

Marco Casu

❖ Programmazione di Sistemi Multicore ❖



SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Copyright (c) 2024 Marco Casu. Permission is granted to copy, distribute and/or modify this document under the terms of the [GNU Free Documentation License](#), Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Questo documento è un resoconto degli appunti (eventualmente integrati con libri di testo) tratti dalle lezioni del corso di Programmazione di Sistemi Multicore per la laurea triennale in Informatica. Se dovessi notare errori, ti prego di segnalarli.



INDICE

1 Parallelismo : Motivazioni	4
1.1 Introduzione	4
1.2 Modelli di Parallelismo	5
2 Memoria Distribuita : MPI	8
2.1 La libreria OpenMp	8
2.2 Rank e Comunicazione	9
2.3 Design di Programmi Parallelî	11
2.3.1 Pattern di Design Parallelo	12
2.4 Comunicazione non Bloccante e Comunicazione Collettiva	13
2.4.1 Send e Recv Immediate	14
2.4.2 Esempi di Applicazione	14
2.4.3 Operazioni Collettive	16
2.5 Valutazione del Tempo	20
2.5.1 Scalabilità Forte e Scalabilità Debole	23
2.6 Operazioni su Vettori e Matrici	24
2.6.1 Scatter e Gather	24
2.6.2 Ultime Collettive di tipo "All"	28
2.7 Tipi di Dato Custom	28
3 Memoria Condivisa : Posix Threads	30
3.1 Introduzione ai Thread	30
3.1.1 Prodotto Matrice-Vettore con Pthread	31
3.2 Sezioni Critiche	32
3.2.1 Busy Waiting e Mutex	33
3.2.2 Semafori, Barriere e Variabili di Condizione	34
3.2.3 Stima di π con Pthread	37
3.3 Read-Write Lock	38
3.4 Funzioni Thread-Safe	41
3.5 Combinazione di Thread ed MPI	42
3.5.1 Thread Pinning	43
4 Richiamo di Architetture	44
4.1 Caching	44
4.1.1 Livelli della Cache	44
4.2 La Cache nei Sistemi Multicore	45
4.2.1 False Sharing	46
4.2.2 Organizzazione della Memoria	46

5 Gestione dei Thread : OpenMP	48
5.1 Direttive pragma	48
5.2 Mutua Esclusione	50
5.2.1 Integrazione Numerica con OpenMP	50
5.2.2 Riduzione	51
5.3 Scoping	52
5.4 Cicli For Parallel	53
5.4.1 Cicli Annidati	54
5.4.2 Iterazioni Dipendenti	56
5.4.3 Risoluzione delle Dipendenze RAW	57
5.4.4 Rimozione di dipendenze WAR e WAW	61
5.5 Schedluing Loops	62
5.6 Argomenti Extra sulle Sezioni Critiche	64
5.6.1 Sezioni Parallel	64
5.7 Impatto del False Sharing	65
6 Programmazione di GPU : CUDA	67
6.1 Architettura della GPU	67
6.2 Introduzione a CUDA	69
6.2.1 Modello di Esecuzione	70
6.3 Struttura del Codice	70
6.3.1 Disposizione dei Thread	72
6.3.2 Cambio di Contesto e Proprietà del Dispositivo	74
6.4 Gestione della Memoria	76
6.4.1 Somma fra Vettori in CUDA	76
6.4.2 Tipi di Memoria	77
6.4.3 Constant Memory	79
6.4.4 Global Memory	80
6.4.5 Memoria Pinnata	80
6.5 Stima delle Performance	80
6.5.1 Roofline Model	81
6.5.2 MPI e CUDA	81
6.6 Tiling	81

CAPITOLO

1

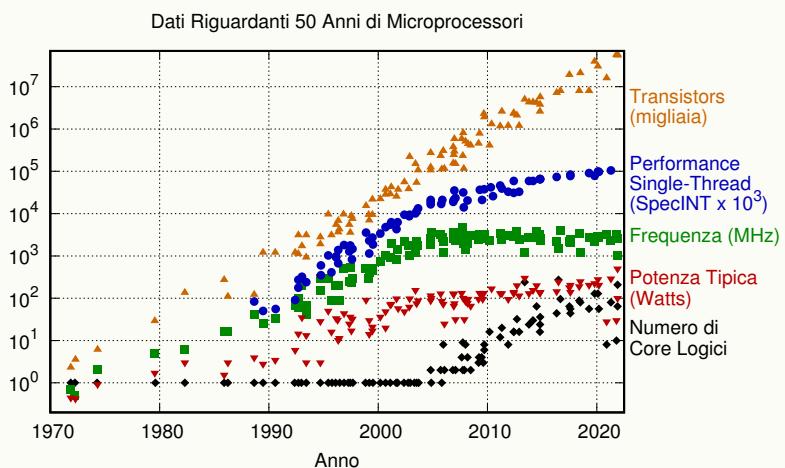
PARALLELISMO : MOTIVAZIONI

1.1 Introduzione

In una *GPU* (Graphics Processing Unit), nota anche come scheda video, ci sono circa 80 miliardi di transistor, e vengono utilizzate per allenare i grossi modelli di intelligenza artificiale, i quali necessitano migliaia di GPU, non è un caso se *Nvidia* ad oggi, con il boom dell'IA, è una delle aziende più quotate al mondo. Le GPU, e la loro programmazione, sono uno fra i principali argomenti di questo corso.

L'evoluzione dell'hardware, ha portato i grossi sistemi di computazione, ad essere formati da svariate unità di calcolo piuttosto che una singola unità molto potente, i processori stessi di uso comune, ad oggi sono composti da più *core*.

La legge di Moore riguarda una stima empirica che mette in correlazione lo scorrere del tempo con l'aumentare della potenza di calcolo dei processori, se inizialmente, a partire dagli anni 70, tale potenza raddoppiava ogni due anni, ad oggi tale andamento è andato rallentando, raggiungendo un incremento 1.5 in 10 anni.



L'obiettivo di costruire calcolatori sempre più potenti è dipeso dalla necessità dell'Uomo di risolvere problemi sempre più complessi, come ad esempio, la risoluzione del genoma umano.

Il motivo per il quale non è possibile costruire processori monolitici sempre più potenti, risiede in un *limite fisico* riguardante la densità massima possibile dei transistor in un chip.

1. transistor più piccoli → processori più veloci
2. processori più veloci → aumento del consumo energetico
3. aumento del consumo energetico → aumento del calore
4. aumento del calore → problemi di inaffidabilità dei transistor

♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪

1.2 Modelli di Parallelismo

L'informatico che intende scrivere del codice per un sistema multicore, deve esplicitamente sfruttare i diversi core, limitandosi a scrivere un codice sequenziale, non starebbe sfruttando a pieno l'hardware a disposizione, rendendo il processo meno efficiente di quanto potrebbe essere.

La maggior parte delle volte, un algoritmo sequenziale, non può essere direttamente tradotto in un algoritmo parallelo, per questo bisogna scrivere il codice facendo riferimento all'hardware di destinazione. Si consideri adesso il seguente codice sequenziale, che ha lo scopo di sommare n numeri dati in input.

```

1 sum = 0;
2 for (i=0; i<n; i++){
3     x = compute_next_value (...);
4     sum += x;
5 }
```

Si vuole rendere tale algoritmo parallelo, sapendo di essere a disposizione di p core.

```

1 local_sum = 0;
2 first_index = ...;
3 last_index = ...;
4 for (local_i=first_index; first_index<last_index; local_i++){
5     local_x = compute_next_value (...);
6     local_sum += local_x;
7 }
```

In tale esempio, ogni core possiede le sue variabili private non condivise con gli altri core, ed esegue indipendentemente il blocco di codice. Ogni core conterrà la somma parziale di n/p valori.

Esempio (24 numeri, 8 core) :

valori : 1, 4, 3, 9, 2, 8, 5, 1, 1, 6, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1, 2, 3, 9

core	0	1	2	3	4	5	6	7
local_sum	8	19	7	15	7	13	12	14

A questo punto, per ottenere la somma totale, vi sarà un core *master* che riceverà le somme parziali da tutti gli altri core, per poi eseguire la somma finale.

```

1 if (master){
2     sum = local_sum;
3     for c : core{
4         if (c!=self){
5             sum += c.local_sum;
6         }
7     }
8 } else{
9     send local_sum to master;
10 }
```

Dividere i dati per poi far eseguire la stessa computazione ai diversi nodi è la forma più semplice di parallelismo. La soluzione adottata non è ideale, in quanto, in seguito al calcolo delle somme parziali, tutti i core escluso il master non staranno eseguendo calcoli. Una possibile idea alternativa è di far sì che a coppie i nodi si condividano le somme parziali per poi calcolarne una somma comune, sviluppando uno scambio di dati ad albero, come mostrato in figura 1.1.

