue tra istruzioni scalari (eseguite direttamente) e istruzioni vettoriali (inviate in parallelo a tutti gli Elementi di Elaborazione dell'array)
Sincronismo della computazione: l'Unità di Controllo 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

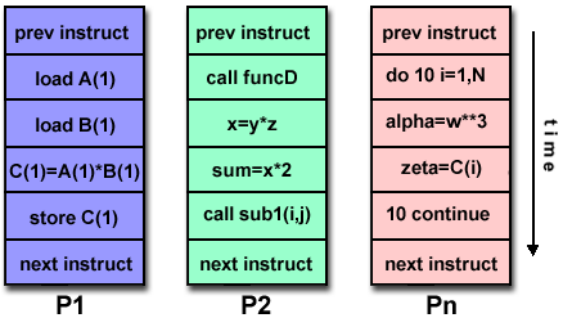
Istruzioni vettoriali
I processori moderni supportano un set di istruzioni vettoriali (o istruzioni SIMD) che si aggiunge al set di istruzioni di istruzioni scalari. Le istruzioni vettoriali specificano una particolare operazione che deve essere eseguita su un determinato insieme di operandi detto vettore.
Le unità funzionali che eseguono istruzioni vettoriali sfruttano il pipelining per eseguire la stessa operazione su tutte le coppie di operandi.



Attualmente istruzioni SIMD sono incluse in quasi tutti i microprocessori, tra cui:

- Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX512
- AMD: 3DNow!

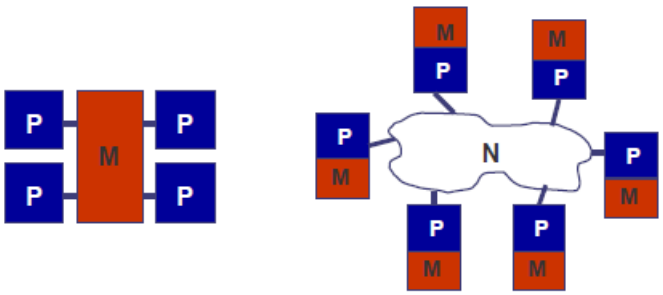
MIMD – Multiple Instruction Multiple Data



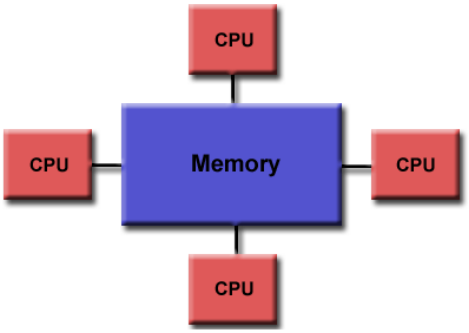
Ogni processore può eseguire un differente flusso di istruzioni. Ogni flusso di istruzioni lavora su un differente flusso di dati. Molte architetture MIMD includono SIMD come caso particolare. I più moderni sistemi di calcolo parallelo ricadono in questa categoria.

Limiti della tassonomia di Flynn

Classificazione non consente di esprimere caratteristiche come la distinzione tra architettura a memoria distribuita e architettura a memoria condivisa.



Sistemi a memoria condivisa UMA



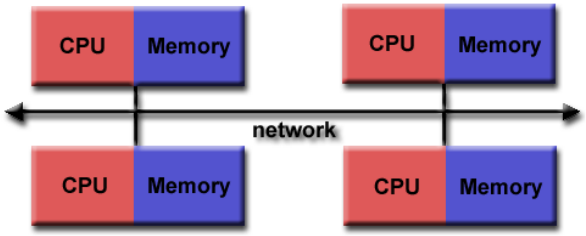
Caratteristica principale: tutti i processori accedono alla memoria come spazio di indirizzamento globale. Modifiche alla memoria da una CPU sono visibili da tutti gli altri. Esistono 2 sottocategorie principali, UMA e NUMA.

UMA – Uniform Memory Access

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 crescente di processori

Questi multiprocessori sono chiamati tightly coupled systems per l'alto grado di condivisione delle risorse.

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

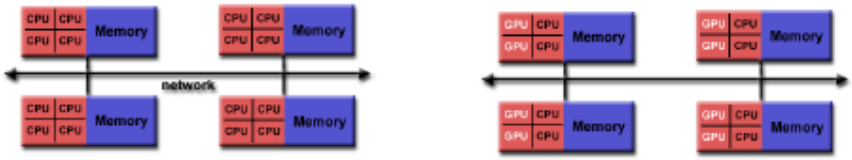
- Il numero di processori e la memoria complessiva scalano con il numero di nodi.
- Costi contenuti (l'hardware può essere commodity).

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, ma generalmente può essere elevato, soprattutto verso i nodi più lontani

Sistemi Ibridi

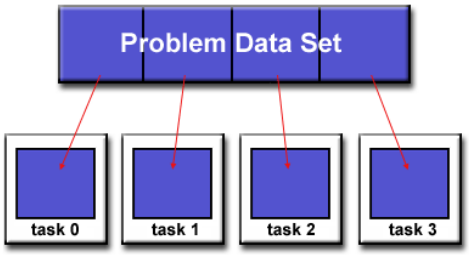
I principali sistemi paralleli oggi sono ibridi, ovvero sono composti da più nodi a memoria condivisa (UMA o NUMA) interconnessi tra loro da una rete ad alta velocità.
Nelle ultime generazioni di sistemi paralleli i nodi a memoria condivisa possono disporre di acceleratori basati su GPU.
Questi acceleratori dispongono di un proprio spazio di memoria e comunicano con il nodo attraversi i bus di I/O (e.g. PCI).



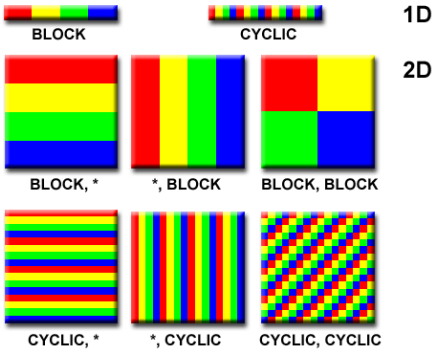
Parallel computing

Tecnica di programmazione necessaria per scomporre il carico computazionale in Task (sottoprogramma che si prende carico di una porzione del problema che elabora in parallelo agli altri) 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

- 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.
- Decomposizione funzionale (task parallelism)
  Distribuzione delle funzioni tra più soggetti: non tutti eseguono le stesse operazioni

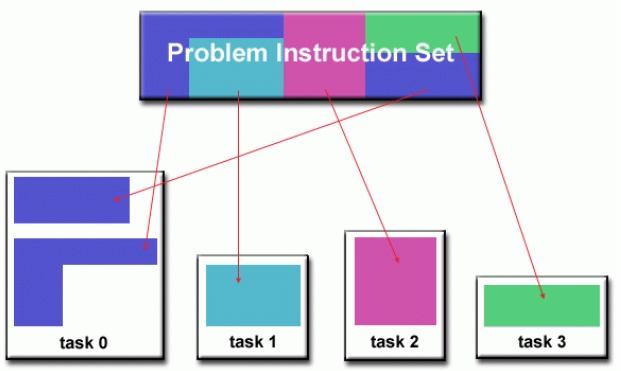


Decomposizione di dominio
Si utilizza quando è necessario elaborare un data set di grandi dimensioni con dati strutturati che vengono suddivisi in sottodomini di dimensioni ridotte.



Ogni sottodominio viene elaborato da un task specifico. Il dominio dei dati ha una propria dimensione (1D, 2D come nella propagazione del calore, 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. Ogni processo prende in carico una particolare elaborazione



Il vantaggio è che possibile scalare con il numero di elaborazioni indipendenti, ma risulta vantaggiosa solo per elaborazioni sufficientemente complesse.

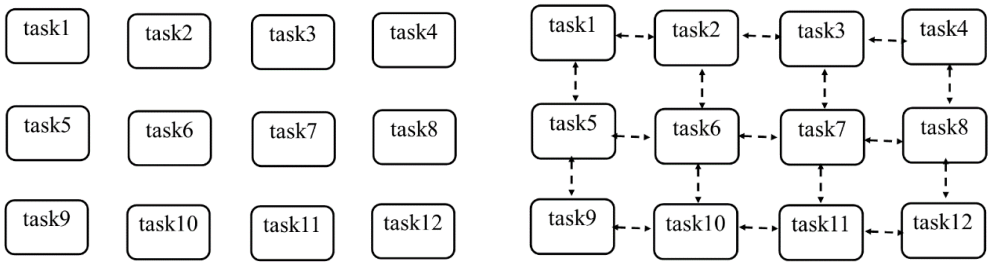
Comunicazione tra i task

Le comunicazioni possono essere

- Punto – punto
- Collettive

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

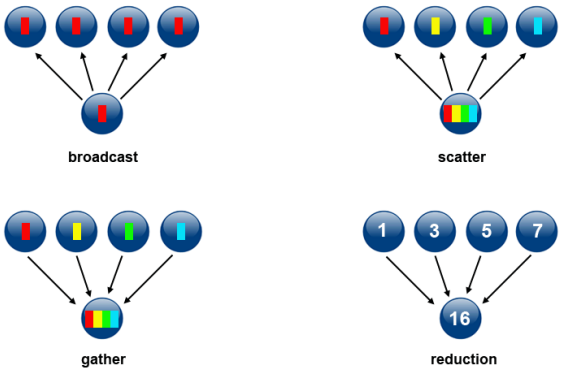
- Lo scambio del colore dei pixel di una immagine tra bianco e nero è parallelizzabile a decomposizione di dominio ma non è richiesta comunicazione tra i task
- La propagazione del calore è parallelizzabile a decomposizione di dominio ma il nuovo valore di temperatura di un sottodominio dipende dalla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



Se non abbiamo comunicazione tra i task si dice che il problema è embarassingly parallel, il risultato finale è l'insieme dei risultati parziali.

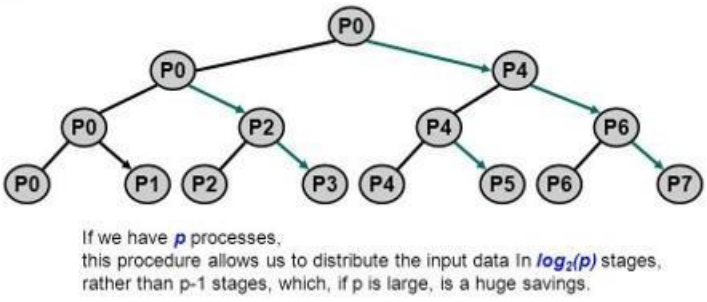
Comunicazioni collettive

Le comunicazioni collettive possono avere diverse varianti:



- Broadcast
  Stessa informazione per tutti gli altri
- Scatter
  Primo quarto al task 1, secondo al task 2 ecc. (ad esempio con gli array), è importante ad esempio con il data parallel per un dominio da elaborare, distribuisce i dati in maniera regolare
- Gather
  Viene dopo lo scatter, il programma raccoglie i vari frammenti
- Reduction
  Operazione di tipo gather, aritmetico-logica

Costo della Comunicazione



Ogni messaggio inviato richiede un tempo che influisce sulle prestazioni.

- Comunicazioni punto – punto
  $$Tmess = Tlat + M/Band$$
  Tlat è la latenza (secondi)
  M numero byte del messaggio
  Band è la Bandwidth (B/s)
- Comunicazioni collettive
  $Tcoll = Tmess \cdot (P-1)$ (P è il numero di Task)

Se utilizziamo la strategia divide-et-impera possiamo ottenere un tempo $O(lg(P))$.

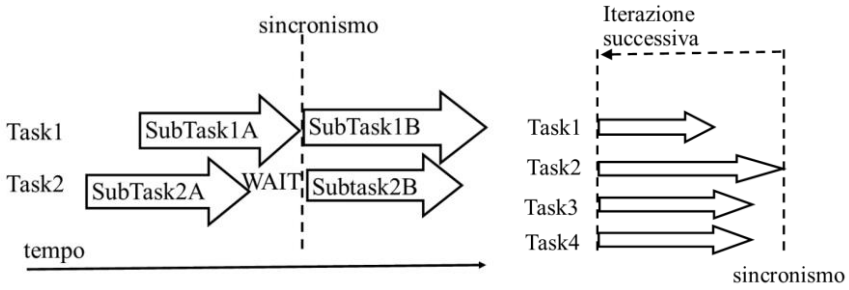
Sincronizzazione

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.

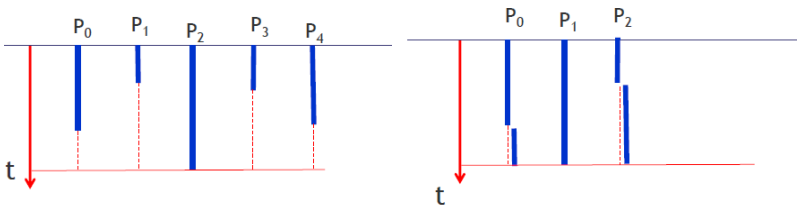


Gli strumenti per la sincronizzazione sono

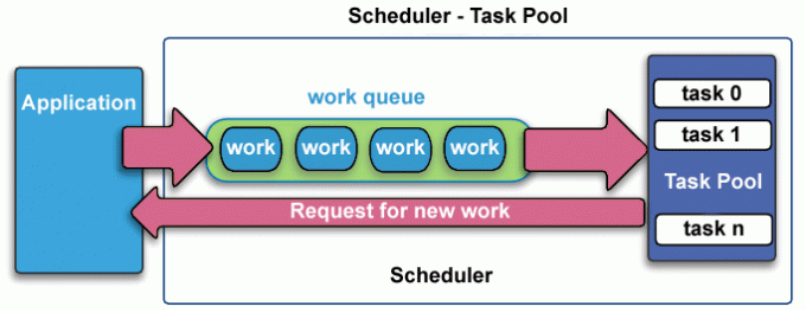
- Le barriere (barrier)
  Implicano il coinvolgimento di tutti i task
- Lock e semafori
  Possono coinvolgere qualsiasi numero di task

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

Bilanciamento del carico

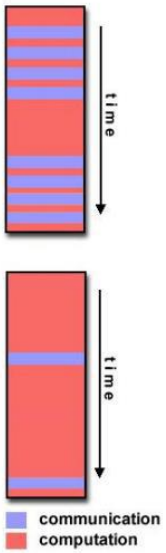


Il bilanciamento del carico (load balancing) è una tecnica di progetto con l'obiettivo di distribuire il lavoro in modo da minimizzare il tempo di Idle dei processi.
Una tecnica comune per bilanciare il carico è il modello master-slave (detto a volte manager-worker), in cui un task (master) ha il compito 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
- Interazione limitata tra master e slave

Parallelismo a grana grossa (coarse grain)

- Migliora il rapporto tra calcolo e le comunicazioni
- Può essere difficile bilanciare il carico
- Probabile aumento delle performance

Performance del parallelismo

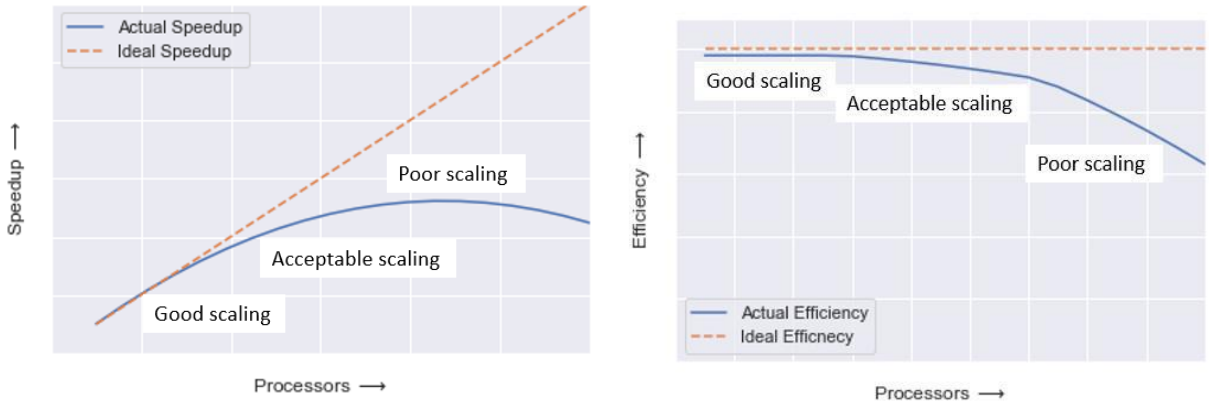
Con il termine Speedup si intende la misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato

$$SpeedUp = \frac{Tseriale}{Tparallelo}$$

che ne caso ideale coincide con il numero di processori.

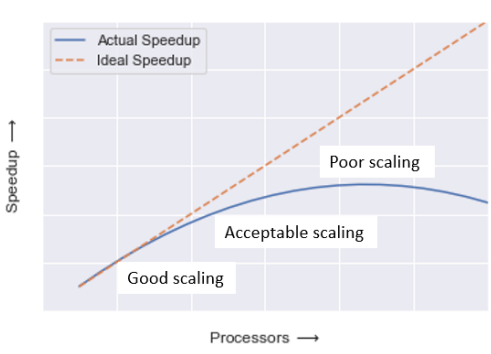
L'efficienza (efficiency) è il rapporto tra Speedup e numero di processori P, nel caso ideale vale 1.

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



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.

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

$$Speedup = \frac{Ts}{\left(\frac{Ts}{P} + Toh\right)}$$

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

Overhead principali

- Tempo speso per le comunicazioni
  $$Tcomm = \Sigma\,(Tmess) = \Sigma\left(Tlat + \frac{M}{Band}\right)$$
  Può essere rilevante per la programmazione a memoria distribuita
- Tempo di avvio e chiusura dei task paralleli
  Può essere rilevante nella programmazione a memoria condivisa
- Tempo di sincronismo
  È meno facile da calcolare a priori

Legge di Amdahl

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

$$Samdahl = \frac{Tseriale}{Tparallelo} = \frac{(Tnp + Tp)}{\left(Tnp + \frac{Tp}{P}\right)} = \frac{1}{Qnp + \frac{Qp}{P}}$$

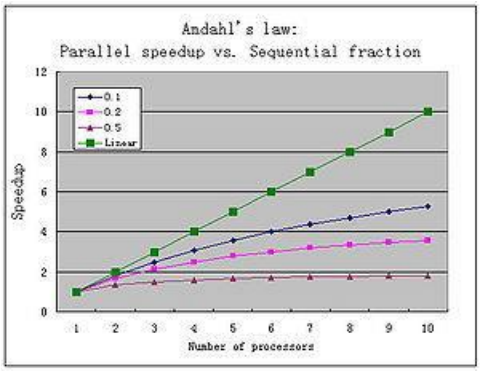
Tp è il tempo parallelizzabile
Tnp è il tempo non parallelizzabile

In termini di quote percentuali:

$$Qnp = \frac{Tnp}{(Tnp + Tp)Qp} = \frac{Tp}{Tnp + Tp}$$
$$dove\ Qnp + Qp = 1$$

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

$$Sreal = \frac{(Tnp + Tp)}{\left(Tnp + \frac{Tp}{P} + Toh\right)}$$



Programma Parallelo

È un programma in grado di suddividere l'algoritmo in diversi task 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)

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

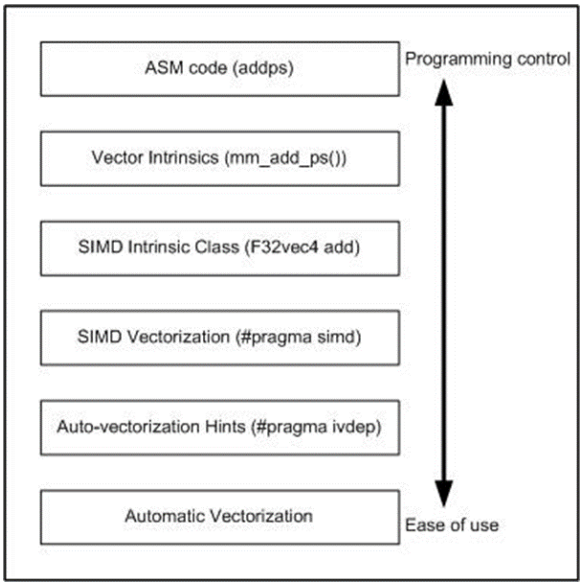
Parallelismo automatico, guidato e manuale

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 al compilatore sulle parti di codice da parallelizzare e come farlo.
  Ad esempio, le direttive di preprocessing pragma che inserite nel codice seriale guidano il parallelismo.
  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 attraverso delle funzioni. (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.



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.

Esempio di parallelismo automatico, guidato o esplicito – Vettorizzazione esplicita con `intrinsics`

Le `intrisics` sono funzioni che si mappano direttamente su una o più istruzioni assembler vettoriali SIMD.

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
```

Esempio di parallelismo automatico, guidato o esplicito – Autovettorizzazione

I compilatori possono individuare e vettorizzare parti di codice.
Ad esempio con gcc

```
#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 -ffastmath -c  prog.c
```

Esempio di parallelismo automatico, guidato o esplicito – Vettorizzazione guidata

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;
}
```

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:

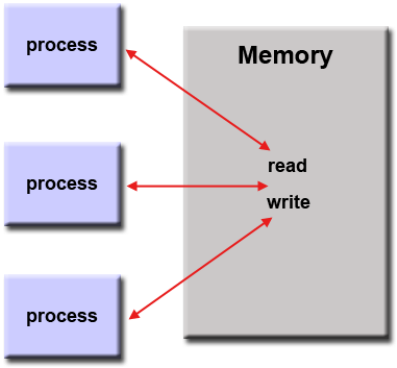
- Paradigma a memoria condivisa o ad ambiente globale (Shared memory)
  I processi interagiscono esclusivamente operando su risorse comuni
- Paradigma a memoria locale o ad ambiente locale (Message passing)
  Non esistono risorse comuni, i processi gestiscono solo informazioni locali e l'unica modalità di interazione è costituita dallo scambio di messaggi

Paradigmi per la programmazione parallela

Memoria condivisa

Nel paradigma shared memory i task 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()` ecc.)
- Sincronizzazione con semafori (`semget()` `semctl()` ecc.)

Thread

Posixthread (Pthreads)

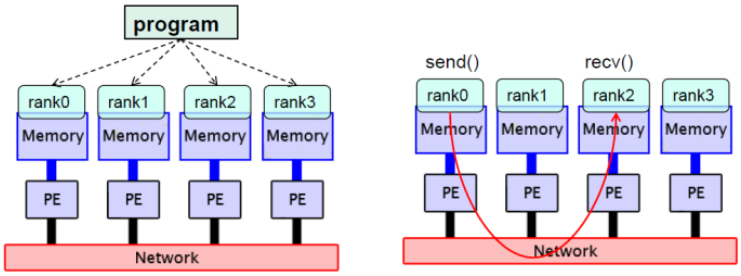
- Comunicazione a memoria condivisa tra più thread
- Sincronizzazione con semafori (`pthread_mutex_*()`)

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

Paradigmi per la programmazione parallela

Memoria distribuita



Nel paradigma message passing i processi comunicano scambiandosi messaggi

Primitive message passing di base

```
send (parameter list)
receive (parameter list)
```

MPI – Message Passing Interface

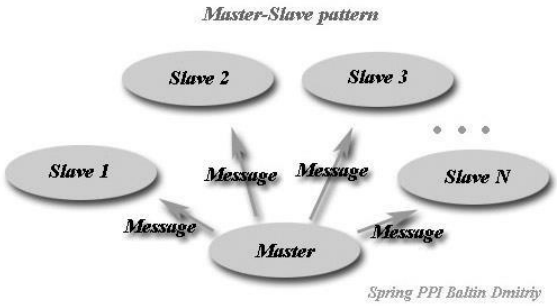
La libreria MPI è uno standard per il message passing.

SPMD – Single Program Multiple Data

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.
Il modello SPMD è largamente usato anche nella programmazione MPI, `mpirun` è il comando MPI che mette in esecuzione n istanze dello stesso programma.

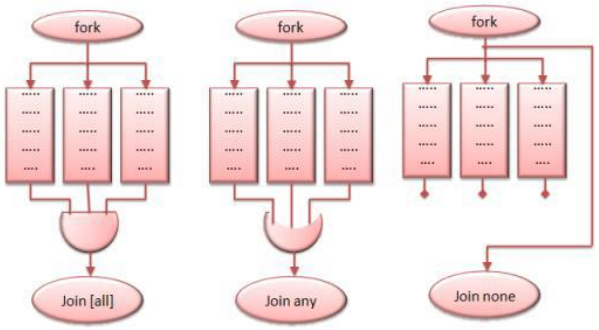
Master – Slave



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.

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

Fork/Join



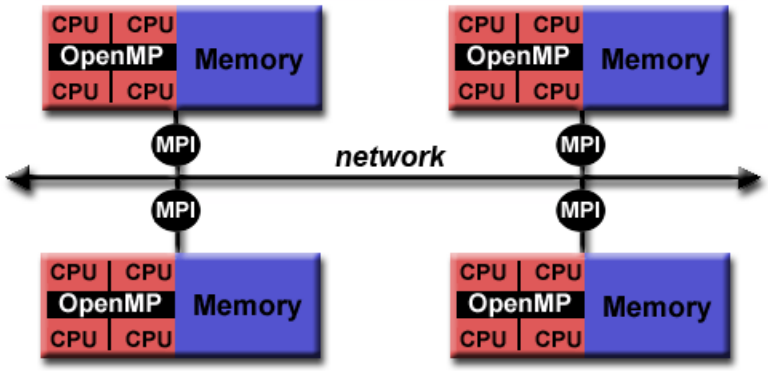
La fork eseguita dal task master genera dinamicamente uno o più nuovi task che eseguono un nuovo flusso di controllo parallelamente al master.
La Join viene eseguita da tutti (o parte) dei task concorrenti.

- 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.
OpenMP si basa sul modello Join All, l'utilizzo elevato di strutture Fork/Join può introdurre overhead dovuti a start e stop dei thread.

Paradigmi per la programmazione parallela – Sistemi ibridi

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



Laboratorio – Performance

Il programma `normal` mette in linea le operazioni `normal` di numeri double.
L'obiettivo è osservare fino a che punto la pipe è performante al crescere del numero di operazione floating point in un ciclo.
Abbiamo un duplice aspetto, capire meglio come funziona il processore per sfruttare le sue caratteristiche di performance e capire come interagire con il compilatore per sfruttare in modo automatico le caratteristiche hardware.

Una operazione Normal è una operazione di somma e moltiplicazione di numeri reali

$$(a \cdot b + c)$$

molto comune nei programmi che trattano array (o tensori) e che nei processori viene implementata in un'unica istruzione FMA (Fused Multilpy-Add).
Le operazioni Floating Point richiedono diversi cicli di clock ma possono generare un risultato per ciclo di clock grazie all'architettura pipeline.

Il programma `normal.c` itera `MAXCICLE` volte un ciclo di N operazioni NORMAL, dove N varia da 1 a 15, attraverso le istruzioni di `if`.
Questi costrutti vengono ripetuti n volte per avere un tempo di calcolo apprezzabile dal punto di vista della direttiva `gettime`.
Ogni operazione NORMAL consiste di 2 operazioni Float.

$$2 \cdot N \cdot \frac{MAXCICLE}{tempo_{totale}}$$

Il programma ripete il calcolo dei MFlops al crescere di N e stampa N, il tempo di calcolo, il numero di operazioni eseguire e i MFLOPS calcolati.

Il programma `normal.slurm` chiede il processore sulla partizione cpu o cpu guest, si legge l'informazione dei MHz sulla CPU per una pipe che cresce fino a 15.

Il comando `hpc-sinfo` fornisce il numero di nodi, stato (mixed è in parte utilizzato e in parte no), CPU, socket, core (per processore), memoria, features.

Con l'opzione `partition` si chiede la CPU, con l'opzione `costraint` si può specificare il tipo di CPU da assegnare, riducendo i nodi assegnabili e richiedendo quindi più tempo di esecuzione.
Si noti che chiedendo tanta memoria il nodo che sta ospitando altri programmi che usano a loro volta una parte della memoria disponibile, potrei non trovare quella che mi serve sul nodo.
Lo script `normal-test.slurm` presenta test in base a diverse opzioni di compilazione.

Disktest
Si ha uno script slurm che lancia uno script di shell, il quale usa il comando `dd` per scrivere e leggere un file su disco, con input file, output file e block size definite.
`/dev/zero` sono sequenze di 0 scritte su un file misurandone la velocità.
Si hanno diverse tipologie di dischi, presumendo che le diverse caratteristiche (come l'uso massiccio di disco) comportano diverse prestazioni (in questo caso spostandosi sulla memoria scratch si potrebbero migliorare).
In generale ci si sposta in diverse partizioni per poi eseguire lo script. Non è importante il tipo di processore, ma il disco.

La distribuzione CUDA include programmi per testare la GPU.
Si può utilizzare attraverso `module load cuda`.

- `bandwidthtest`
  Chiede un nodo con partizione GPU.
- `gres`
- Equivalente della `costraint`, ovvero si seleziona una particolare GPU (in questo caso p100). Si copia il contenuto della directory, e dopo un make si lancia il programma. In questo caso potrebbe essere più difficile trovare una risorsa disponibile.
- nbody
  Identifica quanti nodi mettiamo in gioco, quante GPU utilizziamo, -benchmark indica la modalità, che determina quante sono le operazioni fp coinvolte e determina il numero di megaflops ottenuti.
- `bandlat`
  Risulta interessante in quanto coinvolge più nodi, attraverso due programmi che utilizzano primitive di comunicazione tra processi con lo scopo di verificare banda e latenza della comunicazione.

Laboratorio – Parallelizzazione di programmi seriali

I seguenti programmi seriali verranno analizzati per valutare le possibili performance nella parallelizzazione, tenendo conto della legge di Amdahl e degli overhead.

CPI

Il programma calcola pigreco utilizzando due diversi metodi di integrazione numerica, per il calcolo dell'integrale viene utilizzato il metodo dei rettangoli nell'intervallo tra 0 e 1.

Nel programma `cpi2` sono presenti due funzioni che implementano i metodi di integrazione, la funzione `getopt` per la gestione dell'input dei parametri (selezione della funzione e del numero di intervalli) e `clock_gettime` per determinare i tempi di calcolo.

Considerazioni sulla parallelizzazione di cpi.c

Un ciclo `for` ha il compito di calcolare tutte le aree degli intervalli tra 0 e 1.
Le iterazioni sono indipendenti tra loro e possono essere distribuite tra i diversi thread o processi ed eseguite contemporaneamente.
La decomposizione può avvenire a grana grossa (dividiamo gli intervalli in P parti, dove P è il numero di processori) oppure a grana fine se lavoriamo a memoria condivisa con thread.
Tutto il resto del programma non è parallelizzabile.

Occorre valutare i tempi Tp e Tnp misurati con dei timer per applicare la legge di Amdahl.
La quota non parallelizzabile è indipendente da n e viceversa.

Comunicazione a memoria distribuita

Ogni unità di processamento accumula nella variabile locale `sum` i contributi del ciclo for che ha gestito.
Al termine delle iterazioni tutti inviano la propria somma parziale (un double) al processo principale (task master) che somma tutti i contributi (operazione di riduzione della libreria `mpi`).
Il processo principale riceve $p - 1$ messaggi di 6 byte (un double), $O(P)$ oppure $O(lg2(P))$ se usiamo divide-et-impera.

Con partizionamento a grana grossa ad ogni processo è affidato un piccolo task, questo è ottimale per il bilanciamento del carico perché tutti i thread saranno occupati al 100%, è un partizionamento più dinamico adatto alla memoria condivisa in cui lo scheduler a suddividere i compiti è un ciclo gestito da OpenMP.

Startup

Prima di iniziare la regione parallela è necessario creare P thread e eliminarli al termine, $O(P)$.

Sincronismo
Prima di comunicare i risultati e terminare i thread occorre attivare una barriera (implicita o esplicita), attendendo che tutti abbiamo completato il loro lavoro.
Se usiamo decomposizione a grana fine non abbiamo tempi di Idle.
A grana grossa occorre fare attenzione che i sottodomini siano di pari dimensione. La fase di comunicazione è assente nel caso di memoria condivisa, mentre è necessaria per la memoria distribuita.

Esercizio heat

Il programma `heat.c` simula la propagazione del calore su una superficie rettangolare bidimensionale con alcune parti mantenute a temperatura costante che permettono alle altre zone di ricevere calore per propagazione.
Lo spazio viene discretizzato in 2 dimensioni attraverso una griglia e il tempo attraverso degli step temporali. Lanciando il programma si ottengono i valori della temperatura in ogni punto.

Le funzioni `Init_center()` `Init_left()` e `Init_top()` mantengono la temperatura nelle corrispondenti parti del rettangolo (parte centrale, bordo in alto e bordo a sinistra).

Le funzioni `copy_rows()` e `copy_cols()` realizzano le condizioni periodiche al contorno nelle due direzioni, ovvero mettono a contatto termico i bordi alto/basso e destra/sinistra, a seconda della configurazione scelta.

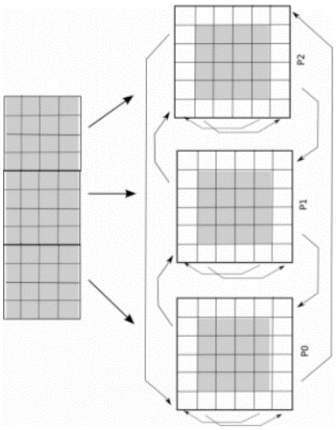
La funzione `jacobi()` determina la nuova temperatura allo step `t1` come media della temperatura nei punti vicini allo step `t0` (in questo caso solo i 4 vicini, nord, sud, est e ovest).
Si hanno due griglie, una che indica la temperatura nei diversi punti in tempi diversi (new e old nel codice). Una volta eseguito il calcolo per la modifica della temperatura new e old vengono invertiti. Tutto ciò perché non è possibile lavorare su una sola griglia, in quanto la temperatura nella griglia old potrebbe influire sul calcolo della new. La nuova griglia viene scritta attraverso due cicli for.

Procedimento
In modo indipendente aggiorno le celle della superfice nella nuova superficie, calcolo la copia delle righe e delle colonne (dopo l'operazione di `jacobi`, per poi fare il rinfresco delle temperature costanti) avendo così delle righe di bordo aggiuntive che servono a contenere una copia necessaria per le condizioni periodiche di contorno.

Ad ogni iterazione servono tre puntatori, per effettuare lo swap delle due griglie, dopo il quale si rinfresca la temperatura nelle zone costanti. Il tempo discretizzato si identifica nel for che itera fino a max iter.
Il programma scrive su stdout i valori della temperatura finale nei punti della superficie e su stderr i parametri e tempi di esecuzione. Possiamo salvare la mappa del calore in un file `heatmap.out`.
Il programma heatmap_plot.py esegue il programma e determina heatmap.png utilizzando la funzione `imsave` di python possiamo generare una colormap in formato png partendo dall'output del programma, in questo caso un file .dat, assegnando colori a diversi valori (0 nero, 1 bianco, gradazioni intermedie colori che vanno dal nero al bianco).
Il programma `heatmap_animation.py` determina diverse heatmap al crescere del numero di iterazioni che poi utilizza per generare una animazione in formato gif. Le singole immagini png vengono incluse nel file gif in ordine alfabetico.

Considerazioni sulla parallelizzazione di heat.c
Decomposizione dominio
La funzione `Jacobi`, che ad ogni iterazione aggiorna la temperatura di tutti i punti della superficie, può essere parallelizzata partizionando la superficie tra i processi/thread. Anche in questo caso possiamo lavorare a grana grossa, ad esempio suddividendo la superficie in P strisce prestando attenzione ai tempi di idle, oppure a grana fine se lavoriamo con i thread.
Ovviamente il ciclo for esterno `for(iter=0; iter<MAX_ITER; iter=iter+1)` non può essere parallelizzato poiché rappresentata la progressione temporale.
In prima approssimazione consideriamo tutto il resto non parallelizzabile (principalmente l'aggiornamento della temperatura nei punti caldi).
Occorre analizzare i tempi Tp e Tnp per la legge di Amdahl.

Overhead – Comunicazione



Se la memoria è distribuita (coarse grain) partizioniamo la superficie, ad esempio in P righe (1D) e distribuiamo un sottodominio ad ogni processo.
Ad ogni iterazione ogni processo aggiorna il proprio sottodominio. Al termine delle iterazioni i sottodomini aggiornati devono essere ricomposti inviandoli al processo principale.