Possiamo identificare due tipi di parallelismo :

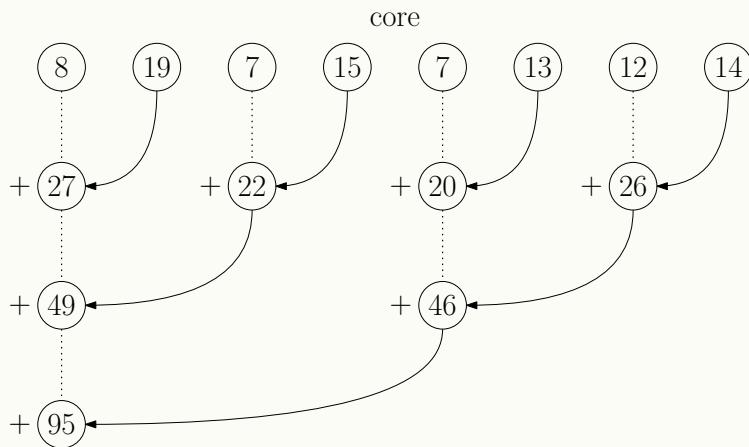


Figura 1.1: calcolo somme a coppie

- **parallelismo dei task** : fra i core vengono divise diverse attività che vengono svolte autonomamente.
- **parallelismo dei dati** : i dati da elaborare vengono divisi, ogni core eseguirà la stessa computazione ma su una porzione diversa dei dati.

Quando si scrive un programma parallelo bisogna prestare attenzione alla *sincronizzazione* dei processi, in quanto potrebbero dover accedere ad una stessa area di memoria. Risulta cruciale saper mettere in *comunicazione* i vari core, e suddividere equamente il *carico di lavoro* fra di essi. Verranno considerate 4 diverse tecnologie per la programmazione multicore :

- *MPI* (Message Passing Interface) [libreria]
- *Poix Threads* [libreria]
- *OpenMP* [libreria e compilatore]
- *CUDA* [libreria e compilatore]

La programmazione delle GPU richiederà un diverso compilatore, e non il solito *gcc*, in quanto l'architettura della scheda video differisce da quella del processore, e con essa le istruzioni.

I sistemi paralleli possono essere categorizzati sotto vari aspetti.

- **shared memory** : Tutti i core accedono ad un'area di memoria comune. L'accesso e la sincronizzazione vanno gestiti con cautela.
- **distributed memory** : Ogni core ha un area di memoria privata, e la comunicazione avviene attraverso un apposito canale per lo scambio dei messaggi.

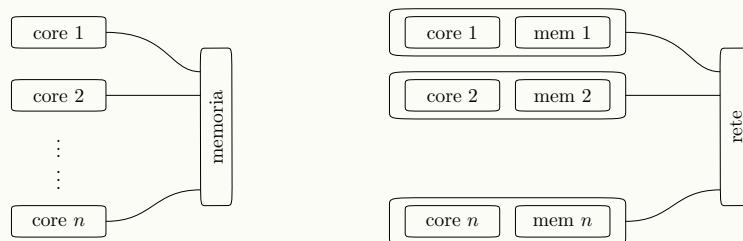


Figura 1.2: modelli di parallelismo

Vi è un'altra suddivisione nei sistemi paralleli :

- **MIMD** : Ogni core ha una control unit indipendente, diversi core possono eseguire diverse istruzioni nello stesso momento.

- **SIMD** : Vi è un singolo program counter per tutti i core, che eseguono in maniera parallela le stesse istruzioni. Due core non possono eseguire operazioni diverse nello stesso momento.

Le GPU hanno una struttura *SIMD*.

	shared memory	distributed memory
SIMD	CUDA	
MIMD	Pthreads/OpenMP	MPI

Fin'ora sono stati utilizzati 3 termini chiave riguardante i tipi di programmazione, sebbene non vi sia una definizione comunemente accettata, la seguente verrà adottata in tale contesto :

- *concorrente* : più processi sono attivi in uno stesso momento
- *parallela* : diverse entità cooperative che operano in maniera ravvicinata per un obiettivo comune.
- *distribuita* : diverse entità cooperative.

La programmazione parallela o distribuita implica che sia anche concorrente, non è vero il contrario.

CAPITOLO

2

MEMORIA DISTRIBUITA : MPI

MPI è una libreria standard (avente varie implementazioni) necessaria allo sviluppo di codice multiprocesso a memoria distribuita. Precisamente, ogni core ha una memoria privata inaccessibile dall'esterno, e la comunicazione avviene attraverso una rete di interconnessione, (ad esempio, un bus), tale modello è detto **message passing**.

2.1 La libreria OpenMpi

Alla compilazione ed avvio di un programma che sfrutta MPI, ogni core eseguirà il programma, sarà la logica di esso a suddividere il carico di lavoro, tramite i costrutti decisionali. Verrà utilizzata un'implementazione nota come *openMpi*, è possibile installare la libreria su sistemi operativi linux tramite il comando

```
sudo apt-get install libopenmpi-dev
```

Il seguente esempio, mostra un programma che scrive sulla console una stringa, e tramite MPI, tale processo è avviato su ogni core.

```
1 #include <stdio.h>
2 #include <mpi.h>
3 //voglio lanciare il programma su piu unita di calcolo
4 int main(int argc, char **argv){
5     int p = MPI_Init(NULL,NULL);
6     //Il parametro in output di MPI_Init e' uno status sull'errore
7     if(p == MPI_SUCCESS){
8
9         } else{
10             printf("qualcosa e' andato storto");
11             MPI_Abort(MPI_COMM_WORLD,p);
12             //Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
13         }
14         printf("hello world");
15         MPI_Finalize(); //Serve per terminare la libreria
16         return 0;
17 }
```

I programmi MPI non vengono compilati con `gcc`, ma con `mpicc`

```
mpicc hello_world.c -o hello_world.out
```

Una volta ottenuto l'eseguibile, è possibile lanciare il programma con `mpirun` specificando il numero di core sulla quale verrà eseguito il programma, tale numero, se non specificato con apposite flag, deve

essere minore o uguale al numero di core fisici presenti sulla macchina.

```
mpirun -n 4 hello_world.out
```

Ogni funzione della libreria ha una dicitura che inizia con "*MPI_*". Ogni funzione di libreria deve essere chiamata fra

- `MPI_Init` - configurazione ed avviamento della libreria
- `MPI_Finalize` - chiusura e deallocazione della memoria

Tali righe stabiliscono il blocco di codice in cui verranno eseguite funzioni MPI.

```
~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~
```

2.2 Rank e Comunicazione

Ogni processo MPI è univocamente identificato da un numero intero detto *rank*, se p processi sono attivi, avranno gli identificatori $0, 1, 2 \dots, p - 1$.

Un **comunicatore** è un insieme di processi, i quali hanno la possibilità di scambiarsi messaggi, si può pensare ad un comunicatore come un etichetta, e processi con la stessa etichetta possono comunicare fra loro. È identificabile nel codice tramite la struttura dati `MPI_Comm`, e all'avvio di MPI, viene sempre definito un comunicatore di default `MPI_COMM_WORLD` che contiene tutti i processi.

L'identificatore di ogni processo è in realtà relativo ad ogni comunicatore, due processi diversi possono condividere il rank se relativo a comunicatori diversi. Ci sono due funzioni importanti che riguardano questi ultimi

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)` : Prende in input un comunicatore ed un numero intero, e salva dentro tale numero il rank del processo chiamante relativo al comunicatore dato.
- `int MPI_Comm_size(MPI_Comm comm, int *size)` : Prende in input un comunicatore ed un numero intero, e salva dentro tale intero il numero di processi all'interno del comunicatore.

```

1 #include <stdio.h>
2 #include <mpi.h>
3 //voglio lanciare il programma su piu unita di calcolo
4 int main(int argc, char **argv){
5     int p = MPI_Init(NULL,NULL);
6     //Il parametro in output di MPI_Init e' uno status sull'errore
7     if(p == MPI_SUCCESS){
8
9         } else{
10             printf("qualcosa e' andato storto");
11             MPI_Abort(MPI_COMM_WORLD,p);
12             //Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
13         }
14         int size;
15         MPI_Comm_size(MPI_COMM_WORLD, &size);
16         int rank;
17         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18         printf("hello world, im the process %d/%d",rank,size);
19         MPI_Finalize(); //Serve per terminare la libreria
20         return 0;
21     }
```

La comunicazione avviene tramite due funzioni, il cui comportamento è simile alla comunicazione tramite `pipe`.