I sottodomini devono essere inizialmente distribuiti dal processo principale verso i $P-1$ altri processi (scatter), $O(P)$.
Al termine occorre fare l'operazione opposta (gather), $O(P)$.
Inoltre, ad ogni iterazione le righe di bordo devono essere scambiate tra processi adiacenti, per garantire al task di eseguire il calcolo corretto delle celle sul bordo.

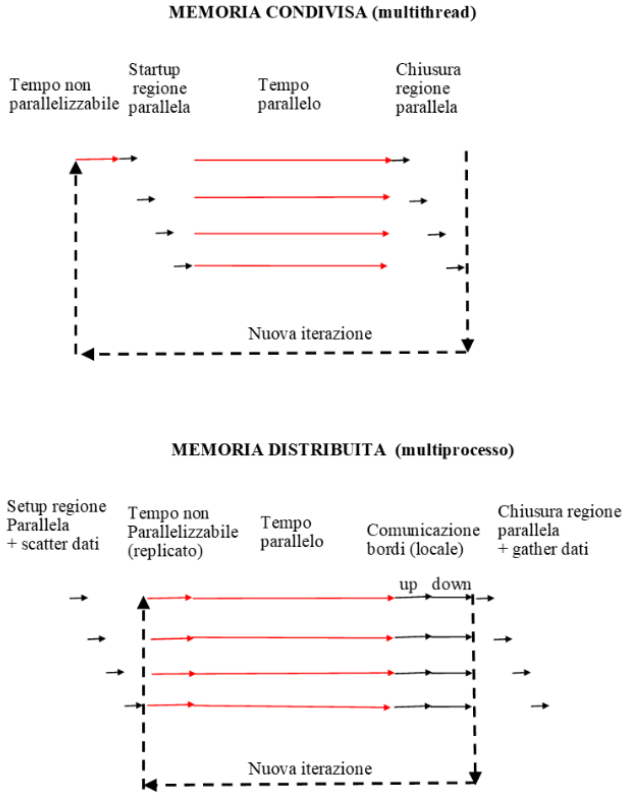
Se abbiamo `I` iterazioni e ogni riga contiene N punti, ogni processo scambia quattro messaggi di N float ad ogni iterazione.

$$Tcomm = \ 4 \cdot I \cdot Tlat \cdot \left(\frac{N}{Band}\right)$$

Se utilizziamo il modello a memoria condivisa i thread aggiornano parti diverse della stessa superficie. In questo caso possiamo operare "fine grain" ovvero distribuiamo i cicli for della funzione `Jacobi` tra i thread disponibili.

Startup
Nel modello a memoria distribuita vengono creati P processi indipendenti all'inizio del programma e cancellati al termine $O(P)$.
Nel modello a memoria condivisa i thread vengono creati ed eliminati ad ogni iterazione $O(P * I)$

Sincronismo
Barriera (implicita o esplicita) al termine di ogni iterazione prima di inviare i messaggi.
Se il dominio è decomposto in sottodomini della stessa dimensione non abbiamo tempi di Idle. Il seguente diagramma riporta la profilazione temporale nell'esecuzione parallela. Per semplicità è riportato lo scambio di un solo messaggio.



Esercizio Moltiplicazione Matrici

Il programma esegue la moltiplicazione di due matrici quadrate di lato n e stampa i tempi di calcolo Tp e Tnp, considerando parallelizzabile il calcolo della matrice C.
Si compone di cicli `for` innestati che eseguono somma e moltiplicazione di righe e colonne. Si tratta di tipiche operazioni `normal` messe in sequenza.
I cicli `for` sono indipendenti, è infatti possibile calcolare i punti della matrice risultante in qualsiasi ordine.
Le iterazioni dei cicli vengono dati in gestione a OpenMP che le distribuisce su n thread a disposizione.

Esercizio numeri primi

I programmi `primi.c` e `crivello.c` calcolano i numeri primi fino a N per forza bruta (primi.c) e con il crivello di Eratostene (crivello.cpp).

OpenMP

OpenMP è una libreria che consente di gestire il multithreading finalizzato a una programmazione parallela.
La gestione avviene attraverso tre componenti

- 44 direttive per il compilatore (pragma) che indicano come avviene il parallelismo all'interno del codice seriale
- 35 routine runtime (API) che consentono di interagire con l'ambiente per ottenere informazioni riguardo ai thread (come l'identificativo, ad esempio omp_get_wtime che restituisce un double e misura i tempi di esecuzione)
- 13 variabili d'ambiente con cui interagisce la libreria per monitorare lo stato

Ci sono diverse versioni di openMP in quanto varia frequentemente l'hardware da supportare, per cui è importante usare quella corretta.
Le direttive sono nella forma `#pragma omp` seguito dalla direttiva stessa (es. parallel genera la regione parallela).
Ci sono poi clausole a seguire che descrivono il comportamento della direttiva (ad esempio, `default(shared)` che indica la zona condivisa delle variabili).
Le librerie a run-time sono delle subroutine, nel caso del C++ è necessario includere l'header file `<omp.h>`.

OpenMP Directives: C/C++ Directive Format

Format:

#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.

Example:

```
#pragma omp parallel default(shared) private(beta,pi)
```

Modello di programmazione

OpenMP è stato realizzato per multiprocessori a memoria condivisa. Le architetture sottostanti possono essere a memoria condivisa UMA o NUMA. Nel nostro caso si ha una architettura NUMA, con la possibilità di controllare dove il thread viene collocato rispetto all'hardware, con una memoria più veloce e accessibile.

Modello Fork – Join
OpenMP utilizza il modello fork – join di dinamica per l'esecuzione parallela

- Si parte con un singolo processo, il master thread, che esegue in modo sequenziale fino a quando non incontra la prima regione parallela
- Fork
  l master thread crea quindi un gruppo di thread paralleli
  Le istruzioni nel programma racchiuse nella regione parallela sono eseguite in parallelo attraverso i vari thread del gruppo
- Join
  Quando il gruppo di thread completa le istruzioni nella regione parallela, si sincronizzano e terminano, lasciando attivo solo il master thread

Il numero di regioni parallele e di thread che eseguono le istruzioni in esse contenute è arbitrario.

Thread dinamici
È possibile modificare il numero di thread dinamicamente, durante l'esecuzione.

I/O
OpenMP non fornisce specifiche sull'input/output parallelo.
Questo aspetto è particolarmente importante se thread multipli tentano di leggere o scrivere lo stesso file. Se ogni thread conduce I/O a un file diverso, non ci sono particolari problemi.
È a discrezione del programmatore assicurarsi che l'I/O sia costruito correttamente nel contesto di un programma multi-thread.

Programma openMP
Elementi principali

- Direttiva di inclusione della libreria
- Variabili globali varie
- Codice seriale
- Direttiva pragma che indica l'inizio di una regione parallela (racchiusa tra parentesi graffe)
- Sezione parallela eseguita da tutti i thread
- Altre direttive OpenMP
- Chiamate a librerie run-time
- Sincronizzazione e termine dei thread (tranne il master)

Clausole delle direttive pragma

- Private/shared
  Definiscono le variabili
- Reduction (operator: List)
  I risultati locali di ogni thread o processo vengono raccolti dal master thread che effettua operazioni aritmetico-logiche e di comunicazione
- Num_threads (integer expression)
  Ssovrascrive variabile ambientale e indica quanti thread lavoreranno

La maggior parte dei costrutti OpenMP agiscono sui cicli for:

C/C++:

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

Questo tipo di direttive agisce quando la regione parallela è già attivata, e vanno inserite prima dei cicli for (con le varie clausole possibili).

C/C++ - for Directive Example

```
#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 */

}
```

Senza la direttiva ogni thread eseguirebbe il suo ciclo for, mentre in questo modo le iterazioni sono distribuite sui diversi thread disponibili da OpenMP.
È compito del programmatore garantire l'indipendenza delle iterazioni.

Direttiva sections
Distribuisce sui diversi thread diverse sezioni di codice esplicitabili nelle opzioni della direttiva.

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```

Laboratorio – OpenMP base

Uno degli overhead possibili nell'introdurre parallelismo consiste nel fatto che bisogna attivare i thread per iniziare e poi chiuderli alla fine.
In multithreading la cosa diventa più dinamica, quindi si necessita di più risorse.

Il tempo di esecuzione è

$$T = Sleep \cdot nIterazioni + Toverhead$$

File `omp_overhead.c`

```
omp_set_num_threads(NT);

t1=omp_get_wtime();

for(i=0; i< ITER; i++){ //itera
  #pragma omp parallel for
  for (j=0; j< NT; j++) //crea un thread per ogni iterazione
    usleep(SLEEP);
}

t2=omp_get_wtime();
tr=t2-t1;
```

Ci si aspetta che l'overhead sia variabile in base al numero di thread, lunghezza delle sleep ecc.

```
./omp_overhead 28 1000 1000 >> omp_overhead_mod.csv
```

Con questo comando di esecuzione si richiede un nodo con 28 core, 1000 iterazione e 1 secondo di sleep.

La relazione che c'è tra overhead dovuto all'apertura e chiusura dei thread (o overhead nelle comunicazioni, con MPI) e la grandezza del problema è che più è piccolo il problema più il tempo di overhead diventa importante, andando a diminuire lo speedup.

Anche al crescere del numero di thread il tempo di overhead rimane più o meno costante, non sono direttamente proporzionali.

File omp_rand.c
Andando ad aggiungere una direttiva `#pragma` in testa al `for`, come nell'esempio, succede che si hanno tanti thread che in parallelo iterano.

```
#pragma omp parallel
for(i=0; i<n; i++){
    x = drand48();
    printf ("tid:%d/%d i:%d x:%.4f \n", omp_get_thread_num(),
                                omp_get_num_threads(), i, x);
    usleep(x*100000);
    sum += x;
  }
```

Ci possono essere dei tempi di idle che i thread devono aspettare, non sono efficienti.
Successivamente si trova una riduzione, ad ogni iterazione viene effettuato l'accumulo di una variabile.
Quando si parallelizza bisogna stare attenti, se tutti i thread eseguono lo stesso codice tutti cercheranno di incrementare la stessa variabile, andando a generare una race condition.

```
#pragma omp parallel for schedule(static,1)
```

Inserendo una direttiva come questa si ha che ogni thread esegue un chunk, sempre con lo stesso ordine, la rotazione è quindi statica

```
#pragma omp parallel for schedule(dynamic,1)
```

Al contrario, in questo caso la rotazione è dinamica.
In questo caso questa soluzione è migliore, non vincola ogni thread a dover eseguire un chunk determinato a priori, dato che, ad esempio, il primo thread che esegue potrebbe essere l'ultimo a terminare, andando quindi a bloccare gli altri.
Utilizzato uno scheduler dinamico si ottiene un bilanciamento del carico migliore.

Altro discorso è la gestione della variabile `sum`, che, dato che può essere acceduta in lettura e scrittura da tutti i thread, deve essere racchiusa in una zona critica

Si può fare così

```
#pragma omp critical
sum += x;
```

ma ci sono delle alternative migliori, come ad esempio una riduzione (tutti i thread hanno la propria variabile da incrementare e solo alla fine vengono sommate)

```
#pragma omp parallel for schedule(dynamic,1) reduction(+:sum)
```

Tutti i conti vengono ora effettuati su copie della variabile, una per ogni thread.

File omp_mm.cpp

```
#pragma omp parallel {
   //#pragma omp for schedule (static, chunk)
   //#pragma omp for schedule (dynamic, 100)
   #pragma omp for
   for(i=0;i<NRA;i++)
      for(j=0;j<NCB;j++)
         for(k=0;k<NCA;k++)
            *(c+i*NCB+j)+= *(a+i*NCB+k) * *(b+k+j*NRA);
}
```

In questo caso non ci sono problemi di bilanciamento del carico e di variabile condivise.

Nota

```
#pragma omp parallel for schedule(static,1)
```

Nello specificare lo scheduler, il primo parametro è il tipo di scheduler, il secondo è la dimensione che si vuole assegnare ad un chunk.

Laboratorio – OpenMP heat

Per parallelizzare il programma `heat.c` è possibile modificare il ciclo for della funzione

```
void Jacobi_Iterator_CPU(float * __restrict T, float * __restrict T_new,
                                               const int NX, const int NY);
```

aggiungendo anche il tempo di calcolo e la stampa dei tempi e del numero di thread.

```
#include <omp.h>
//...
t2=omp_get_wtime();
    for(iter=0; iter<MAX_ITER; iter=iter+1)  { //... }
t3=omp_get_wtime();
```

Per stampare il numero di thread della regione parallela

```
#pragma omp parallel
#pragma omp single
fprintf(stderr,"%f %d \n", t3-t2, omp_get_num_threads());
```

Prima del ciclo for della funzione di Jacobi si inserisce la direttiva per la parallelizzazione

```
#pragma omp parallel for private (i)
    for(j=1; j<NY-1; j++)
        for(i=1; i<NX-1; i++) { }
```

La variabile `i` è stata definita al di fuori della regione condivisa, quindi è privata.
Ogni thread prende un insieme di righe e itera sulle colonne, quindi ogni thread ha la propria variabile `i` sulla quale iterare.

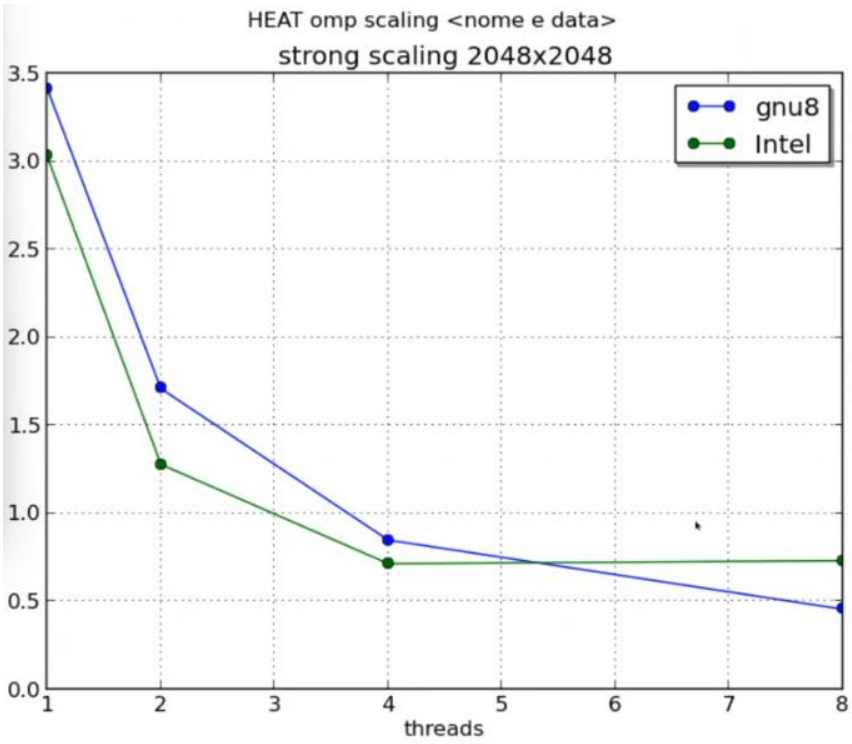
Per l'esecuzione si crea un file slurm con all'interno

```
module load intel
icc -O2  omp_heat.c -fopenmp -o omp_heat

for T  in   1 2 4 8 16
do
    OMP_NUM_THREADS=$T ./omp_heat -c 2048 -r 2048
done
1> /dev/null      2> omp_heat.dat
```

Si fa variare $T per cambiare il numero di thread paralleli ad ogni iterazione.

Allo stesso modo, si può effettuare la compilazione con il compilatore normale, anche per confrontare le prestazioni.

```
module purge
module load gnu8
gcc -O2  omp_heat.c -fopenmp -o omp_heat

for T  in   1 2 4 8 16 do
    OMP_NUM_THREADS=$T ./omp_heat -c 2048 -r 2048
done    1> /dev/null      2> omp_heat.dat
```



Laboratorio – `mpirun` e `mpicc`

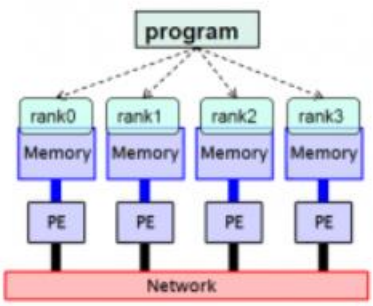
MPI (Message Passing Interface) è una libreria per lo scambio di messaggi in un ambiente di calcolo distribuito su più processi.
È uno standard, le implementazioni sono, ad esempio, OpenMPI e Intel – MPI.
Si tratta di estensioni del compilatore di base per il calcolo distribuito, si ha sempre bisogno del compilatore di base.

Per caricare i moduli necessari sul cluster si può eseguire il comando

```
module purge
module load gnu8 openmpi3
module purge
module load intel impi
```

Una volta caricati i moduli si hanno a disposizione tutti i comandi per la compilazione e l'esecuzione di programmi MPI e tutte le librerie necessarie.

Comando `mpirun`
Serve a mettere in esecuzione i programmi che deve essere messo in esecuzione su più istanze con il modello SPMD (Single Program Multiple Data).
Si ha un programma unico da mettere in esecuzione su diversi processi in modo che ogni processo abbia a disposizione la propria memoria.
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.
La gestione dei processi sui diversi nodi avviene tramite demoni (Orted in openMPI) che comunicano tra loro via SSH (o altri protocolli di rete).

Opzioni principali di `mpirun`

- `-np` `-n` (OpenMPI e Intel)
  Numero di processi da attivare
- `-ppn` (Intel) `-npernode` (OpenMPI)
  Numero di 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

I vari rank possono comunicare tra di loro per scambiare i dati, utilizzando la rete veloce e quindi protocolli più di basso livello.

Ogni nodo ha un numero fissato di slot (che può anche corrispondere al numero di processori) per l'esecuzione, e, quando vengono specificati gli host tramite `mpirun`, questo si occuperà di scegliere su quale host creare i processi (in base alla disponibilità) oppure come distribuirli.

```
module load intel impi
mpirun -np 6 hostname       # esegue 6 istanze di hostname all'interno di 6
                            # processi eseguiti sullo stesso host da cui è stato
                            # eseguito il comando
mpirun --host wn53,wn54 hostname
mpirun --host wn53,wn54 -np 2 hostname   # esegue 2 istanze di hostname
mpirun --host wn53,wn54 -npernode 1 -np 2 hostname
```

Con il comando

```
mpirun -np 4 hostname : -np 2 lscpu
```

si richiedono 4 istanze di hostname e 2 istanze di lscpu, in questo caso l'host è sempre la macchina di login.

```
mpirun -np 4 hostname : -np 1 -host wn51 lscpu
```

In questo modo si va a specificare anche l'host che si intende utilizzare.

Con Intel si può eseguire un comando tipo

```
mpirun --machine machine.txt --ppn 1 hostname
```

dove `machine.txt` specifica il nome della macchina e il numero di nodi

```
wn51:1
wn52:1
wn53:2
wn54:2
```

In modo equivalente, con OpenMPI

```
$(which mpirun)  --hostfile hostfile.txt  --npernode 1 hostname
```

dove hostfile.txt è

```
wn51  slots=4
wn52  slots=4
wn53
wn54
```

Il file rankfile.txt consente di definire i rank, si può dire ad esempio che il rank 0 giri sul nodo wn01 con uno slot che varia tra 16 e 31.

```
rank 0=wn01 slot=16-31
rank 1=wn01 slot=16-31
rank 2=wn01 slot=16-31
rank 3=wn01 slot=16-31
rank 4=wn02 slot=0-15
rank 5=wn02 slot=0-15
rank 6=wn02 slot=0-15
rank 7=wn02 slot=0-15
```

Tramite SSH è possibile eseguire un comando su un nodo

```
ssh wn51 numactl -H #esegue il comando numactl su wn51
```

Se ci si trova in un ambiente di code come Slurm, che non ha bisogno di una gestione esterna per la scelta dei nodi e delle impostazioni, si utilizza l'integrazione tra MPI e Slurm, quindi il controllo della gestione delle risorse viene delegato a Slurm.
Si selezionano le risorse con Slurm e queste vengono comunicate a `mpirun` tramite dei wrapper.

Così facendo, si specificano le risorse in testa e si lancia `mpirun` senza parametri, dato che sarà `mpirun` stesso ad andarle a recuperare.

```
#!/bin/bash
#SBATCH --output=%x.o%j  # Nome del file per lo standard output
#SBATCH --partition=cpu              # Nome della partizione
#SBATCH --nodes=2                    # numero di nodi richiesti
#SBATCH --ntasks-per-node=2          # numero di cpu per nodo
#SBATCH --time=0-00:10:00            # massimo tempo di calcolo
module purge
module load  gnu8 openmpi4

echo  "#SLURM_JOB_NODELIST      : $SLURM_JOB_NODELIST"
echo  "#SLURM_JOB_CPUS_PER_NODE : $SLURM_JOB_CPUS_PER_NODE"

mpirun hostname
```

Per la compilazione non basta più il compilatore di default `gcc`, bisogna utilizzare `mpicc`, che è un MPI Wrapper.

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

File `mpi_latency.c`
Il programma viene eseguito con diverse istanze e, tramite MPI, le varie istanze comunicano tra loro.

Bisogna includere la libreria

```
#include "mpi.h"
   //...
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank == 0 && numtasks != 2) {
      printf("Number of tasks = %d\n",numtasks);
      printf("Only need 2 tasks - extra will be ignored...\n");
   }
   MPI_Barrier(MPI_COMM_WORLD); //si va aventi solo se tutti sono arrivati alla
                                //barriera
```

Invio di un messaggio

```
      dest = 1;
      source = 1;
      for (n = 1; n <= reps; n++) {
         T1 = MPI_Wtime();     /* start time */
         /* send message to worker - message tag set to 1.  */
         /* If return code indicates error quit */
         rc = MPI_Send(&msg, 1, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
         /* Now wait to receive the echo reply from the worker  */
         /* If return code indicates error quit */
         rc = MPI_Recv(&msg, 1, MPI_BYTE, source, tag, MPI_COMM_WORLD,
                       &status);
         T2 = MPI_Wtime();
      }
      //...
      MPI_Finalize();
```

OpenMPI

È una delle implementazioni MPI.

Compilazione

```
mpicc –o myprog myprog.c
```

Esecuzione

```
mpirun [-np <num>] [-hostfile <hostfile>] <execfile>
```

Dove

- `<num>` è il numero dei processi
- `<hostfile>` è il nome del file con la lista di tutti gli IP delle macchine
- `<execfile>` è il nome del programma da eseguire

Hostfile

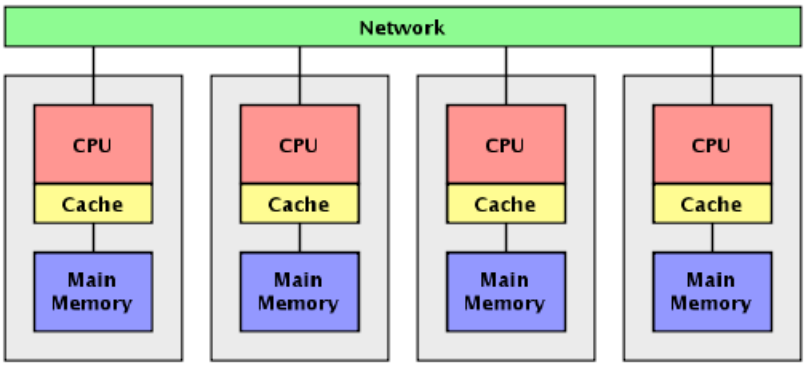
In questo file sono specificati tutti gli host, uno per riga.

Per ogni host si possono specificare il numero massimo di processori disponibili.

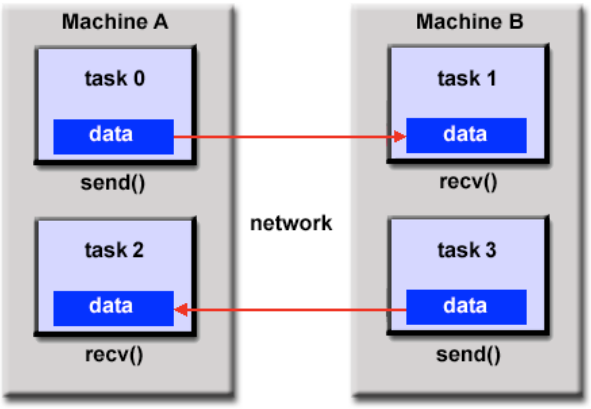
```
# This is an example hostfile.  Comments begin with #
#
# The following node is a single processor machine:
foo.example.com

# The following node is a dual-processor machine:
bar.example.com slots=2

# The following node is a quad-processor machine, and we absolutely
# want to disallow over-subscribing it:
yow.example.com slots=4 max-slots=4
```

Message Passing con MPI

Vale dire che Task = Process = Istanza di un programma in esecuzione.



La cooperazione tra processi è basata sulla comunicazione esplicita.
Un processo `sender` invia messaggi che saranno ricevuti da un processo `receiver`.



Ogni processo è un'istanza di un sottoprogramma in esecuzione e, solitamente, gli stessi sottoprogrammi sono eseguiti con diversi dataset.

Nei sistemi SPMD (Single Program Multiple Data) ogni processo è identificato da un numero intero, chiamato rank, compreso tra 0 e n – 1, dove n è il numero di processi.

Messaggi

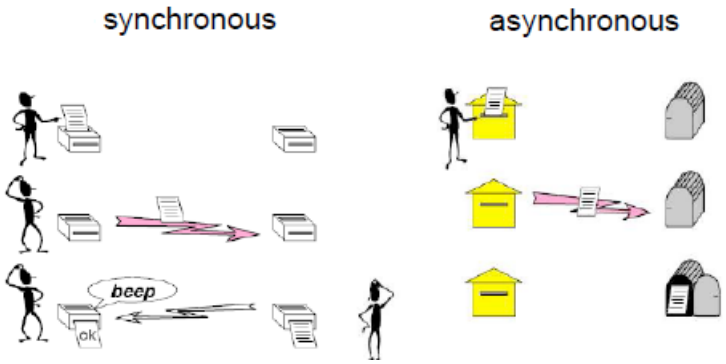
I messaggi sono composti da due parti

- Envelope
  - 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

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 Point – to – Point

Include sono due processi, `sender` e `receiver` e può essere sincrona o asincrona.

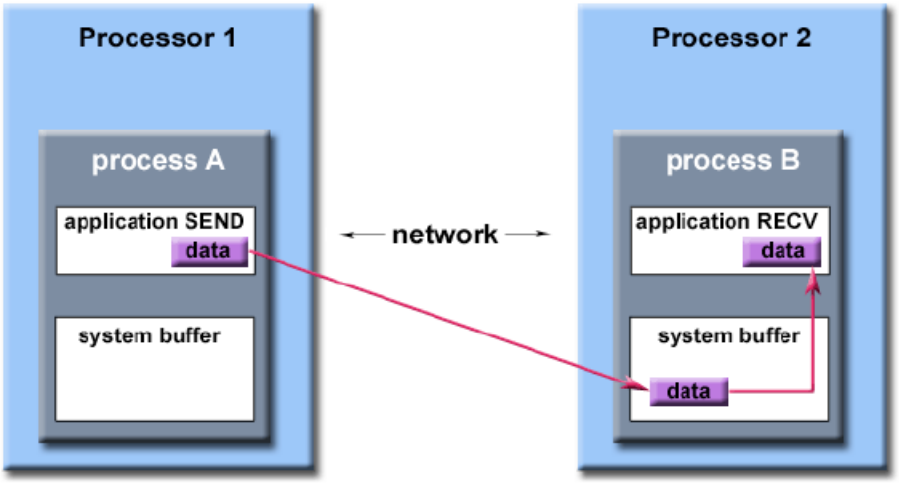


Buffering

I buffer di sistema possono essere presenti da entrambi i lati della comunicazione, seppur di dimensioni limitate.

Bisogna ricordarsi che non può essere controllato dal programmatore e quindi può avere dei comportamenti inaspettati.

I buffer sono implementati in modo diverso in ogni libreria MPI.



Funzionamento

Alcune operazioni potrebbero causare il bloccaggio del chiamante.
Le operazioni non bloccanti consentono al processo di continuare subito dopo la chiamata.
Il processo può effettuare `test` o `wait` sul processo remoto fino al suo completamento.
Bisogna ricordarsi che se si effettua una `wait` dopo un'operazione non bloccante, il risultato sarà comunque un'operazione bloccante.

Operazioni bloccanti

- `synchronous send`
- `asynchronous`
    - `buffered send`
    - `standard send`
- `ready send`
- `recv`
- `sendrecv`

Operazioni non bloccanti
Stesse primitive ma il `sender` non si blocca mai. Bisogna controllare quando i buffer sono riutilizzabili, quando la comunicazione è stata completata ecc. utilizzando

- `test`
- `wait`

Operazioni collective
Questo tipo di comunicazione coinvolge più di due processi, anche tutti.

- Barrier (per la sincronizzazione)
- Data Movement (comunicazioni collective)
    - Broadcast
    - Scatter
    - Gather
- Reduction (computazioni collective)
    - Minimo e massimo
    - Somma
    - OR, AND, ecc.
    - Definite da utente

Funzioni in MPI

`MPI_` è un namespace riservato per le costanti e le routine MPI.
Dopo il prefisso, solo la prima lettera è maiuscola.
Tutte le funzioni MPI ritornano un code integer.
Le costanti hanno nomi scritti in maiuscolo.

```
error = MPI_Xxxxx(parameter, ...);
MPI_Xxxxx(parameter, ...);
```

Header file di MPI

L'header file contiene definizioni, macro e prototipi di funzione necessarie alla compilazione di programmi MPI.

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

Communicators

I communicators sono un insieme di processi che possono comunicare tra di loro
Ognuno di loro possiede

- Nome
- Dimensione (numero di processi)
- Un identificativo per ogni processo, anche se sono uguali tra di loro

Due processi possono comunicare se appartengono allo stesso comunicatore.
Il communicator di default è `MPI_COMM_WORLD`, che è un handler definito in `mpi.h`.

Un processo, per conoscere il proprio rank invocherà il metodo

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Invece, per conoscere la dimensione del proprio communicator il metodo è

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Inizializzare e uscire da MPI

La prima routine MPI che inizializza il communicator di default è

```
int MPI_Init(int *argc, char ***argv)
```

Per uscire dall'ambiente MPI si chiama invece

```
int MPI_Finalize()
```

Programma che stampa il rank del processo e la dimensione del suo communicator

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

#define  MASTER 0

int main (int argc, char **argv)
{
int   numtasks, taskid, len;
char hostname[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Get_processor_name(hostname, &len);
printf ("Hello from task %d on %s!\n", taskid, hostname);

if (taskid == MASTER)
  printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

MPI_Finalize();
}
```

Comunicazione Point – to – Point

Invio

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

Dove

- buf
  Puntatore al messaggio da inviare
- count
  Numero di elementi nel messaggio
- datatype
  Tipo degli elementi del messaggio
- dest
  Rank del recipient
- tag
  Intero non negativo il cui scopo può essere definito dall'utente
- comm
  Communicator

Ricezione

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

Dove

- buf
  Puntatore all'array dove i messaggi ricevuti devono essere salvati
- count
  Numero di elementi nel messaggio
- datatype
  Tipo degli elementi del messaggio

- source
  Rank del mittente, solo i messaggi con un tag specifico possono essere ricevuti
- comm
  Communicator
- status
  Contiene informazioni riguardo il pacchetto che contiene il messaggio da ricevere

Una comunicazione PP ha successo se e solo se

- Il mittente specifica un destinatario con un rank valido
- Il destinatario specifica un mittente con un rank valido
- Mittente e destinatario si trovano nello stesso communicator
- I tag corrispondono
- I datatype corrispondono
- Il destinatario ha un buffer sufficiente

È possibile utilizzare delle wildcards (non per il communicator)

- `MPI_ANY_SOURCE`
  Non si specifica il mittente
- `MPI_ANY_TAG`
  Non si specifica il tag

Bisogna ricordare che i messaggi spediti dallo stesso mittente, nello stesso communicator, con lo stesso tag, allo stesso destinatario, sono consegnati nello stesso ordine con il quale sono stati spediti.
Inoltre, bisogna evitare la starvation, quindi se due processi inviano un messaggio uguale ad un terzo processo che ha impostato un solo `MPI_Recv`, una delle due `MPI_Send` fallirà.

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

int main(int argc, char **argv)
{
	MPI_Status status;
	int rank, size;
	/* data to communicate */
	int data_int;
	/* initialize MPI */
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
	if (rank == 0) {
		data_int = 10;
		MPI_Send(&data_int, 1, MPI_INT, 1, 666, MPI_COMM_WORLD);
	} else if (rank == 1) {
		MPI_Recv(&data_int, 1, MPI_INT, 0, 666, MPI_COMM_WORLD, &status);
		printf("Process 1 receives %d from process 0.\n", data_int);
	}
	MPI_Finalize();
	return 0;
}
```

Misurazione del tempo di esecuzione

```
double MPI_Wtime(void);
```

I timer standard non sono adatti per i programmi MPI, non hanno un'accuratezza sufficiente e non sono portable.
Il tempo di esecuzione di un task è calcolato tramite la differenza del tempo corrente quando il task inizia e il tempo quando l'esecuzione termina.

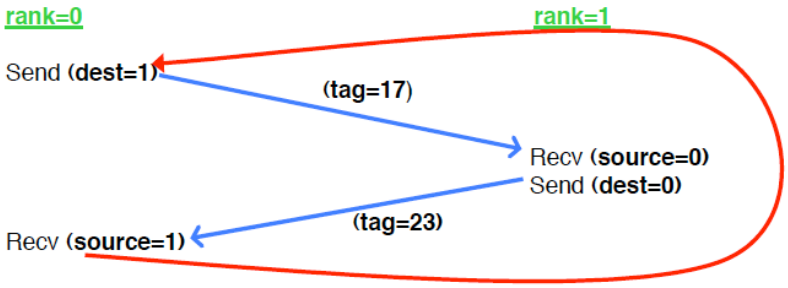
File `mpi_token.c`
Si può usare una send e una receive in cascata per ogni recv dove ognuno prende l'indirizzo del rank dove va spedito il token.