L'invio dei messaggi avviene tramite `int MPI_Send`, i cui parametri sono

- `void* msg_buf_p` l'area di memoria da trasferire al processo destinatario

- `int msg_size` il numero di elementi (non l'occupazione in byte) del messaggio da trasferire
- `MPI_Datatype msg_type` il tipo di elemento da trasferire. Sono definiti dei tipi standard che incorporano tutti i tipi più comuni del *C*
- `int dest` il rank del processo destinatario
- `int tag` un tag da dare al messaggio per identificarlo
- `MPI_Comm communicator` il comunicatore su cui avviene la comunicazione

Può dipendere dall'implementazione, ma solitamente quando un processo fa una `MPI_Send`, si arresta finché il messaggio inviato non viene ricevuto dal destinatario, allo stesso modo, un destinatario che si appresta a ricevere un messaggio viene arrestato fino al ricevimento. Le chiamate di comunicazione MPI sono quindi bloccanti.

Per ricevere dati, viene utilizzata la chiamata `MPI_Recv` i cui parametri sono

- `void* msg_buf_p` l'area di memoria su cui verrà salvato il messaggio
- `int buf_size` il numero di elementi (non l'occupazione in byte) del messaggio da ricevere
- `MPI_Datatype buf_type` il tipo di elemento da ricevere
- `int source` il rank del processo mittente
- `int tag` il tag del messaggio da ricevere
- `MPI_Comm communicator` il comunicatore su cui avviene la comunicazione
- `MPI_Status* status` lo status riguardante l'esito della comunicazione

OpenMp definisce la seguente lista di tipi `MPI_Datatype` :

<code>MPI_CHAR</code>	carattere
<code>MPI_INT</code>	intero
<code>MPI_FLOAT</code>	float a singola precisione
<code>MPI_DOUBLE</code>	float a doppia precisione
<code>MPI_LONG</code>	intero long
<code>MPI_SHORT</code>	intero short
<code>MPI_UNSIGNED_CHAR</code>	carattere senza segno
<code>MPI_UNSIGNED_INT</code>	intero senza segno
<code>MPI_UNSIGNED_LONG</code>	intero long senza segno
<code>MPI_UNSIGNED_SHORT</code>	intero short senza segno

Il seguente programma fa sì che ogni processo invii un messaggio al processo di rank 0, e quest'ultimo lo stampi a schermo.

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char **argv)
5 {
6     int p = MPI_Init(NULL, NULL);
7     // Il parametro in output di MPI_Init è uno status sull'errore
8     if (p != MPI_SUCCESS)
9     {
10         printf("qualcosa e' andato storto");
11         MPI_Abort(MPI_COMM_WORLD, p);
12         // Con MPI_Abort tutti i processi su tutti i core avviati verranno terminati
13     }
14     int size;
15     MPI_Comm_size(MPI_COMM_WORLD, &size);
16     int str_size = 256;
17     int rank;
```

```

18 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
19 if (rank == 0)
20 {
21     printf("hello world, I am process 0. I will receive and print.\n", rank, size);
22     char str[str_size];
23     for (int i = 1; i < size; i++)
24     {
25         MPI_Recv(str, str_size, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26         printf("(STRING RECEIVED) : %s", str);
27     }
28 }
29 else
30 {
31     char str[str_size];
32     sprintf(str, "hello world, I am process %d of %d\n", rank, size);
33     // Si invia al processo 0
34     MPI_Send(str, str_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
35 }
36
37 MPI_Finalize(); // Serve per terminare la libreria
38 return 0;
39 }
40 }
```

Quando un processo esegue una `MPI_Recv`, fra i vari messaggi, viene cercato quello di cui matchano il tag, il comunicatore, ed il mittente, lo scopo del `tag` è quello di essere un ulteriore separatore logico per la comunicazione. Anche i tipi dei messaggi devono combaciare, inoltre il numero di byte da ricevere deve essere maggiore o uguale al numero di byte inviati

$$\text{ByteRecv} \geq \text{ByteSent}$$

Nella chiamata `MPI_Recv`, i campi `source` e `tag` possono essere riempiti con, rispettivamente, `MPI_ANY_SOURCE` e `MPI_ANY_TAG` per non eseguire il controllo su mittente e tag nel ricevimento. È comunque possibile sapere qual'è il mittente, dato che tale informazione è salvata nel campo `MPI_Status`.

~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~

2.3 Design di Programmi Parallelvi

Data la specifica di un programma, quali sono le regole da seguire per partizionare il carico di lavoro fra i vari processi? Non esistono delle regole adatte ad ogni evenienza, ma è stata definita una metodologia largamente generica, la **Foster's methodology**.

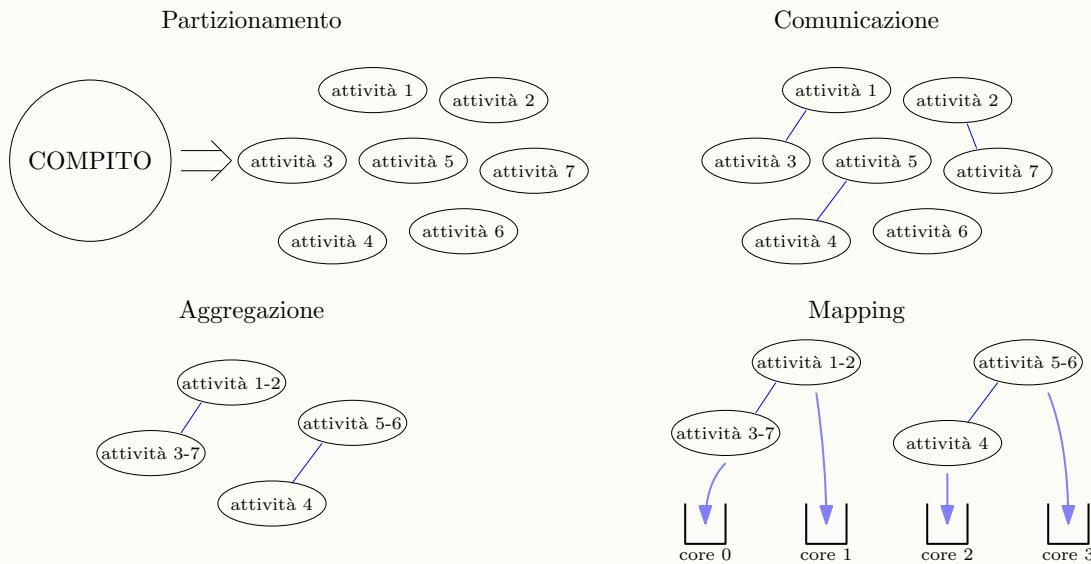


Figura 2.1: Foster's methodology

1. *Partizionamento* : si identificano delle attività di base indipendenti fra loro che possono essere eseguite in parallelo.
2. *Comunicazione* : determinare quali sono le attività stabilite nel punto precedente che per essere eseguite necessitano di uno scambio di messaggi.
3. *Aggregazione* : identificare le attività precedentemente stabilite che devono necessariamente essere eseguite in sequenza, ed aggregare in un'unica attività.
4. *Mapping* : assegnare ai vari processi le attività definite in precedenza in modo che il carico di lavoro sia uniformemente distribuito. Idealmente la comunicazione deve essere ridotta al minimo.

2.3.1 Pattern di Design Parallello

La struttura di un programma parallelo può essere definita secondo due pattern, si può dire che esistono due modi di *parallelizzare* un programma

- **GPLS (Globally Parallel, Locally Sequential)** : L'applicazione vede diversi task sequenziali venire eseguiti in parallelo.
- **GSLP (Globally Sequential, Locally Parallel)** : L'applicazione segue uno specifico "flusso" di esecuzione sequenziale, di cui alcune parti vengono eseguite in parallelo.

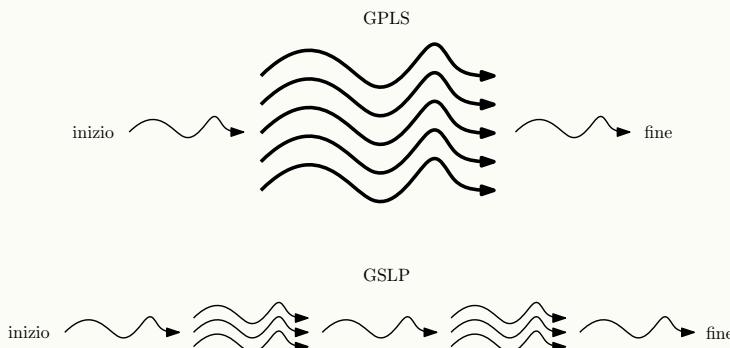


Figura 2.2: GPLS e GSLP

Esempi di GPLS

- **Single Program Multiple Data** : La logica dell'applicazione viene mantenuta in un unico eseguibile, tipicamente il programma segue la seguente struttura
 1. Inizializzazione del programma
 2. Ottenimento degli identificatori
 3. Esecuzione del programma in diverse ramificazioni in base ai core coinvolti
 4. Terminazione del programma
- **Multiple Program Multiple Data** : Quando la memoria da utilizzare è elevata è necessario suddividere il carico su più programmi, che spesso vengono eseguiti su differenti piattaforme.
- **Master-Worker** : Ogni processo può essere
 - Worker - Esegue la computazione
 - Master - Gestisce il carico di lavoro e lo assegna ai processi worker, colleziona i risultati ottenuti da questi ultimi e si occupa spesso delle operazioni di I/O o interazione con l'utente.
- **Map-Reduce** : Una versione modificata del paradigma Master-Worker, in cui i nodi worker eseguono due tipi di operazioni
 - Map : Esegue la computazione su un insieme di dati che risulta in un insieme di risultati parziali (ad esempio, esegue la somma su ogni elemento di un vettore)
 - Reduce : Colleziona i risultati parziali e ne deriva un risultato finale (ad esempio, somma tutti gli elementi di un vettore ottenendo un unico scalare)

Esempi di GSLP

- **Fork-Join** : C'è un unico "padre" in cui avviene l'esecuzione, quando necessario, tale padre potrebbe eseguire una `fork` generando dei nodi figli, che eseguono la computazione per poi terminare, facendo sì che il padre continui.
- **Loop-Parallelism** : Risulta estremamente semplice da utilizzare e viene spesso applicata quando un programma sequenziale deve essere adattato al multiprocesso. Consiste nel parallelizzare ogni esecuzione di un ciclo `for`, è necessario che le iterazioni però siano indipendenti fra loro.