È possibile unire le cose

```
MPI_Sendrecv(&token, 1, MPI_INT, next_rank, 666, &token, 1, MPI_INT, prev_rank,
                                                 666, MPI_COMM_WORLD, &status);
```

Esempio ping pong

Due processi si inviano un messaggio a vicenda.




```
#include <stdio.h>
#include <mpi.h>
#define proc_A 0
#define proc_B 1
#define ping 100
#define pong 101
#define number_of_messages 50
#define length_of_message 1

int main(int argc, char *argv[]) {
    int my_rank, i;
    float buffer[length_of_message];
    double start, finish, time;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == proc_A) {
        start = MPI_Wtime();
        for (i = 1; i <= number_of_messages; i++) {
            MPI_Ssend(buffer, length_of_message, MPI_FLOAT, proc_B, ping,
                                                 MPI_COMM_WORLD);
```

```
            MPI_Recv(buffer, length_of_message, MPI_FLOAT, proc_B, pong,
                                                MPI_COMM_WORLD, &status);
        }

        finish = MPI_Wtime();
        time = finish - start;
        printf("Time for one message: %f seconds.\n",
        (float)(time / (2 * number_of_messages)));
    }
    else if (my_rank == proc_B) {
        for (i = 1; i <= number_of_messages; i++) {
            MPI_Recv(buffer, length_of_message, MPI_FLOAT, proc_A, ping,
                                                MPI_COMM_WORLD, &status);
            MPI_Ssend(buffer, length_of_message, MPI_FLOAT, proc_A, pong,
                                                        MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
}
```

Synchronous nonblocking send

```
MPI_Issend(buf, count, datatype, dest, tag, comm, &request_handle);
MPI_Wait(&request_handle, &status);
```

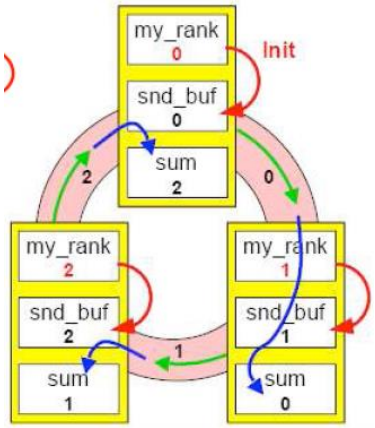
- buf
  Non può essere utilizzato tra le due funzioni
- Issend
  La “I” sta per immeidate, quindi la funziona ritorna il controllo del processo immediatamente
- status
- request_handle
  Consente di riferirsi ad una precisa send
- wait and test
  Utilizzate per sapere quando il receiver ha ricevuto il messaggio

Comunicazioni Nonblocking multiple

È possibile che più comunicazione siano iniziate nello stesso momento, per questo MPI mette a disposizione questi metodi

- `MPI_Waitany / MPI_Testany`
- `MPI_Waitall / MPI_Testall`
- `MPI_Waitsome / MPI_Testsome`

Esempio – Propagazione di un messaggio su un anello




```c
#include <stdio.h>
#include <mpi.h>
#define to_right 201
int main (int argc, char *argv[]) {
    int my_rank, size;
    int snd_buf, recv_buf;
    int right, left;
    int sum, i;
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    right = (my_rank+1) % size;
    left = (my_rank-1+size) % size;
    sum = 0;
    snd_buf = my_rank;
    for( i = 0; i < size; i++) {
        MPI_Issend(&snd_buf, 1, MPI_INT, right, to_right, MPI_COMM_WORLD,
                                                          &request);
        MPI_Recv(&recv_buf, 1, MPI_INT, left, to_right, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
        sum += recv_buf;
        snd_buf = recv_buf;
    }
    printf ("PE%i:\tSum = %i\n", my_rank, sum);
    MPI_Finalize();
}
```

Comunicazioni collettive

MPI fornisce una funzione che implementa dei pattern di comunicazione che coinvolgono più processi.
Esistono tre classi

- All to one
- One to all
- All to all

Operazioni

- Tutti i processi devono comunicare
- Blocking e Nonblocking
- No tag
- I buffer di ricezione devono essere grandi esattamente quanto il messaggio

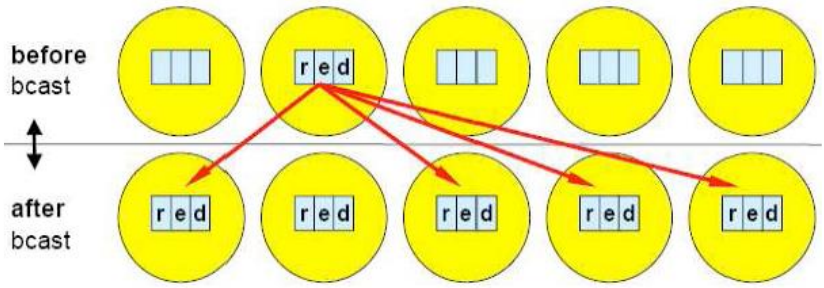
Comunicazioni collettive Blocking – Barrier synchronization

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

Solitamente non necessaria dato che tutti i processi si sincronizzano con la comunicazione stessa.

Comunicazioni collettive Blocking – Broadcast

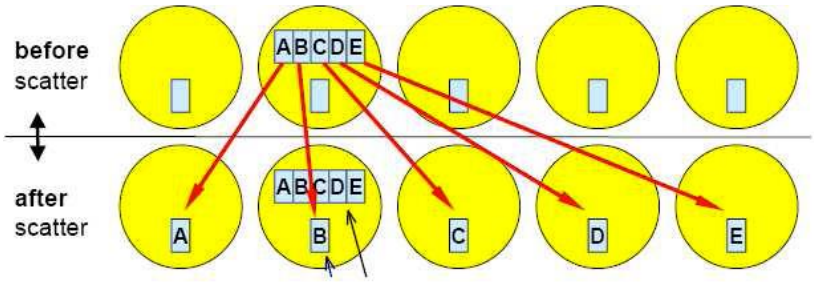
```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
                                                        MPI_Comm comm);
```



- buffer (in/out)
  Indirizzo del buffer
- count (in)
  Numero di elementi che formano il dato da inviare
- datatype (in)
  Tipo di dati da inviare
- root (in)
  Rank del sender
- comm (in)
  Communicator

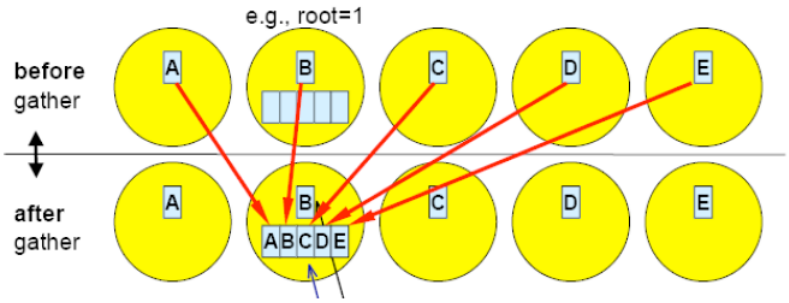
Comunicazioni collettive Blocking – Scatter

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



Comunicazioni collettive Blocking – Gather

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



Ogni processo invia il contenuto al buffer di invio del processo root, che li ordinerà in base al rank.

`MPI_All_gather` è equivalente a `MPI_Gather` ma il processo root effettua anche un broadcast.

Gather appartiene alla classe All to one.

Comunicazioni collettive Blocking – Reduction

Si utilizza per le computazioni con dati distribuiti tra un gruppo di processi.

Supponiamo di avere un'operazione che dati due elementi dello stesso datatype produce un risultato dello stesso tipo, come ad esempio

$$d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1}$$

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

Handle predefiniti per le operazioni	Funzione
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum & location
MPI_MINLOC	Minimum & location

```
for (i = 1 to len)
    inoutvec[i] = inoutvec[i] OP invec[i]
```

Dove OP è un'operazione arbitraria con la sintassi come

```
void my_operator(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype)
```

Per registrare un'operazione definita da utente

```
int MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)
```

Per eliminarla

```
int MPI_Op_Free(op)
```

File `mpi_reduction.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
    MPI_Status status;
    int i, mpi_rank, mpi_size, reduction=0, received;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    if (mpi_rank != 0) {
        MPI_Send(&mpi_rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else {
        for (i = 1; i < mpi_size; i++) {
            MPI_Recv(&received, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
            reduction += received;
        }
    }

    if (mpi_rank==0)
        printf ("reduction: %d \n", reduction);

    MPI_Finalize();
    return 0;
}
```

Laboratorio – MPI CPI

Per la parallelizzazione di `mpi_cpi.c` si utilizza la scomposizione del dominio 1D in sottodomini.
L'intervallo di integrazione tra 0 e 1 viene suddiviso tra i task, se se si hanno N intervalli e p task possiamo suddividere il dominio in p sottodomini.
Ogni sottodominio è ampio $range = \frac{N}{p}$.
Ogni task si occupa della gestione dei dati del sottodominio

```
for (i = mystart; i <= mystart+range ; i++)
```

Al termine vengono sommati i diversi contributi con `MPI_Reduce()`.
Effettuando questa operazione, l'unico overhead che si introduce è proprio la reduction.

Laboratorio – MPI Heat

La tecnica di parallelizzazione usata è la scomposizione delle righe in sottodomini 1D.
Il numero complessivo di righe WNY (Whole NY, numero righe per task) è suddiviso per il numero di task (+2 per le lo scambio dei bordi tra task adiacenti).
Ogni task gestisce quindi un numero di righe NY pari a

$$NY = \frac{WNY}{mpi_{size}} + 2$$

Ogni task esegue le stesse operazioni del programma seriale, ma su un numero inferiore di righe.
Al termine di ogni iterazione è però necessario che ogni task scambi le righe di bordo con i task adiacenti.
Questo viene realizzato dalla funzione `MPI_copy_rows()`, che utilizza due chiamate a `MPI_Senderecv()` per lo scambio delle righe.
Questa comunicazione introduce un overhead che dipende dalla dimensione delle righe e dal numero di iterazioni.
Per ogni task si ha questo tempo di comunicazione per il trasferimento delle righe ad ogni iterazione

$$Tcomm = 4 \cdot \left(Tlat + \frac{N}{Band} \right)$$

Se ad esempio consideriamo righe da 2048 punti (float), abbiamo N = 8KB.

Trascurando la componente non parallelizzabile possiamo calcolare lo speedup

$$S = \frac{Ts}{\frac{Ts}{P} + Toh} = P \frac{Ts}{(Ts + P\,Tcomm)}$$

ovvero, al crescere di P aumenta il peso della comunicazione.

Ad ogni iterazione si prende l'ultima riga di un task e copiarla come prima riga del task successivo.
Le righe da copiare si inviano con una `MPI_Send` della dimensione della riga, all'interno della funzione `MPI_copy_rows`.

La funzione `MPI_print_clormap` per stampare la colormap completa.
Sono necessarie delle comunicazioni perché ogni task ha il proprio stato finale del proprio sottodominio.

```
   //dividiamo le righe tra i rank MPI
   prev_rank = (mpi_rank-1+mpi_size) % mpi_size;
   next_rank = (mpi_rank+1) % mpi_size;
   NX = WNX; // Local NX: numero colonne per rank
   NY = WNY/mpi_size+2; // Local NY: numero righe   per rank
```

Il rank 0 è il rank master.

Esecuzione con Slurm

```
#!/bin/sh
#SBATCH --partition=cpu_guest          # Nome della partizione
#SBATCH --nodes=2                     # numero di nodi richiesti
#SBATCH --ntasks-per-node=8          # numero di cpu per nodo
#SBATCH --time=0-00:10:00             # massimo tempo di calcolo

echo  "#SLURM_JOB_NODELIST      : $SLURM_JOB_NODELIST"
echo  "#SLURM_JOB_CPUS_PER_NODE : $SLURM_JOB_CPUS_PER_NODE"

module purge
module load intel impi
mpicc -O2 mpi_heat.c -o mpi_heat

for P in 1 2 4 8 16
do
mpirun -np $P mpi_heat -c 2048 -r 2048  1> /dev/null  2>> mpi_heat_strong1.dat
done
```

Si può effettuare un checkpointing, in modo da memorizzare su memoria permanente lo stato di un'esecuzione.
Un singolo task dovrebbe agire da interfaccia con lo storage raccogliendo tutti gli stati dei vari task (con un Gather ad esempio), ricostruire la griglia intera e salvarla su disco.
Da disco si caricherà la griglia e con uno Scatter le varie parti vengono inviate ai vari task che possono riprendere l'esecuzione.
Si può realizzare anche con le funzioni di I/O di MPI avendo un filesystem parallelo come GPFS, in modo da poter scrivere un file con più task contemporaneamente.

Laboratorio – MPI + OMP heat

Nella programmazione ibrida (MPI + openMP) viene attivato un solo task MPI per nodo che si occupa della comunicazione con gli altri task.
Il calcolo all'interno del nodo viene parallelizzato con openMP.
Dalle versioni MPI e OpenMP realizziamo la versione ibrida mpi+openMP:

Realizzazione della versione MPI + OMP
Come nella versione OpenMP pura occorre distribuire su più thread le iterazioni sulle righe (`j`) della funzione di Jacobi (Jacobi Iterator)

```
#pragma omp parallel for private (i)
    for(j=1; j<NY-1; j++)
        for(i=1; i<NX-1; i++) { }
```

Stampa dei tempi di esecuzione al termine di ogni iterazione

```
if(mpi_rank == 0) {
    #pragma omp parallel
    #pragma omp single
    fprintf(stderr,"%d %d %d %f %d %d \n", NX, NY, MAX_ITER, t2-t1, mpi_size,
                                                    omp_get_num_threads());
}
```

La stampa viene fatta solo dal master (rank = 0).
Il master attiva la regione parallela (altrimenti la funzione `omp_get_num_threads()` ritornerebbe 1) ma la stampa è fatta da un solo thread (`omp single`).

Compilazione

```
module load intel impi
mpicc -O2 mpi+omp_heat.c -o mpi+omp_heat  -fopenmp
sbatch  mpi+omp_heat_strong.slurm
```

Nella compilazione l'opzione `-fopenmp` serve ad effettuare l'handling delle direttive OpenMP durante la compilazione.
Da notare che lo script slurm chiede due nodi e 8 cores per nodo, poi lancia il programma con un task per nodo, mentre il numero di thread cresce da 1 fino a utilizzare tutti i cores disponibili, mantenendo fissa la dimensione del problema (strong scaling).

Architettura CUDA

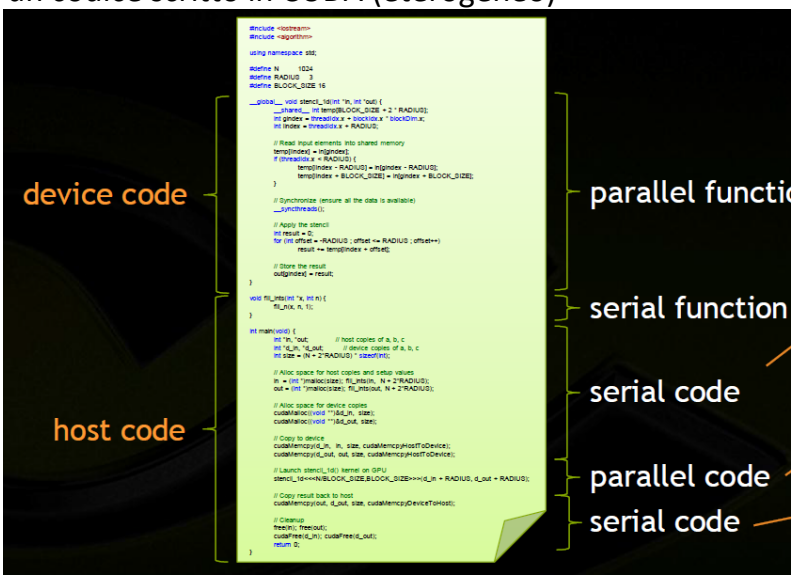
CUDA è un framework mappato sui vari linguaggi di programmazione, utilizzato per la programmazione su GPU.

A differenza del calcolo parallelo, con uno o più processori che dialogano, con il calcolo GPU ci sono solo una CPU e una GPU, quindi due architetture che dialogano.
Ogni CPU ha la sua memoria e la sua GPU dedicata.
È possibile anche mettere più GPU in un singolo nodo.
CPU e GPU possono calcolare in modo collaborativo.

Terminologia

- Host
  CPU e la sua memoria (host memory)
- Device
  GPU e la sua memoria (device memory) si trova in relazione master – slave con l'host, il device dipende completamente dall'host

Come si struttura un codice scritto in CUDA (eterogeneo)



- Device code
  Programmazione della GPU, intrinsecamente parallelo
  - Parallel function
- Host code
  Codice seriale (normale) e poi delle chiamate al device, che sono parallele.
  Quando si ritorna dalle chiamate alla GPU si torna all'esecuzione seriale (si punta ad avere meno esecuzioni seriali possibili).
  È possibile far lavorare l'host mentre è in esecuzione il device, in modo parallelo.
  - Serial function
  - Serial code
  - Parallel code
  - Serial code

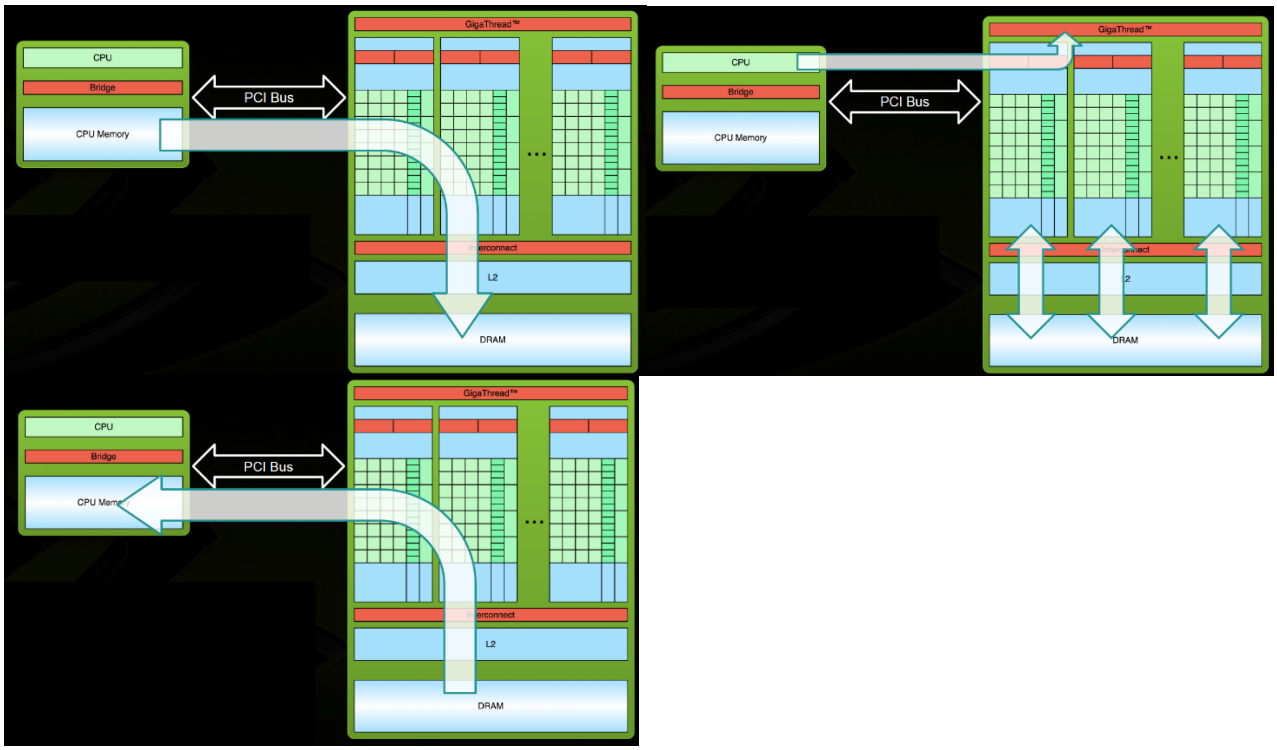
Simple Processing Flow

Le bande di trasferimento (PCI Bus) dei dati possono trasportare grandissime quantità di dati, molto di più rispetto ai normali bus.

Nella memoria della GPU si ha un'alta latenza ma si guadagna sulla banda, si devono riuscire a trasferire tanti byte contemporaneamente per far sì che tutti i processori della GPU possano accedere ai dati contemporaneamente e quindi lavorare parallelamente.

Per lavorare sulla GPU

1. Si copiano i dati di input (programma parallelo) dalla memoria della CPU alla memoria della GPU
2. La CPU carica il programma nella GPU e parte l'esecuzione, la CPU potrebbe continuare ad eseguire altro in modo parallelo
3. Terminata l'esecuzione, si copiano i risultati dalla memoria della GPU a quella della CPU



La best practice è mantenere il più possibile i dati sulla GPU per evitare di sfruttare troppo il bus tra CPU e GPU, che creerebbe un collo di bottiglia.

Programma seriale in C standard

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Per compilare un programma seriale rendendolo adatto alla GPU si utilizza `nvcc`.

```
$ nvcc hello_world.cu
$ a.out
Hello World!
```

Programma con codice per GPU

```c
__global__ void mykernel(void) {
}
int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Si definisce una funzione `void` (anche con parametri) con la keyword `__global__` che indica che la funzione è scritta per l'esecuzione su GPU e che sarà invocata dall'host.

Ci sono anche funzioni che possono sia essere chiamate che eseguite dalla GPU sfruttando la keywork `__device__`.

La chiamata `mykernel<<<1,1>>>()` indica che si avrà un thread a disposizione per l'esecuzione.
I segni maggiore e minore indicano l'organizzazione logica del carico di lavoro.

Il compilatore `nvcc` separa automaticamente i codici per la GPU da quelli per la CPU.
I primi saranno compilati dal compilatore NVIDIA e i secondi dal normale compilatore su CPU.

Tipo di parallelismo

Il calcolo vettoriale è un'attività molto parallelizzabile.

Somma di due numeri in un terzo

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b; //riferimenti memoria GPU
}
```

Gli array (in questo caso array con un solo elemento) si trovano nella RAM, quindi vanno allocati in DRAM (memoria GPU).
Si deve preallocare la memoria perché mentre si esegue sul device non si può allocare memoria.

Per la gestione della memoria del device si hanno le funzioni `cudaMalloc()`, `cudaFree()` e `cudaMemcpy()`.

```
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);
    a = 2; // Setup input values
    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;
}
```

`cudaMemcpyHostToDevice` e `cudaMemcpyDeviceToHost` sono costanti.

Le chiamate per le copie delle memorie sono sincrone, finché la funzione che gira sulla GPU non ritorna non si possono effettuare operazioni sulla memoria. È comunque possibile eseguire codice CPU.

Effettuando una chiamata del tipo

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

si comunica che si vuole lanciare il kernel N volte in parallelo.

Quando si effettua una chiamata del genere ogni esecuzione viene chiamata blocco.
Tutti i blocchi sono riuniti in una griglia di lavoro.
Ogni invocazione si riferisce al proprio blocco, referenziabile tramite `blockIdx.x`.
Ogni blocco gestisce un diverso elemento dell'array.

Si avranno quindi n add che girano e ciascuno gestirà una cella dell'array risultato, per questo è possibile utilizzare `blockIdx.x` come indice.

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

Programma modificato per un array

```
#define N 512
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;
}
```

Thread

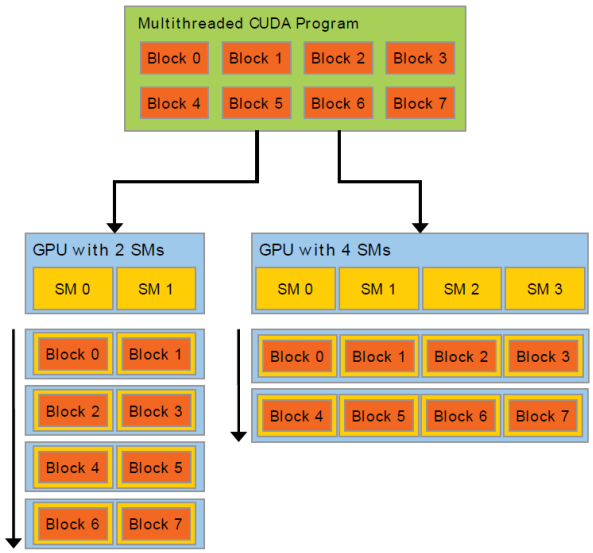
Un blocco può essere suddiviso in tanti thread paralleli.

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

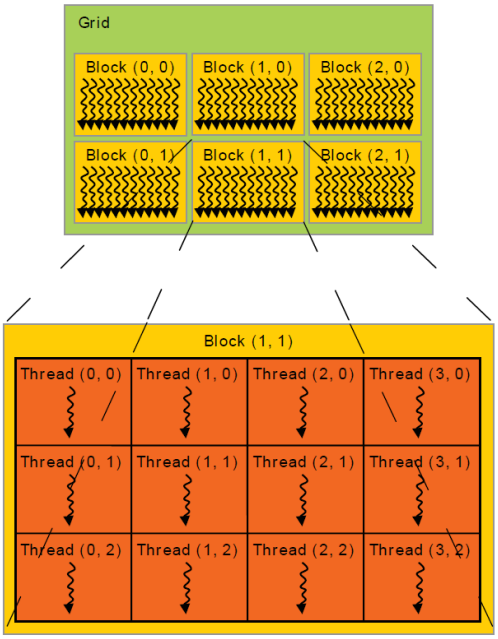
Nel `main` cambia solo la chiamata

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

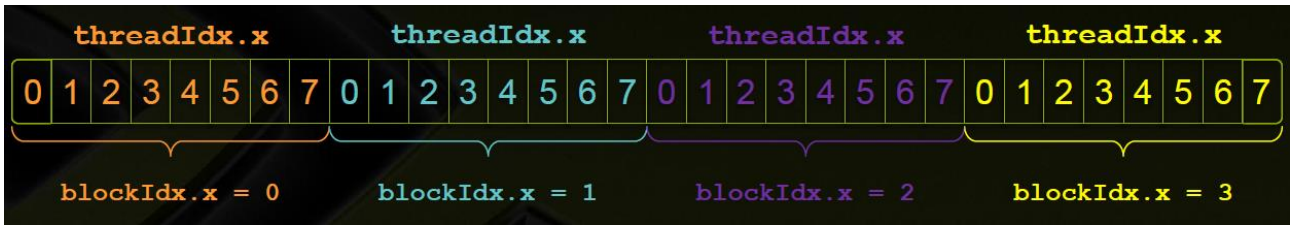
Si ha un grado di parallelismo pari a N come prima, ma ora si parallelizza sui thread e non sui blocchi.



Se si divide il lavoro in blocchi paralleli, a seconda della GPU è possibile mappare i blocchi sui vari processori SM con un grado di parallelismo calcolato in base alla GPU a disposizione.



Sulla griglia di lavoro si ha un'organizzazione con un certo numero di blocchi che a loro volta hanno un numero di thread al loro interno.
Si possono avere un certo numero di processori fisici che possono eseguire contemporaneamente un certo numero di blocchi (al resto pensa lo scheduler).
Ciascun blocco elabora i thread in parallelo, al massimo 1024.
Il processore esegue al massimo 32 thread alla volta in parallelismo vero, se ce ne sono di più questi vengono schedulati ed eseguiti a rotazione.
I thread dello stesso blocco possono condividere informazioni, thread di blocchi diversi per comunicare hanno bisogno di sincronismo della memoria globale, non possono comunicare direttamente.



Un indice per i thread di ogni blocco (con M thread) è dato da

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

Codice aggiornato

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

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

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

Se l'input non è una potenza di due si può calcolare l'indice ma viene creato un blocco in più parzialmente vuoto, i thread vengono comunque creati.

```
__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];
}

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

Supponiamo di avere un array, con tutte le celle che devono essere processato allo stesso modo.
Se la dimensione dell'array è molto grande, la GPU non riuscirebbe a eseguire tutti i thread parallelamente, quindi è stata introdotta la possibilità di suddividere i processori in più SM (Streaming Multiprocessor).
A seconda della GPU si hanno un certo numero di SM, che elaborano in modo indipendente un blocco di lavoro, mappando in parallelo finché ci sono SM disponibili.
Ogni thread all'interno del blocco sarà gestito da un SM (massimo 1024).
I thread vengono raggruppati in gruppi da 32, chiamati WARP.
I blocchi possono essere eseguiti con un ordine qualunque.

Esempio
Supponiamo di avere un array di 8 elementi, si può creare un blocco di lavoro con 8 thread (lanciato ad esempio con `<<<1, 8>>>`).
Si possono anche creare 2 blocchi da 4 thread ognuno (`<<<2, 4>>>`).

L'indice sarebbe in questo caso

```
int i = threadIdx.x + blockIdx.x * 4;
```

Questa operazione si può effettuare fino ad ottenere una situazione del tipo `<<<8, 1>>>`, ovviamente bisogna bilanciare.

Il problema dal punto di vista architetturale è che ogni SM fornisce un certo numero di registri, e, dato che ogni thread ha i propri registri, si rischia di finirli.
Diminuire il numero di thread potrebbe consentire di sfruttare tutti i registri di un SM al meglio.

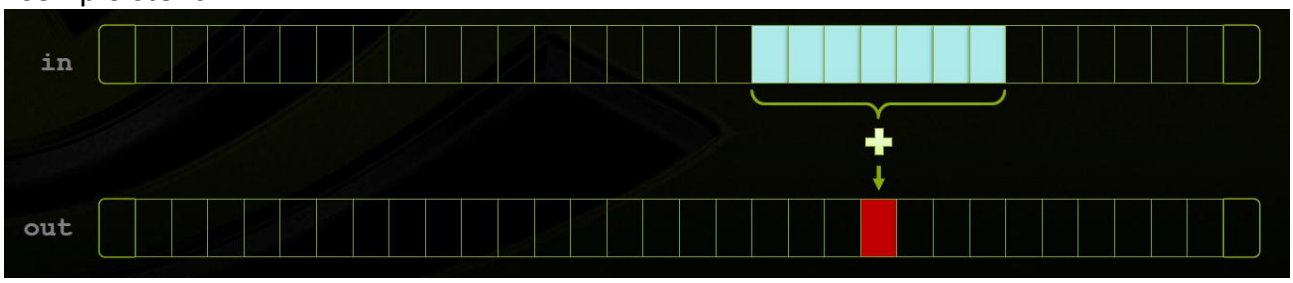
Inoltre, i thread all'interno di un blocco possono essere sincronizzati e hanno a disposizione una memoria condivisa.
La velocità di questa memoria è simile a quella delle cache della CPU, risultando molto comoda seppur limitata (64KB totali, avendo 1024 thread, ogni thread avrebbe a disposizione 64B).

Creare tanti blocchi potrebbe essere sconveniente dato che i thread si ritroverebbero a dover utilizzare la DRAM per comunicare (con costi molto alti).
Tutti i blocchi di un kernel avranno lo stesso numero di thread.
Una scelta tipica è di avere 256 thread per blocco.

dim3 è un vettore di interi con il quale si può passare le dimensioni del blocco e della griglia nella chiamata al kernel.

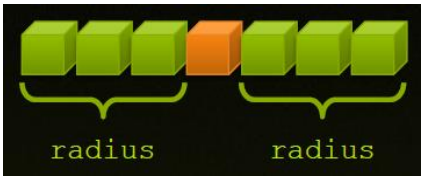
```
dim3 grid(512);          // 512 x 1 x 1
dim3 block(1024, 1024); // 1024 x 1024 x 1
kernel<<< grid, block >>>();


int numBlocks = 1; //1 blocco
dim3 threadsPerBlock(N, N); //N*N thread
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Esempio stencil

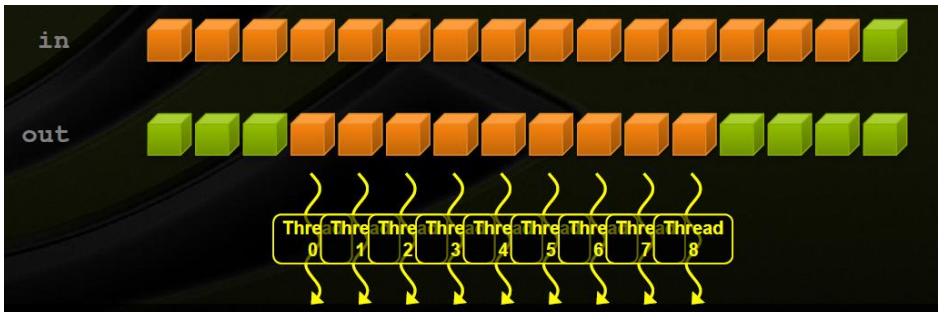


In questo caso si devono elaborare un certo numero di informazioni per calcolarne un'altra.
Solo la cella di output è di competenza del thread.

Esempio pattern di accesso



Ogni elemento lavora tre elementi prima e tre elementi dopo, ogni elemento sarà letto più volte.
Dato che più thread andranno a leggere lo stesso dato, ci sarà uno spreco di memoria.



Una soluzione è scrivere la lettura che viene fatta in DRAM nella memoria condivisa in modo da leggere una sola volta in DRAM.

Nella pratica si può dichiarare una variabile con la keyword __shared__ per fare in modo che venga salvata nella memoria condivisa.

```
__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;
}
```

La shared memory è organizzata per rispondere in parallelo solo se tutti i thread chiedono un banco di memoria diverso (ad esempio il thread 0 chiede la cella 0, il thread 1 la cella 1 ecc.).

Data race

Supponiamo che un thread legga la shared memory prima che questa sia stata scritta.

```
shr[i] = ram[index];
a = shr[i]; //race condition
```

Con queste due istruzioni ci sono problemi di data race perché le istruzioni vengono eseguite con i WARP, che vengono schedulati.
Può succedere che un WARP si trovi più avanti di un altro e che esegua istruzioni successive che potrebbero dipendere da dati non ancora pronti o non ancora gestiti.
La soluzione è una barriera di sincronizzazione, nessun thread può andare avanti se tutti gli altri non hanno finito.

Si utilizza la funzione

```
void __syncthreads();
```

Si può utilizzare per casi di probabile ReadAfterWrite, WriteAfterRead e WriteAfterWrite.

Esecuzioni divergenti

```
if(threadidx.x < 10){
    //solo per i primi 10 thread
}
```

In un controllo del genere entrano solo i primi 10 thread, gli altri no.
Gli altri thread saranno quindi liberi di proseguire la loro esecuzione.
Si rende necessaria quindi una barriera di sincronizzazione per evitare situazioni in cui gli altri thread vadano ad utilizzare dati non ancora creati o gestiti dai primi 10.

Inoltre, si può effettuare branching del codice, eseguendo istruzioni diverse con più thread allineati che fanno le stesse operazioni, spezzando in più WARP.
Si può inserire una maschera di bit che consente di gestire queste esecuzioni differenziate, durante un ciclo di clock sarà eseguito un WARP e al prossimo l'altro WARP.
Non è possibile effettuare una cosa del genere con un'architettura SIMD, per questo operazioni del genere sono utilizzabili nelle architetture SIMT (Single Instruction Multiple Thread).
Tutto questo consente di tenere sotto controllo anche eventuali divergenze del codice con la possibilità di riunificare l'esecuzione una volta terminata la divergenza, sempre sotto una barriera di sincronizzazione.

Coordinazione tra host e device

Tutti i lanci di kernel sono asincroni, quindi il controllo ritorna immediatamente alla CPU.
La CPU deve però sincronizzarsi prima di leggere i risultati.

- `cudaMemcpy()`
  Blocca la CPU finché la copia non è completa.
  La copia comincia solo quando tutte le chiamate CUDA sono terminate
- `cudaMemcpyAsync()`
  Non blocca la CPU
- `cudaDeviceSynchronize()`
  Blocca la CPU finché tutte le chiamate CUDA non sono terminate, è molto costosa ed è preferibile la sincronizzazione a livello di thread

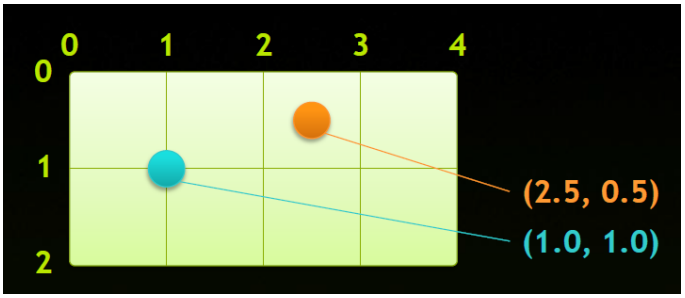
Potrebbe essere ritornato il codice di errore `cudaError_t`

```
cudaError_t cudaGetLastError(void)
char *cudaGetErrorString(cudaError_t)
printf("%s\n", cudaGetErrorString(cudaGetLastError()));

cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

Più host possono condividere un device.
Un solo host può gestire più device

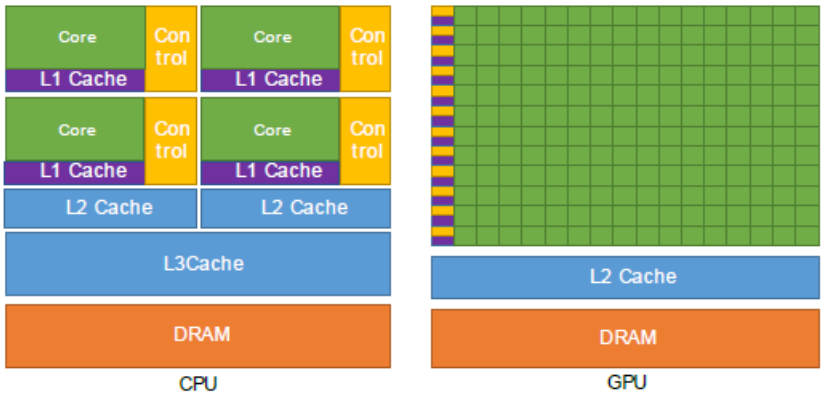
```
cudaSetDevice(i) //select current device
cudaMemcpy(…) //peer-to-peer copies
```

Texture



Nel calcolo custom possono essere utilizzate per mappare delle costanti su un triangolo, la GPU riesce a interpolare, identificare un punto nella texture e rimapparlo sul triangolo.
Possono essere utilizzate per fare cache, ad esempio.
Gestione dei bordi delle matrici, quindi si possono programmare le risposte della texture nel caso in cui si esca dai bordi di una matrice.