```

1 //Esempio di Fork-Join
2 mergesort(A, lo , hi){
3     if lo < hi{
4         mid = lo + (hi-lo) / 2
5         fork mergesort(A, lo , mid)
6         mergesort(A, mid , hi)
7
8         join
9         merge(A, lo , mid , hi)
10    }
11 }
```

~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~

2.4 Comunicazione non Bloccante e Comunicazione Collettiva

Il contesto canonico di utilizzo di MPI è su un'insieme di server connessi fra loro (memoria privata), quando un processo esegue una `MPI_Send`, il buffer in cui è contenuto il messaggio viene copiato e salvato dalla memoria principale alla memoria dell'interfaccia di rete (NIC Memory), per poi venire trasferito attraverso la rete verso la memoria NIC del destinatario, da lì, verrà poi trasferita nella memoria principale di quest'ultimo.

L'utilizzo di una `MPI_Send` è quindi dispendioso dal punto di vista computazionale, in quanto sono coinvolte molteplici operazioni di scrittura e chiamate di sistema, è quindi buona regola, eseguire il minor numero di `MPI_Send` possibile

Ad esempio, è più conveniente eseguire una sola chiamata in cui si trasferiscono 200 byte piuttosto che due chiamate in cui si trasferiscono 100 byte ciascuna.

Si è detto in precedenza che `MPI_Send` è bloccante, in realtà, MPI utilizza, se non specificato diversamente, una metodologia di comunicazione standard, se il messaggio da trasferire è piccolo, è probabile che venga immediatamente trasferito venendo salvato su un buffer del destinatario. Diversamente, nel caso di un messaggio grande, la chiamata sarà bloccante in quanto MPI deve assicurarsi che il destinatario abbia allocato la memoria sufficiente per riceverlo.

In entrambi i casi, MPI si assicura che il messaggio da inviare non vada perso, il programma riottiene il controllo solo quando il buffer utilizzato per contenere il messaggio è di nuovo disponibile, si dice che la `MPI_Send` è *locally blocking*. Oltre la comunicazione standard, vi sono altri modi di inviare messaggi

- **Buffered** : Tramite la chiamata `MPI_Bsend`, l'operazione è sempre locally blocking, ma l'utente deve fornire manualmente un buffer in cui salvare il messaggio da inviare.
- **Sincrona** : Tramite la chiamata `MPI_Ssend`, l'operazione è globalmente bloccante, il controllo viene restituito esclusivamente quando il destinatario ha ricevuto il messaggio chiamando `MPI_Recv`. Risulta utile per far sì che un processo attenda che un altro arrivi ad un certo punto della computazione.
- **Ready** : Tramite la chiamata `MPI_Rsend`, se il destinatario non ha già effettuato una `MPI_Recv`, tale chiamata fallisce, è quindi necessario che esso sia già in attesa di ricevere.

2.4.1 Send e Recv Immediate

Le chiamate `MPI_Recv` e `MPI_Send` sono considerate poco performanti in quanto il processo chiamante potrebbe bloccare la sua esecuzione, in alcuni casi può essere utile una chiamata non bloccante per la trasmissione dei dati, soprattutto quando il mancato ricevimento di essi non causa errori nell'esecuzione del programma. Le funzioni non bloccanti messe a disposizione da MPI sono dette **funzioni immediate**, e permettono l'overlap fra computazione e comunicazione. Se al momento di una chiamata di ricevimento non ci sono dati da leggere, il programmatore dovrà gestire esplicitamente la situazione.

La chiamata `MPI_Isend` ha gli stessi parametri della funzione non immediata, eccetto un parametro aggiuntivo, `MPI_Request *req`, necessario per avere informazioni sullo status della chiamata.

La chiamata `MPI_Irecv` ha gli stessi parametri della funzione non immediata, eccetto per l'assenza del parametro sullo status originario, e l'aggiunta del parametro `MPI_Request *req`, necessario per avere informazioni sullo status della chiamata.

La funzione `int MPI_Wait(MPI_Request *request, MPI_Status *status)` fa sì che il processo si blocchi finché un invio o una ricezione non è andato a buon termine. È una chiamata bloccante.

La funzione `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)` controlla se una chiamata di invio o ricezione è andata o no a buon fine, salvando l'esito del risultato nel campo `flag`.

Esistono altre varianti di `Wait` e `Test`

- `Waitall`
- `Waitany`
- `Testany`
- etc...

2.4.2 Esempi di Applicazione

Il seguente esempio mostra un programma in cui n processi (in questo caso 4) si scambiano informazioni in una configurazione "ad anello", in cui ognuno invia e riceve a/da i suoi vicini, l'utilizzo di chiamate non bloccanti è utile per evitare situazioni di deadlock.