Confronto CPU con GPU

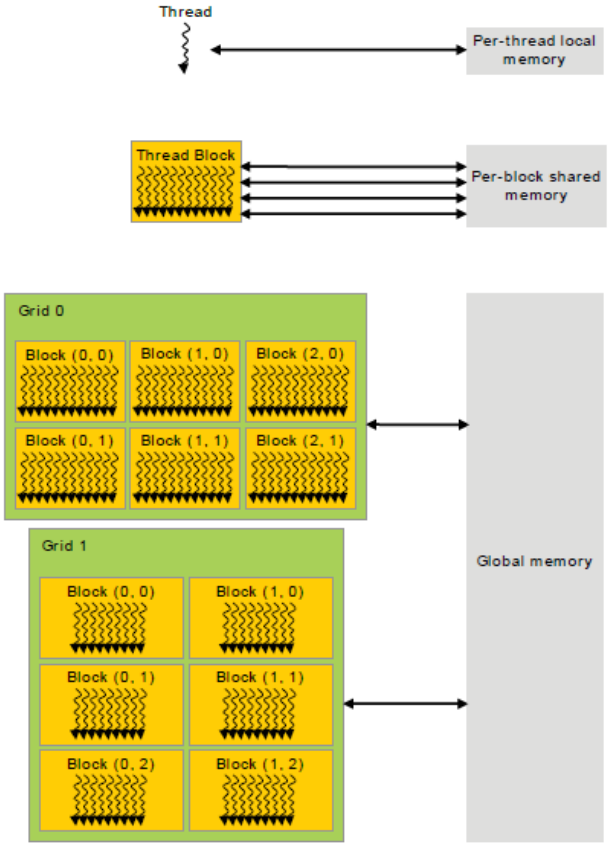


Nella GPU ogni riga verde è un SM.
Si parla di programmazione eterogenea se il lavoro è svolto sia da CPU che da GPU.

Gerarchia di memoria

Un thread ha a disposizione una memoria locale formata da registri (massimo 255, altrimenti il compilatore inizia la rotazione stile caching con la memoria locale).
Un blocco di thread ha a disposizione una memoria condivisa che nasce e muore con il blocco di lavoro e può essere letta solo dal blocco di lavoro.

Tutti i blocchi e tutti i thread possono accedere alla memoria globale, ma è molto costoso e richiede sincronizzazione.



Stream

Ci sono più stream di lavoro.
Le sincronizzazioni, di default sullo stream 0, possono essere suddivise in più stream concorrenti anche a livello di kernel che vengono lanciati, se non ci sono dipendenze sui dati.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
                                    cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size,
                                    inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
                                    cudaMemcpyDeviceToHost, stream[i]);
}
```

Sincronizzazione esplicita

- `cudaDeviceSynchronize()`
  Aspetta finché tutte le operazioni di tutti i thread dell'host non sono completate
- `cudaStreamSynchronize()`
  Prende uno stream come parametro e aspetta che tutte le operazioni di quello stream siano completate. Utilizzabile per la sincronizzazione di uno stream con l'host
- `cudaStreamWaitEvent()`
  Prende uno stream e un evento come parametri e mette in attesa tutti le operazioni dello stream finché l'evento non è completato
- `cudaStreamQuery()`
  Per sapere se tutti i comandi in uno stream sono stati completati

Sincronizzazione implicita

Due comandi di stream diversi non possono eseguirsi concorrentemente se una di queste operazioni viene lanciata dall'host

- Una pagina di memoria viene bloccata dall'host
- Allocazione di memoria da parte del device
- Copia di memoria tra due indirizzi dello stesso device
- Qualsiasi comando CUDA sullo stream NULL

Unified Virtual Address Space

Possibilità di indirizzare allo stesso modo GPU e sua memoria e CPU e sua memoria in modo da avere un solo spazio di indirizzamento, semplificando il codice.
La gestione e la sincronizzazione sono gestite in maniera blackbox.

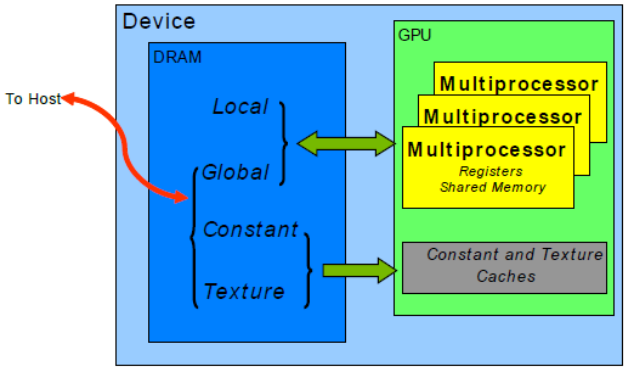
Strategie di ottimizzazione delle prestazioni complessive

- Massimizzare le esecuzioni parallele per sfruttare al massimo i processori della CPU
  L'algoritmo deve genare una quantità di lavoro parallelo sufficiente
- Utilizzo della memoria per sfruttare al massimo il throughput
  Utilizzare il rapporto tra memoria shared e globale, in modo da limitare al massimo gli accessi in memoria globale.
  Si possono sfruttare anche i registri, ma su quelli non si ha un controllo esplicito
- Ottimizzare l'utilizzo delle istruzioni
  Si devono avere i thread il più possibile allineati per quanto riguarda il codice, si devono modificare solo i dati
- Minimizzare gli sprechi di memoria

Timer della GPU

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y,
NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Trasferimento dati tra host e device



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.

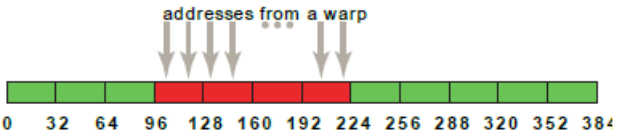
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Modalità di accesso alla memoria globale

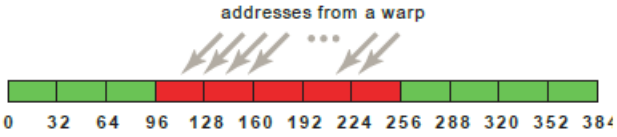
Se non si progetta bene l'accesso alla memoria globale da parte dei thread si perde molto speedup (alta latenza).
Tutte le richieste R/W dei thread di un WARP devono essere ben organizzate in modo che l'accesso sia effettuato con una transazione unica a blocchi creando dei pattern di accesso.

L'accesso concorrente dai vari thread del WARP si unisce automaticamente in un numero di transazioni equivalente al numero di transazioni a 32 byte necessarie a servire tutti i thread, quindi c'è un'organizzazione a blocchi di 32 byte.

Pattern di accesso
Se si ha un thread k che un accesso alla parola `k-th` in un array allineato a 32 byte, questo è un buon pattern.
Se si hanno thread successivi tra loro che accedono a celle contigue di memoria, si ha un buon pattern.



Supponiamo ora che ci sia un accesso sequenziale ma un disallineamento complessivo



Accesso completamente disallineato

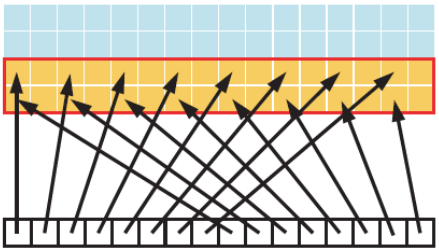
```
__global__ void offsetCopy(float *odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

Supponiamo ora di organizzare i dati in stride (strisce)

```
__global__ void strideCopy(float *odata, float* idata, int stride) {
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
```

```
}
```

Facendo una cosa del genere, si ha un pattern di accesso del tipo



Il consiglio è di non allocare array di struct, ma struct di array.
Siccome si scrivono i dati in modo consecutivo sulla GPU le struct sono sconvenienti.

```
__global__ void coalescedMultiply(float *a, float *c, int M) {
    __shared__ float aTile[TILE_DIM][TILE_DIM],
    transposedTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] =  a[row*TILE_DIM+threadIdx.x];
    transposedTile[threadIdx.x][threadIdx.y] = a[(blockIdx.x*blockDim.x +
                                                  threadIdx.y)*TILE_DIM + threadIdx.x];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++)
        sum += aTile[threadIdx.y][i] * transposedTile[i][threadIdx.x];
    c[row*M+col] = sum;
}
```

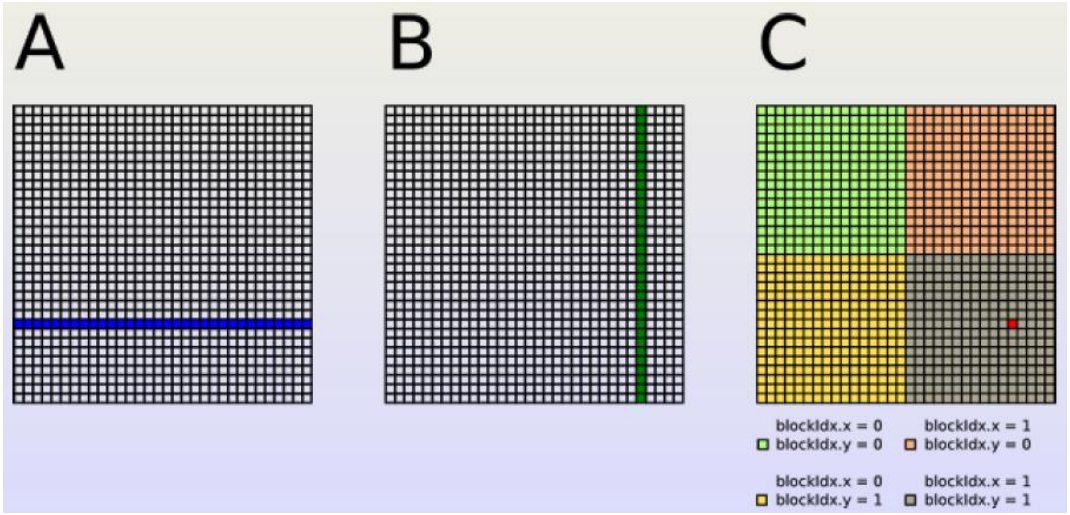
Moltiplicazione di matrici

Il prodotto tra matrici è possibile solo se il numero di colonne della prima è uguale al numero di righe della seconda.
Ogni cella del risultato è il prodotto scalare riga colonna.
Ogni cella di A o B viene letta K volte (K = colonne prima matrice = righe seconda matrice).

```
void main(){
    for i = 0 to M do //per tutte le righe di A
        for j = 0 to N do //per tutte le colonne di B
            /* compute element C(i,j) */
            for k = 0 to K do //si scorrono gli elementi di C
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
                //valore precedente + il prodotto della coppia
            end
        end
    end
}
```

L'unico vincolo per la parallelizzazione è l'accumulo (calcolo degli elementi di C).

Prima idea
Ogni thread calcola una cella di C, quindi produrrà il valore della cella e dovrà eseguire il terzo ciclo `for`.
Ogni thread carica una riga di A e una colonna di B dalla memoria globale.
Calcolato il risultato questo viene caricato in memoria globale.



I vari colori di C indicano 4 blocchi diversi.
Anche se ogni thread lavora una cella sola ci sarà una scrittura coalesced.
Per fare il calcolo del prodotto notiamo che i risultati di ogni blocco sono indipendenti dagli altri, quindi sono parallelizzabili senza sincronizzazione.

```
void main(){
    define A_cpu, B_cpu, C_cpu in the CPU memory
    define A_gpu, B_gpu, C_gpu in the GPU memory
    memcopy A_cpu to A_gpu
    memcopy B_cpu to B_gpu
    dim3 dimBlock(32, 32) //dimensione dei blocchi
    dim3 dimGrid(N/dimBlock.x, M/dimBlock.y) //quanti blocchi generare
    matrixMul<<<dimGrid, dimBlock>>>(A_gpu,B_gpu,C_gpu,K)
    memcopy C_gpu to C_cpu
}
```

Nel metodo `dimGrid` le divisioni sono intere, sarebbe opportuno effettuare dei controlli su N per fare in modo che ci siano abbastanza blocchi disponibili, anche se uno dovesse essere parzialmente pieno.

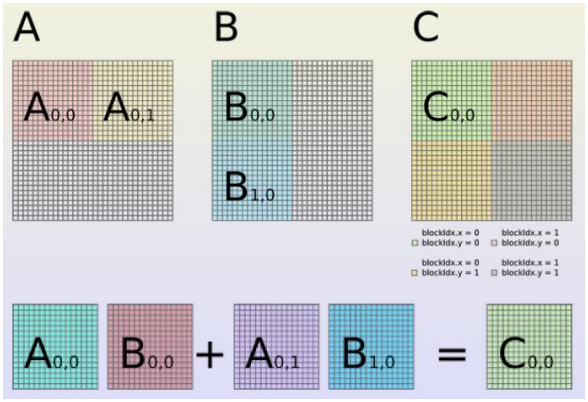
```
__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
    accu = 0 //specifica del thread, ogni thread ne ha una per evitare problemi
    //linearizzazione di i e j
    i = blockIdx.y * blockDim.y + threadIdx.y //riga, è sempre y
    j = blockIdx.x * blockDim.x + threadIdx.x //colonna, è sempre x
    for k = 0 to K-1 do
        accu = accu + A_gpu(i,k) * B_gpu(k,j)
    C_gpu(i,j) = accu
}
```

Uno dei vantaggi è che thread con indici consecutivi andranno a fare una richiesta in memoria di tutta una riga (o colonna), quindi si chiede una fetta di matrice che sarà trasferita come un unico blocco di trasferimento, che è molto efficiente.

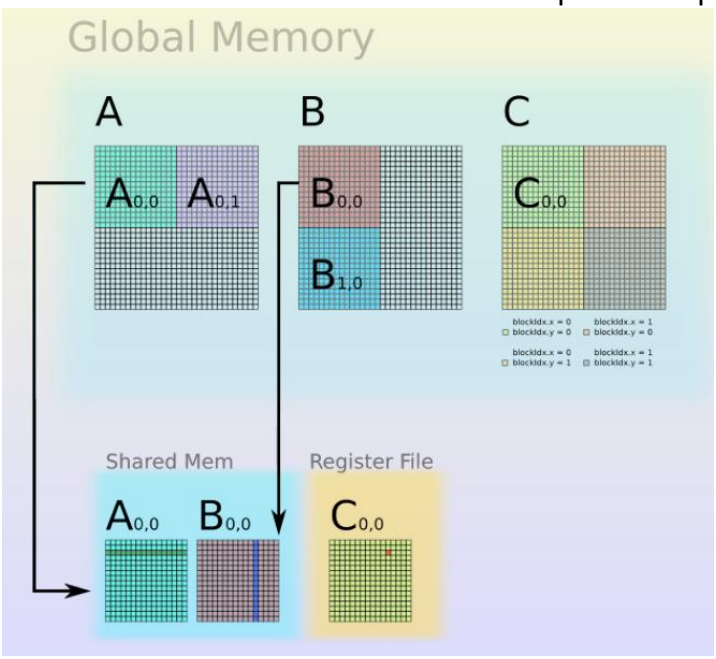
Fissata una riga di A, ci saranno thread per ogni colonna di B, con l'indice `j` che scorre. Ogni thread sulla colonna andrà a richiedere un singolo dato, con l'indice k fissato (inizialmente a 0).

Utilizzo della memoria shared
Per calcolare un prodotto si devono leggere una colonna e una riga.
Spostandosi alla riga successiva si deve rileggere la stessa colonna, e viceversa.
Lavorando in un blocco di lavoro si leggeranno k volte righe e colonne.
L'idea è di effettuare una sola lettura e scrivere tutto in una memoria shared.
Se k fosse molto grande la shared non sarebbe in grado di contenere tutte le informazioni.

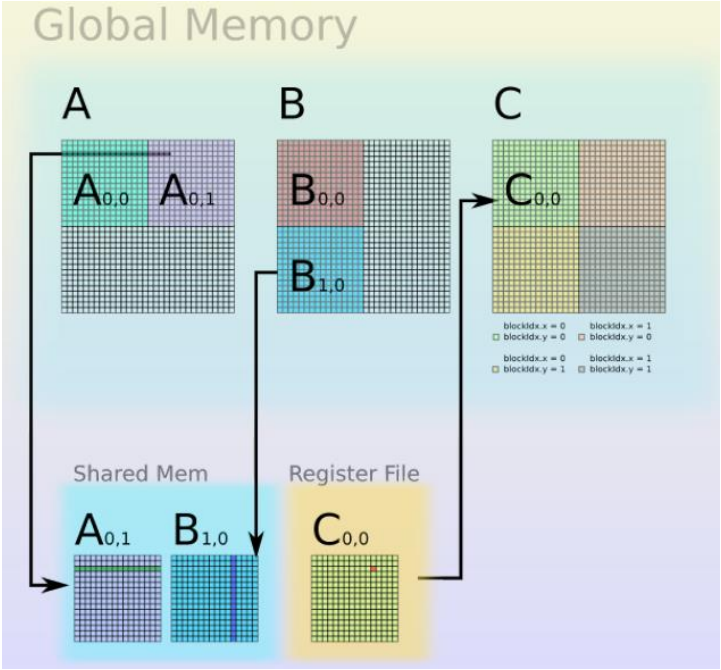
Idea migliore
Partizionare le matrici A e B in diverse zone di lavoro in modo da ridimensionare k e fare operazioni a gruppi.



Il vantaggio è che avendo meno informazioni in una volta si possono spostare in shared.



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.
In `C(0, 0)` ogni thread accumula la sommatoria parziale che gli compete solo per il pezzo di riga e di colonna che gli serve.




```
__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
    __shared__ float A_tile(blockDim.y, blockDim.x) //matrici d'appoggio
    __shared__ float B_tile(blockDim.x, blockDim.y)
    accu = 0 //accumulo
    for tileIdx = 0 to (K/blockDim.x - 1) do
        //si cicla sul numero di blocchi che copre K
        //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)
        //si prendono i dati con trasposizione e si salvano in shared
        __sync() //sincronizzazione
        //da questo punto si può fare il prodotto scalare di competenza
        for tileIdx = 0 to (K/blockDim.x - 1) do
            //carica blocchi di A e B in shared mem
            for k = 0 to blockDim.x do //prodotto scalare (accumulato)
                accu = accu + A_tile(thrIdx.y,k) * B_tile(k,thrIdx.x)
                //k va fino alla dimensione della matrice creata in shared
        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
    //si ha un 32x di prestazioni dato che si fanno meno accessi in ram
}
```

matrixMul.cu snippet

```
    // utilities
    cudaEvent_t start; // si creano i timer
    cudaEvent_t stop;
    float msecTotal;

    // set seed for rand()
    srand(2006); // random per inizializzare matrici

    // allocate host memory for matrices A and B
    unsigned int size_A = WA * HA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float *h_A = (float *)malloc(mem_size_A);
    unsigned int size_B = WB * HB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float *h_B = (float *)malloc(mem_size_B);
    float flop = 2 * (float)WC * (float)HC * (float)WA;

    // initialize host memory
    randomInit(h_A, size_A);
    randomInit(h_B, size_B);

    // allocate device memory
    float *d_A;
    cudaMalloc((void **)&d_A, mem_size_A);
    float *d_B;
    cudaMalloc((void **)&d_B, mem_size_B);

    // allocate device memory for result
    unsigned int size_C = WC * HC;
    unsigned int mem_size_C = sizeof(float) * size_C;
    float *d_C;
    cudaMalloc((void **)&d_C, mem_size_C);

    // allocate host memory for the result
    float *h_C = (float *)malloc(mem_size_C);

    #if CHECK_RESULT == 1 // controllo su CPU, si utilizzano comunque i timer GPU
      // create and start timer
      cudaEventCreate(&start);
      cudaEventRecord(start, NULL);
       // compute reference solution
      float *reference = (float *)malloc(mem_size_C);
      computeGold(reference, h_A, h_B, HA, WA, WB);
      // funzione C++, metodo lento con tre for
      // stop and destroy timer
      cudaEventCreate(&stop);
      cudaEventRecord(stop, NULL);
      cudaEventSynchronize(stop);
      cudaEventElapsedTime(&msecTotal, start, stop);
```

```
    printf("Naive CPU (Golden Reference)\n");
    printf("Processing time: %f (ms), GFLOPS: %f \n", msecTotal,
                                        flop / msecTotal / 1e+6);
#endif

dim3 threads, grid;

/*****************************************************/
/*  CUDA SDK example                                 */
/*****************************************************/

// create and start timer
cudaEventCreate(&start); // timer GPU
cudaEventRecord(start, NULL);
// copy host memory to device
cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
// per il tempo si contano anche le copie
// setup execution parameters
threads = dim3(BLOCK_SIZE, BLOCK_SIZE);
grid = dim3(WC / threads.x, HC / threads.y);
// execute the kernel
matrixMul<<<grid, threads>>>(d_C, d_A, d_B, WA, WB);
// copy result from device to host
cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
// stop and destroy timer
cudaEventCreate(&stop);
cudaEventRecord(stop, NULL);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&msecTotal, start, stop);
printf("GPU SDK Sample\n");
printf("Processing time: %f (ms), GFLOPS: %f \n", msecTotal,
                                    flop / msecTotal / 1e+6);
#if CHECK_RESULT == 1
    // check result
    printDiff(reference, h_C, WC, HC);
#endif
```

matrixMul_naive.cuh

```
__global__ void matrixMul_naive( float* C, float* A, float* B, int wA, int wB) {

  /*
  Array memorizzati linearmente, comodo e funziona bene se la matrice è allocata
  con un numero di elementi potenza di 2 o almeno multiplo di potenza di 2 piccola.
  Il problema è l'allineamento, se la seconda riga della matrice ha 7 elementi,
  il trasferimento a blocchi che va a multipli di 32 parole non basterebbe,
  e sarebbero due blocchi, uno pieno e l'altro per colmare lo spazio mancante.
  */

  /*
  Se si hanno matrici con dimensioni "a caso" conviene fare un tiling delle matrici
  e la GPU effettuerà un pitch, ovvero un'allocazione contigua di dati superiore al
  necessario che garantisce che concatenando con la riga dopo l'inizio sia
  allineato
  */

  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Accumulate row i of A and column j of B
  int i = by * blockDim.y + ty;
  int j = bx * blockDim.x + tx;

  float accu = 0.0;

  for(int k=0; k<wA; k++)
    accu = accu + A[ i * wA + k ] * B[ k * wB + j ]; //linearizzazione a mano

  // Write the block sub-matrix to device memory;
  // each thread writes one element
  C[ i * wB + j ] = accu;

}
```

matrixMul_tiling.cuh

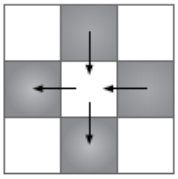
```
__global__ void matrixMul_tiling( float* C, float* A, float* B, int wA, int wB) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(tx, ty) = B[b + wB * tx + ty];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
```

```
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

matrixMul_noBankConflict.cuh

```
/*
Bisogna caricare la shared in modo tale che le scritture fatte da tutti i
thread del WARP siano su banchi diversi invece di scrivere le matrici così
come sono, si scrive la trasposizione sulla shared
*/
      AS(ty, tx) = A[a + wA * ty + tx];
      BS(ty, tx) = B[b + wB * ty + tx];
```

Simulazione trasferimento di calore



Si ha una griglia che contiene valori di temperatura, alcuni sono fissi durante la simulazione, altri devono essere calcolati (convergenza).
C'è bisogno del valore dei vicini di un determinato dato per ricostruire i flussi di calore e aggiornare con un'integrazione il valore successivo in un tempo successivo.

Equazione

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

Si calcolano le differenze tra la cella old e i vicini.

Dato che tutti i conti sono fatti su una "fotografia" bisogna scrivere il risultato su un'altra matrice, altrimenti gli altri calcoli paralleli andrebbero a leggere valori già aggiornati e quindi sbagliati.

Si hanno quindi due buffer, uno con la matrice new (in scrittura) e uno con la matrice old (in lettura), anche se è molto costoso in termini di memoria.

Si può modificare la formula

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

k è la velocità di diffusione del calore, si decide a priori (uguale a 1 in questo caso, non si può aumentare troppo perché si deve tenere conto dei tempi di calcolo).
Si effettua la somma dei 4 vicini immediati meno il proprio valore moltiplicato per 4.

Si ha un thread per ogni cella, in maniera contigua.

Si può immaginare che tutti i thread del WARP chiedano come primo elemento quello sopra, quindi tutti i thread chiederanno la fila di celle sopra, questo sarà il pattern della prima richiesta in blocco in memoria.
Dato che sono indirizzi contigui si ricadrà in un blocco contiguo di memoria, è una lettura efficiente.
Come secondo elemento tutti i thread chiederanno la cella a destra, si tratta sempre di indirizzi contigui. Il terzo valore sarà quello sotto, il quarto quello a sinistra, sempre con lo stesso ragionamento.

Implementazione
Ad alto livello si ha un'iterazione che sostituisce i valori vecchi con i nuovi.
Dato che alcuni elementi non cambiano valore, anche se saranno ricalcolati dovranno essere reimpostati al loro valore costante.

Si ha bisogno, per alcuni punti, di ricopiare i valori costanti nella posizione corretta.
Conviene creare un kernel che si occupa solo della scrittura delle celle costanti.
Questo kernel ha a disposizione la matrice iniziale che descrive le celle costanti, se il valore è diverso da una costante (scelta a priori) si può copiare.

Si preparano quindi kernel separati, il primo per portare tutti i valori da scrivere prima di elaborare, il secondo per l'elaborazione.
Sono sincroni, il secondo parte solo se il primo ha finito.
Per evitare problemi di sincronizzazione e di dipendenze sui dati si può uscire dal kernel che ha terminato, sincronizzare da CPU e far partire un altro kernel.

Se si ha una lettura in memoria globale la suddivisione in blocchi non influenza il processo dato che gli accessi sono organizzati in pattern buoni, anche se si va a richiedere la lettura di un valore gestito da un altro thread di un altro blocco.

Le celle ai bordi, che lavoreranno con 3 o 4 dato al posto di 5, devono essere controllate affinché i thread non facciano richieste in memoria ad indirizzi non definiti.
I controlli in questo caso sono necessari anche se portano a divergenza del codice.

Kernel per la copia dei valori costanti

```
__global__ void copy_const_kernel( float *iptr, const float *cptr ) {
    // 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; //la matrice è linearizzata
    if (cptr[offset] != 0)
        iptr[offset] = cptr[offset];
}
```

Se la memoria costante ha un valore diverso da 0 si va a sovrascrivere sulla matrice old il valore che si trova in modo da resettarlo.

Kernel per il calcolo dell'equazione

```
__global__ void blend_kernel( float *outSrc, const float *inSrc ) {
    //parameters: old and new matrix
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    //gestione bordi
    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) //prima posizione, si incrementa left per riportare x = left
        left++;
    if (x == DIM-1) //ultima posizione, stesso ragionamento
        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);
}
```

Codice completo

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_anim.h"
#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED 0.25f
// globals needed by the update routine
struct DataBlock {
    //tutte le matrici in uno struct di array
    unsigned char *output_bitmap;
    float *dev_inSrc;
    float *dev_outSrc;
    float *dev_constSrc;
    CPUAnimBitmap *bitmap;
    cudaEvent_t start, stop;
    float totalTime;
    float frames;
};

void anim_gpu(DataBlock *d, int ticks) {
    HANDLE_ERROR(cudaEventRecord(d->start, 0));
```

```
    dim3 blocks(DIM/16,DIM/16);
    dim3 threads(16,16);
    CPUAnimBitmap *bitmap = d->bitmap;

    for (int i=0; i<90; i++) {
        //tutti i dati rimangono sempre sulla GPU
        //la CPU serve solo a sincronizzare
        copy_const_kernel<<<blocks,threads>>>(d->dev_inSrc, d->dev_constSrc);
        blend_kernel<<<blocks,threads>>>(d->dev_outSrc, d->dev_inSrc);
        swap(d->dev_inSrc, d->dev_outSrc); //swap di puntatori
    }

    float_to_color<<<blocks,threads>>>(d->output_bitmap, d->dev_inSrc);
    HANDLE_ERROR(cudaMemcpy(bitmap->get_ptr(), d->output_bitmap,
                  bitmap->image_size(), cudaMemcpyDeviceToHost));
    HANDLE_ERROR(cudaEventRecord(d->stop, 0 ));
    HANDLE_ERROR(cudaEventSynchronize(d->stop));
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, d->start, d->stop));
    d->totalTime += elapsedTime;
    ++d->frames;
    printf("Average Time per frame: %3.1f ms\n", d->totalTime/d->frames);
}

void anim_exit(DataBlock *d) {
    cudaFree(d->dev_inSrc);
    cudaFree(d->dev_outSrc);
    cudaFree(d->dev_constSrc);
    HANDLE_ERROR(cudaEventDestroy(d->start));
    HANDLE_ERROR(cudaEventDestroy(d->stop));
}

int main(void) {
    DataBlock data;
    CPUAnimBitmap bitmap(DIM, DIM, &data);
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR(cudaEventCreate(&data.start));
    HANDLE_ERROR(cudaEventCreate(&data.stop));
    HANDLE_ERROR(cudaMalloc((void**)&data.output_bitmap, bitmap.image_size()));
    // assume float == 4 chars in size (i.e., rgba)
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_inSrc, bitmap.image_size()));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_outSrc, bitmap.image_size()));
    HANDLE_ERROR(cudaMalloc((void**)&data.dev_constSrc, bitmap.image_size()));

    //matrice costante
    float *temp = (float*)malloc(bitmap.image_size());

    for (int i=0; i<DIM*DIM; i++) {
```

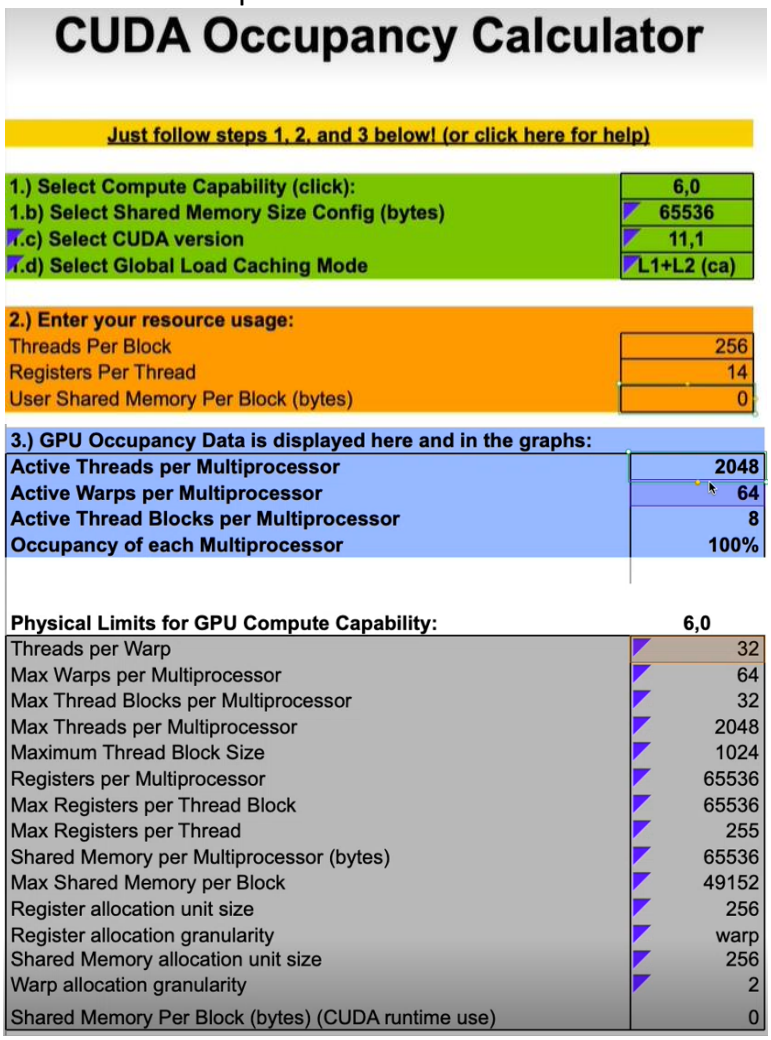
```
        temp[i] = 0;
        int x = i % DIM;
        int y = i / DIM;
        if ((x>300) && (x<600) && (y>310) && (y<601))
            temp[i] = MAX_TEMP;
    }

    temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
    temp[DIM*700+100] = MIN_TEMP;
    temp[DIM*300+300] = MIN_TEMP;
    temp[DIM*200+700] = MIN_TEMP;

    for (int y=800; y<900; y++) {
        for (int x=400; x<500; x++)
            temp[x+y*DIM] = MIN_TEMP;
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_constSrc, temp, bitmap.image_size(),
                                            cudaMemcpyHostToDevice));
    for (int y=800; y<DIM; y++) {
        for (int x=0; x<200; x++)
            temp[x+y*DIM] = MAX_TEMP;
    }

    HANDLE_ERROR(cudaMemcpy(data.dev_inSrc, temp, bitmap.image_size(),
                                            cudaMemcpyHostToDevice));
    free(temp);
    bitmap.anim_and_exit((void (*)(void*,int))anim_gpu, (void(*)(void*))anim_exit);
}
```

Funzione utilizzata in heat_gpu.cu

```
float *h_T = (float *)calloc(NX * NY, sizeof(float));
```

La funzione `calloc` è una funzione C per richiedere l'allocazione di un blocco di memoria contiguo.
Il primo parametro indica il numero di celle richiesto, il secondo la dimensione di ogni cella.

Occupancy Calculator
Foglio Excel per capire come si sta utilizzando la scheda.
Nell'immagine sono descritti tutti i parametri che devono essere inseriti.



CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	6,0
1.b) Select Shared Memory Size Config (bytes)	65536
1.c) Select CUDA version	11,1
1.d) Select Global Load Caching Mode	L1+L2 (ca)

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	14
User Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

Physical Limits for GPU Compute Capability:	6,0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	2
Shared Memory Per Block (bytes) (CUDA runtime use)	0

GPU e MPI

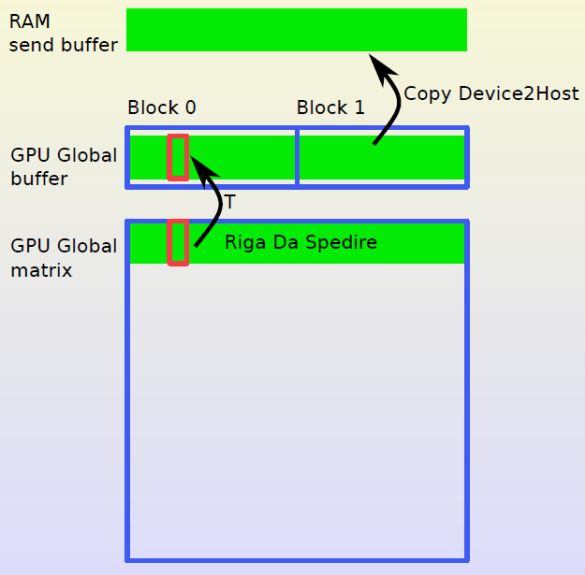
Si aggiunge un altro livello di parallelismo, si possono utilizzare più GPU in parallelo.
Si hanno n processi ciascuno con una GPU dedicata.
Le GPU possono trovarsi anche su nodi diversi.

Si prepara un modello che cresce in maniera regolare, si ha una matrice fatta da due pezzi, si taglia a metà e ogni metà viene assegnata ad una GPU.

Trasferimento del tipo

GPU→RAM (rank 0)→MPI→RAM (rank 1)→GPU

Si ha un trasferimento esplicito che è molto pesante.



Se ci si trova sullo stesso nodo, con n GPU e n processi che girano ci si vuole assicurare che ogni processo prenda la GPU giusta e che tutti ne prendano una diversa.

```
    char host_name[MPI_MAX_PROCESSOR_NAME]; //collezione hostname
    char (*host_names)[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm nodeComm;
    int  n, namelen, color, rank, nprocs;
    int rank_node, gpu_per_node; /// sul singolo nodo
    size_t bytes;
    int dev;
    struct cudaDeviceProp deviceProp;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Get_processor_name(host_name,&namelen);
    bytes = nprocs * sizeof(char[MPI_MAX_PROCESSOR_NAME]);
    host_names = (char (*)[MPI_MAX_PROCESSOR_NAME]) malloc(bytes);
    strcpy(host_names[rank], host_name);
    for (n=0; n<nprocs; n++)
        MPI_Bcast(&(host_names[n]),MPI_MAX_PROCESSOR_NAME, MPI_CHAR, n,
                                                    MPI_COMM_WORLD);
```