```

1 #include "mpi.h"
2 #include <stdio.h>
3 int main(void) {
4     int numtasks, rank, next, prev, buf[2];
5     MPI_Request reqs[4]; // variabili necessarie per le chiamate Irecv e Isend
6     MPI_Status stats[4]; // variabili necessarie per Waitall
7     MPI_Init(NULL, NULL);
8     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    // Determina vicino a sinistra e a destra
11    prev = (rank-1) % numtasks;
12    next = (rank+1) % numtasks;
13    // Operazioni di comunicazione
14    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
15    MPI_Irecv(&buf[1], 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
16    MPI_Isend(&rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[2]);
17    MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[3]);
18    // Qui puo' essere eseguita computazione nel mentre che gli altri processi comunicano
19    // Attende la fine delle operazioni non bloccanti
20    MPI_Waitall(4, reqs, stats);
21    MPI_Finalize();
22 }
```

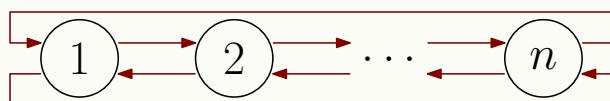


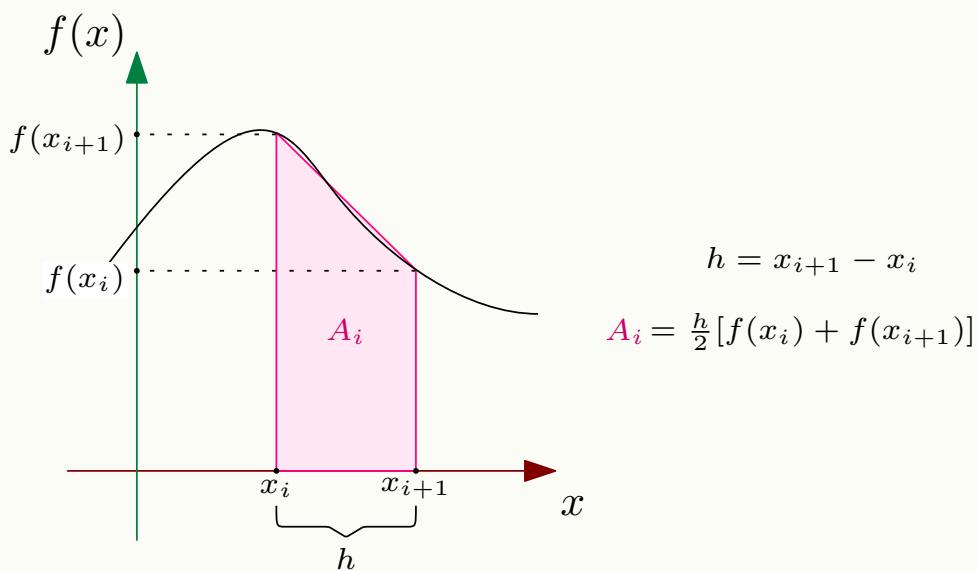
Figura 2.3: configurazione ad anello

Integrazione numerica

Si consideri adesso il seguente esempio, si vuole scrivere un programma che esegua l'integrazione numerica di una generica funzione $f(x)$ tramite la regola del trapezoidale. Tale metodo consiste nel dividere l'intervallo di integrazione in n intervalli

$$\{(x_0, x_1), (x_1, x_2), (x_2, x_3) \dots (x_{n-1}, x_n)\}$$

lunghi h , di cui verrà calcolata l'area approssimandola ad un trapezio.



L'integrale approssimato sarà la somma totale di tutti i trapezoidi

$$\frac{h}{2} \left[[f(x_1) + f(x_2)] + [f(x_2) + f(x_3)] + \dots + [f(x_{n-1}) + f(x_n)] \right]$$

```

1 /* Input : a, b, n */
2 h = (b-a)/n;
3 approx = (f(a)+f(b))/2;
4 for (i=1;i<=n-1;i++){
5     x_i = a+i*h;
6     approx += f(x_i);
7 }
8 approx=h*approx;

```

Se ne vuole dare un'implementazione parallela in cui i vari processi eseguiranno il calcolo di un trapezio, le somme parziali verranno inviate al processo di rank 0 che si occuperà di calcolare la somma totale (paradigma MAP REDUCE).

```

1 double Trap(double left, double right, int count, double base_len){
2
3     double esitmate, x;
4     // f e' la funzione integranda
5     esitmate = (f(left)+f(right))/2.0;
6     for(int i = 1; i<=count-1; i++){
7         x=left+i*base_len;
8         estimate+=f(x);
9     }
10    return estimate*base_len;
11 }

```

```

1 int main(void){
2
3     int my_rank;
4     int comm_size;
5     int n = 1024; //numero di intervalli , piu' e' grande , piu' la stima sara' precisa
6     double a = 0.0; //estremo sinistro di integrazione
7     double b = 3.0; //estremo destro di integrazione
8     double h; //lunghezza intervalli;
9     double local_a , local_b;
10    double local_sum;
11    double total_sum;
12    int source;
13
14    MPI_Init(NULL,NULL);
15    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
16    MPI_Comm_size(MPI_COMM_WORLD,&comm_size);
17
18    h=(b-a)/n;
19    local_n = n/comm_size; //numero di trapezoidi per ogni processo
20
21    local_a = a+my_rank*local_n*h;
22    local_b=local_a+local_n*h;
23    local_sum = Trap(local_a , local_b , local_n , h); //calcolo somma parziale
24
25    if(my_rank!=0){
26        //Invio la somma parziale al processo con rank 0
27        MPI_Send(&local_sum , 1 , MPI_DOUBLE, 0 , 0 , MPI_COMM_WORLD);
28    }
29    else{
30        total_sum = local_sum;
31        for(source = 1;source<comm_sz; source++){
32            MPI_Recv(&local_sum , 1 , MPI_DOUBLE, source , 0 ,
33                      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
34            total_sum+=local_sum
35        }
36
37        printf("con n = %d trapezoidi , la somma approssimata della funzione\n"
38               "da %f a %f e' %.15e .\n" , n , a , b , total_sum);
39    }
40
41    MPI_Finalize();
42    return 0;
43 }
```

2.4.3 Operazioni Collettive

Qual'è il problema con l'implementazione del trapezoide appena mostrata? Il processo di rank zero ha un carico di lavoro superiore rispetto ogni altro processo, infatti, quest'ultimo oltre la somma dei suoi trapezi locali, deve calcolare la somma totale, inoltre deve occuparsi di ricevere i dati da tutti gli altri processi.

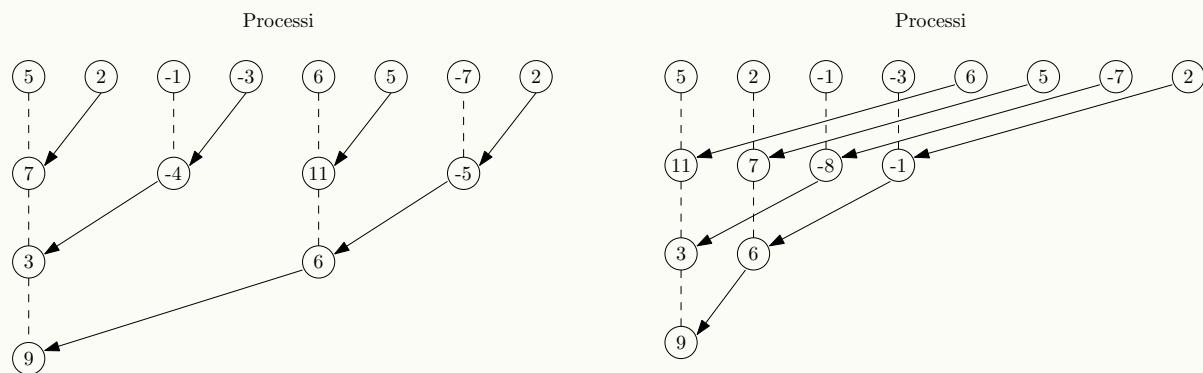


Figura 2.4: alberi differenti (entrambi validi)

Si può pensare di suddividere il carico di lavoro ad albero, come già visto in figura 1.1, facendo sì che il suo carico di lavoro sia logaritmico in funzione del numero dei rank. Nell'esempio visto, ogni processo condivide i suoi dati parziali con quello adiacente (da un punto di vista di numero di identificazione), ma nulla vieta agli ultimi processi di condividere i dati con i primi, come in figura 2.4. L'ottimalità di una soluzione piuttosto che di un'altra può dipendere da diversi fattori non sempre analizzabili, come la topologia fisica della rete attraverso cui sono collegate le macchine che eseguono i processi.

A tal proposito, MPI fornisce una funzionalità che permette di eseguire operazioni di aggregazione di risultati senza preoccuparci della logica di comunicazione per il trasferimento dei dati parziali. La funzione in questione è `int MPI_Reduce`, con i seguenti parametri

- `void* input_data_o` è il puntatore alla variabile in ingresso (somma parziale)
- `void* output_data_o` è il puntatore al valore che sarà riempito con il valore totale aggregato
- `int count` è il numero di elementi da aggregare
- `MPI_Datatype datatype` è il tipo dei valori in questione
- `MPI_Op operator` è l'operazione di aggregazione (somma, moltiplicazione, XOR, etc...)
- `int dest_process` è il rank del processo che riceverà il risultato
- `MPI_Comm comm` il comunicatore in questione

Le operazioni di aggregazione supportate da MPI sono le seguenti

Operazione	Significato
<code>MPI_MAX</code>	Massimo
<code>MPI_MIN</code>	Minimo
<code>MPI_SUM</code>	Somma
<code>MPI_PROD</code>	Prodotto
<code>MPI_BAND</code>	AND logico
<code>MPI_BAND</code>	AND bit a bit
<code>MPI_LOR</code>	OR logico
<code>MPI_BOR</code>	OR bit a bit
<code>MPI_LXOR</code>	XOR logico
<code>MPI_BXOR</code>	XOR bit a bit
<code>MPI_MAXLOC</code>	Massimo insieme al suo indice
<code>MPI_MINLOC</code>	Minimo insieme al suo indice

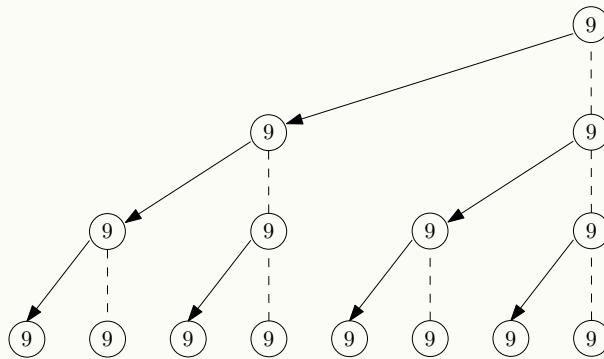
È possibile anche definire delle operazioni personalizzate tramite la chiamata `MPI_Op_create`.

Quando viene chiamata una funzione collettiva, è importante che ogni processo del comunicatore la chiama, altrimenti l'esecuzione rimane bloccata in uno stato di attesa, dato che ogni processo attende che tutti gli altri siano arrivati a tale operazione. Ovviamente, tutti i processi che eseguono un'operazione di questo tipo devono definire lo stesso processo che riceverà l'output. Tutti i processi escluso quello di destinazione, nel campo `void* output_data_o` possono specificare qualsiasi valore.

Non essendo presente alcun tag, nella comunicazione collettiva, le operazioni verranno matchate in base all'ordine di esecuzione. Nell'esempio del trapezoide 2.4.2 vi è un problema, nel caso si volesse decidere arbitrariamente l'intervallo di integrazione, o il numero di trapezi, il processo di rank 0 dovrà occuparsi di leggere i dati da stdin, per poi condividerli ad ogni altro processo.

Si ricordi che in MPI, esclusivamente il processo di rank 0 può interagire con lo stdin.

È chiaro che su di esso sia riportato un carico di lavoro maggiore, è quindi possibile suddividere il carico facendo sì che il processo 0 condivida i dati con due altri processi, e questi due li condividano a loro volta con altri due processi ciascuno, creando un albero di condivisione.



Anche in questo caso, la logica con la quale scambiarsi le informazioni può variare, e quella ottimale può dipendere dalle condizioni della rete ed altri fattori difficilmente analizzabili, per questo MPI fornisce una funzione, `MPI_Bcast` che si occupa di eseguire il broadcast da un processo verso tutti gli altri. I parametri sono i seguenti :

- `void* data_p` è il puntatore alla variabile da condividere
- `int count` è il numero di elementi da condividere
- `MPI_Datatype datatype` è il tipo dei valori in questione
- `int source_process` è il rank del processo che condivide il valore
- `MPI_Comm comm` il comunicatore in questione

Il parametro `void* data_p` fungerà sia da input, che da output, nel caso il processo chiamante sia colui che condivide il valore, in `data_p` sarà presente il valore condiviso, altrimenti, in `data_p` sarà presente il valore ricevuto.

Esempio di funzione per leggere input da tastiera

```

1 void Get_input(int my_rank, int a, int b, int n){
2     if(my_rank==0){
3         printf("enter input:\n");
4         scanf("%d %d %d", &a, &b, &n);
5     }
6     MPI_Bcast(a, 1, MPI_INT, 0, MPI_COMM_WORLD);
7     MPI_Bcast(b, 1, MPI_INT, 0, MPI_COMM_WORLD);
8     MPI_Bcast(n, 1, MPI_INT, 0, MPI_COMM_WORLD);
9 }
```

A questo punto, si supponga di voler fare un'operazione di aggregato, per poi avere il risultato condiviso fra tutti i processi, concettualmente, ciò equivale ad eseguire una `MPI_Reduce` seguita da una `MPI_Bcast`. MPI fornisce una funzione a tal proposito, ottimizzata a dovere, ossia `MPI_Allreduce`, con i seguenti parametri

- `void* input_data_o` è il puntatore alla variabile in ingresso (somma parziale)
- `void* output_data_o` è il puntatore al valore che sarà riempito con il valore totale aggregato
- `int count` è il numero di elementi da aggregare
- `MPI_Datatype datatype` è il tipo dei valori in questione
- `MPI_Op operator` è l'operazione di aggregazione (somma, moltiplicazione, XOR, etc...)
- `MPI_Comm comm` il comunicatore in questione

I parametri sono identici alla `MPI_Reduce`, eccetto per l'assenza del processo di destinazione, dato che in questo caso, ogni processo avrà il risultato.

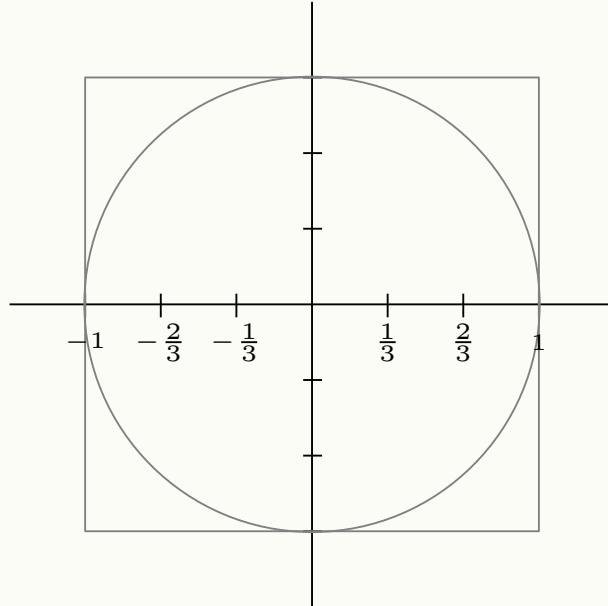
Le operazioni collettive sono diventate particolarmente importanti nell'ultimo periodo in quanto sono utilizzate nella stragrande maggioranza dei programmi paralleli che vengono eseguiti per il training delle reti neurali odierne. Le grosse aziende di informatica, hanno iniziato a produrre delle proprie librerie proprietarie

- NCCL (*Nvidia*)
- RCCL (*AMD*)
- OneCCL (*Intel*)
- MSCCL (*Microsoft*)

MPI durante le operazioni aggregate utilizza delle euristiche per stimare quale sia il miglior modo di condividere i dati fra i nodi, è possibile forzare tale decisione attraverso delle opportune variabili d'ambiente. Il punto è che MPI non è consapevole dell'hardware sul quale i processi sono eseguiti, per questo le aziende hanno iniziato a produrre librerie proprietarie, appositamente ottimizzate per girare sulle piattaforme dedicate.

Stima del π

Si vuole scrivere un programma che tramite il metodo di Montecarlo calcoli il valore stimato di π distribuendo il lavoro su più processi tramite MPI. L'algoritmo utilizzato per il calcolo è semplice, si consideri un cerchio di raggio unitario, inscritto in un quadrato 2×2 .



L'area del cerchio, è uguale a $\pi r^2 \implies \pi$, l'area del quadrato è 4. Sia A_c l'insieme di tutti i punti compresi nel cerchio, e sia A_q l'insieme di tutti i punti compresi nel quadrato. Risulta che il numero di punti nell'area del cerchio stanno all'area π , come il numero di punti che stanno nell'area del quadrato stanno a 4.

$$\frac{|A_c|}{\pi} = \frac{|A_q|}{4}$$

In realtà, non ha senso considerare la cardinalità di $|A_c|$ o di $|A_q|$, in quanto sono insiemi infiniti, supponiamo allora che tali insiemi siano finiti e di cardinalità n , si denotano A_c^n e A_q^n , chiaramente $A_c^n \subseteq A_q^n$. Si ha che

$$\lim_{n \rightarrow \infty} 4 \cdot \frac{|A_c^n|}{|A_q^n|} = \pi$$

L'algoritmo consiste nel calcolare un numero n di punti casuali, sia c il numero di punti interni al cerchio, ossia i punti (x, y) tali da rispettare $x^2 + y^2 \leq 1$. Numericamente, $4 \frac{n}{c}$ approssimerà π , con una precisione sempre maggiore all'aumentare di n .

L'algoritmo si renderà parallelo, distribuendo equamente il numero di punti da calcolare a tutti i processi presenti.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include <time.h>
5
6 int main( int argc , char **argv )
7 {
8
9     int precision = 1000; // Numero di punti generati casualmente
10
11    if ( argc > 1 )
12    {
13        precision = atoi(argv[1]);
14    }
15
16    MPI_Init(NULL, NULL);
17    srand(time(NULL));
18
19    int my_rank;
20    int my_size;
21    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
22    MPI_Comm_size(MPI_COMM_WORLD, &my_size);
23
24    int local_precision = precision / my_size; /* Numero di punti da generare per
25                                              ogni processo */
26    int local_circle_point = 0;
27
28    for ( int i = 0; i <= local_precision; i++ )
29    {
30        double x = (double)rand() / RAND_MAX * 2.0 - 1.0; // Generazione punto
31        double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
32        if ( x * x + y * y <= 1 ) // Controllo se il punto e' nel cerchio
33            local_circle_point++;
34    }
35
36    int total_circle_point = 0;
37    MPI_Reduce(&local_circle_point , &total_circle_point , 1, MPI_INT, MPI_SUM,
38                           0, MPI_COMM_WORLD);
39
40    if ( my_rank == 0 )
41    {
42        double esteem = ((double)total_circle_point / precision * 4);
43        printf("Su %d precision , la stima del pi greco e' : %lf\n",
44                           precision , esteem );
45    }
46
47    MPI_Finalize();
48    return 0;
49 }
```

Risultati della Computazione

Numero punti generati	Valore π stimato
100	4.08
1000	3.408
10000	3.1031
100000	3.13416
1000000	3.143736
100000000	3.141725

~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~

2.5 Valutazione del Tempo

Valutare il tempo di esecuzione di un programma multicore non è banale. MPI fornisce una funzione `double MPI_Wtime`, ritorna un valore che rappresenta il tempo passato da un certo riferimento fisso. Basta valutare questo tempo in due punti diversi del codice e farne la differenza.

```

1 double start , finish ;
2 start=MPI_Wtime() ;
3 /*codice*/
4 finish=MPI_Wtime() ;
5 printf ("%lf" , finish - start ) ;

```

Ogni processo, seguirà un'evoluzione dello stesso codice differente, e non contemporanea fra gli altri. Se verrà calcolato il tempo trascorso per l'esecuzione di una sezione di codice, ogni processo restituirà un tempo diverso. Il tempo totale del programma, sarà dato dal massimo dei tempi forniti da ogni processo. Si può usare l'operazione collettiva

```

1 double local_start , local_finish , local_elapsed , elapsed ;
2 local_start=MPI_Wtime() ;
3 /*codice*/
4 local_finish=MPI_Wtime() ;
5 local_elapsed=local_finish-local_start ;
6 MPI_Reduce(&local_elapsed ,&elapsed ,1 ,MPI_DOUBLE,MPI_MAX,0 ,comm ) ;
7 printf ("%d" , elapsed ) ;

```

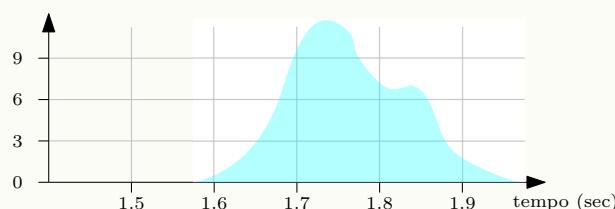
Ovviamente, tale metodo è valido con l'assunzione che tutti i processi inizino nello stesso momento (in particolare, la valutazione del tempo di inizio), un'esecuzione reale però è sfasata, e ciò porterebbe ad un tempo di esecuzione che non corrisponde a quello effettivo.

Quando si esegue un'operazione di reduce, tutti i processi devono arrivare ad uno punto comune nel codice, "sincronizzandosi", esiste un'operazione, `MPI_Barrier`, il cui unico scopo è attendere che tutti i processi la esegano prima di continuare nell'esecuzione. Tale operazione comunque, non garantisce che i processi si sincronizzino una volta eseguita, quindi approssima il comportamento sincronizzato. Nella pratica, è molto complesso garantire tale proprietà.

La misurazione del tempo impiegato da un processo è non deterministica, in quanto quest'ultimo è soggetto alle interruzioni del sistema operativo ed ai cambi di contesto, che possono variare in maniera apparentemente aleatoria. Nel caso in cui il programma venga eseguito in rete, tale *rumore* (il tempo casuale aggiunto all'esecuzione normale di un processo) è ancora maggiore.

Generalmente, è corretto eseguire più volte un processo misurando i tempi ad ogni esecuzione, per avere una misura statistica. Il minimo corrisponderà al caso ideale, il massimo al caso peggiore, la mediana al caso più frequente, è corretto fornire i dati di ogni esecuzione per avere un quadro chiaro dei tempi effettivi. In breve per valutare il tempo

1. Si mette una funzione barriera all'inizio dell'esecuzione
2. Si trova il massimo dei tempi di ogni processo
3. Si provano diverse esecuzioni ottenendo una distribuzione



Le interferenze ed i rumori sono stati grande oggetto di studio nella valutazione dei tempi, in particolare, si è osservato un rapporto di proporzionalità diretta fra il rumore ed il numero di processi impiegati in un calcolo, esiste quindi un limite, in cui l'aumento dei nodi volti ad un calcolo non garantisce un miglioramento dei tempi di esecuzione, bensì il contrario.

La seguente tabella, raccoglie i tempi di esecuzione in secondi di un algoritmo che esegue il prodotto fra matrici quadrate.

tempo (sec)	ordine della matrice				
	1024	2048	4096	8192	16384
1	4.1	16	64	270	1100
2	2.3	8.5	33	140	560
4	2	5.1	18	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

Ovviamente, con l'aumento delle dimensioni dell'input, aumenta anche il tempo di esecuzione, e quest'ultimo decresce con l'aumentare del numero di processi. Questo fattore di proporzionalità inversa non è però valido per un arbitrario numero di processi, de facto, ci sarà un limite per cui vale, ad un certo punto, l'aumentare dei processi non migliorerà il tempo di esecuzione. Per la stima dei tempi si definiscono le seguenti variabili

- $T_s(n)$ è il tempo di esecuzione di un programma se eseguito in maniera sequenziale, con un input di dimensione n .
- $T_p(n, p)$ è il tempo di esecuzione di un programma se eseguito in maniera parallela con p processi, e con un input di dimensione n .
- $S(n, p) = \frac{T_s(n)}{T_p(n, p)}$ è detto **speed up** dell'applicazione e misura il margine di miglioramento di un programma quando si esegue in parallelo piuttosto che in sequenziale.

Ovviamente nel rapporto dello speed up i tempi sequenziali e paralleli vanno misurati sullo stesso hardware. L'ideale sarebbe uno speed up lineare (del tipo, $S(n, p) \simeq p$), nell'effettivo, l'aumentare dei processi fa diminuire lo speed up, invece l'aumentare dell'input lo fa aumentare. La seguente tabella riporta i valori dello speed up del programma che esegue il prodotto fra matrici.

speed up	ordine della matrice				
	1024	2048	4096	8192	16384
1	1	1	1	1	1
2	1.8	1.9	1.9	1.9	2
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Nota bene : $T_p(n, 1) \neq T_s(n)$, generalmente il tempo di esecuzione parallela con 1 processo è maggiore del tempo di esecuzione sequenziale (dato che il setup dell'ambiente per il calcolo parallelo ha un costo).

La **scalabilità** di un processo è definita come segue

$$Sc(n, p) = \frac{T_p(n, 1)}{T_p(n, p)}$$

L'**efficienza** invece è un valore compreso fra zero ed 1 definito come segue

$$E(n, p) = \frac{S(n, p)}{p} \in (0, 1]$$

efficienza	ordine della matrice				
	1024	2048	4096	8192	16384
1	1	1	1	1	1
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.3	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Chiaramente, se le dimensioni dell'input sono modeste, l'utilizzo di tanti processi risulta inutile, l'efficienza è quindi bassa. L'utilizzo del multi processo ha senso quando l'input è di grande dimensione, e si presta ad essere diviso e computato da nodi paralleli.

2.5.1 Scalabilità Forte e Scalabilità Debole

Un programma si dice **strongly scalable** (fortemente scalabile) se, data una dimensione fissa dell'input n , lo speed up ha una buona crescita all'aumentare dei processi. È invece **weakly scalable** (debolmente scalabile) se, l'aumentare dell'input, e l'aumentare del numero dei processi, il tempo di esecuzione varia di poco.

Rendere un'applicazione strongly scalable è estremamente difficile, per questo nell'effettivo viene considerato sempre il weak scaling quando si vuole parallelizzare un compito. È possibile avere una stima approssimativa di quanto un'applicazione possa scalare? Dato un programma, si definisce **frazione seriale** la porzione di esso che è impossibile da parallelizzare, e viene espressa come un valore fra 0 ed 1. Tale parte non potrà essere influenzata dalla parallelizzazione, la **legge di Amdahl** stabilisce che lo scaling di un'applicazione è limitato dalla frazione seriale. Per un n dimensione in input fissata, si ha

$$T_p(p) = (1 - \alpha)T_s + \alpha \frac{T_s}{p}$$

Dove $\alpha \in [0, 1]$ rappresenta la frazione parallelizzabile, e $1 - \alpha$ la frazione seriale. Lo speed up sarà quindi

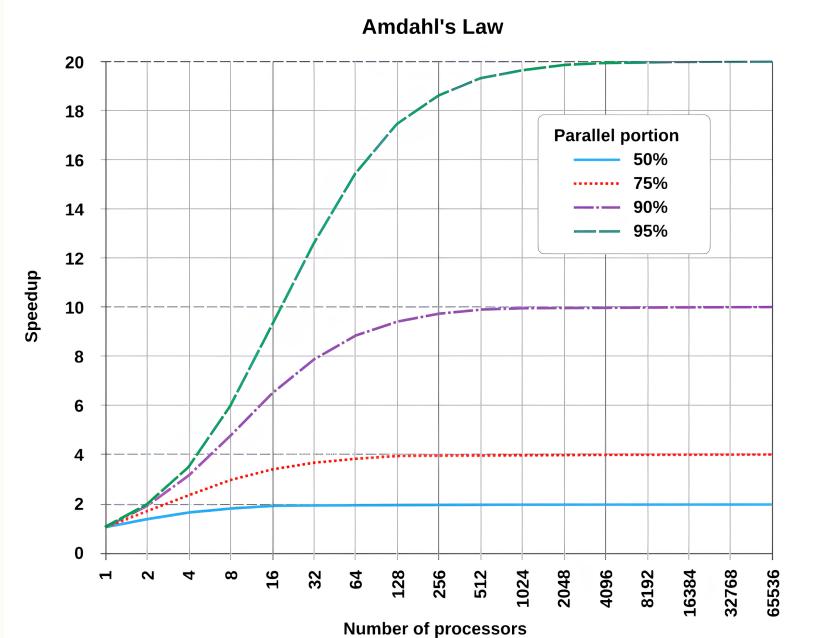
$$S(p) = T_s \frac{1}{(1 - \alpha)T_s + \alpha \frac{T_s}{p}}$$

Il valore dello speed up è limitato superiormente, infatti all'aumentare dei processi :

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{1 - \alpha} \quad (2.1)$$

$$\lim_{p \rightarrow \infty} T_s \frac{1}{(1 - \alpha)T_s + \alpha \frac{T_s}{p}} = \frac{1}{1 - \alpha} \quad (2.2)$$

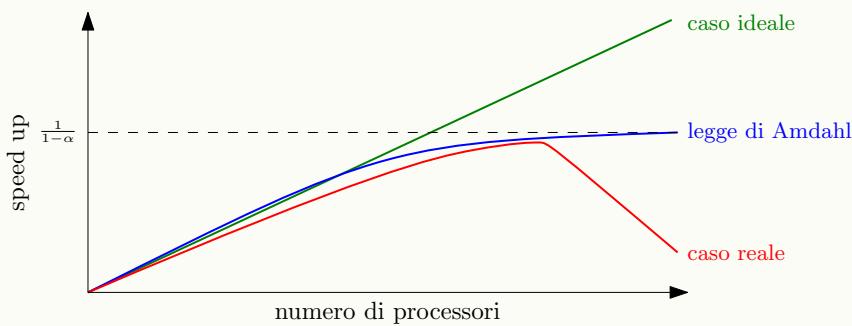
$$\frac{T_s}{(1 - \alpha)T_s} = \frac{1}{1 - \alpha} \quad (2.3)$$



Questa legge non tiene conto del weak scaling e decreta un comportamento ideale/approssimato. La **legge di Gustafson** tiene conto del weak scaling

$$S(n, p) = (1 - \alpha) + \alpha p$$

Nel caso reale, l'aumentare del numero di processi potrebbe far incrementare la frazione seriale.



2.6 Operazioni su Vettori e Matrici

2.6.1 Scatter e Gather

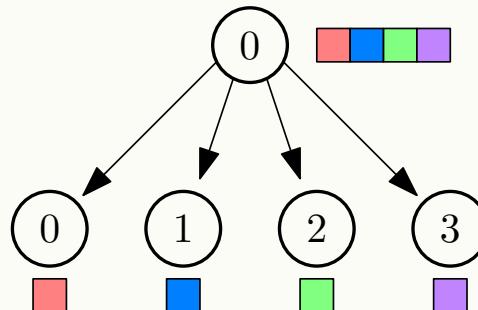
Consideriamo un algoritmo sequenziale che calcoli la somma di due vettori dati in input

```

1 void vector_sum(double x[], double y[], double z[], int n){
2     for(int i=0; i<n; i++){
3         z[i]=x[i]+y[i];
4     }
5 }
```

Come si può distribuire tale compito su diversi processi? Si fa l'assunzione che i vettori da sommare siano stati letti da standard input, quindi sono presenti nella memoria del processo con rank 0. Si vuole dividere il vettore equamente fra i diversi processi.

A tal proposito, esiste la funzione collettiva `int MPI_Scatter`, che si occupa di prendere un vettore e di dividerlo equamente fra tutti i processi di un dato comunicatore.

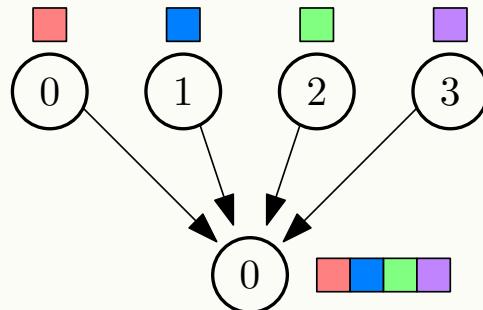


I parametri della funzione sono i seguenti

- `void *send_buf_p` il buffer contenente il vettore da dividere
- `int send_count` il numero di elementi da inviare ad ogni processo, non il numero di elementi totali del vettore
- `MPI_Datatype send_type` il tipo degli elementi del vettore
- `void *recv_buf_p` il buffer che conterrà la frazione di vettore ricevuta, per il mittente, può coincidere con il buffer di invio
- `int recv_count` analogo a `send_count`
- `MPI_Datatype recv_type` analogo a `send_type`
- `int src_proc` il rank del processo che condividerà il vettore
- `MPI_Comm comm` il comunicatore in questione

La funzione assume che il numero di elementi da condividere sia divisibile per il numero di processi, l'ordine di condivisione avviene in base al rank. Il nodo che condivide il vettore può specificare la macro `MPI_IN_PLACE` come buffer di destinazione, e manterrà i valori nel vettore originale.

La funzione collettiva `int MPI_Gather` è l'inversa della `scatter`, ogni nodo definisce un vettore, e ne verrà eseguita la concatenazione, per poi essere inviata ad un processo.



L'ordine di concatenazione è sempre dato dal rank. I parametri sono pressoché identici alla `scatter`

- `void *send_buf_p` il buffer contenente il vettore parziale da inviare
- `int send_count` il numero di elementi contenuti nel vettore parziale
- `MPI_Datatype send_type` il tipo degli elementi del vettore
- `void *recv_buf_p` il buffer che conterrà il vettore finale composto dalla concatenazione dei vettori parziali
- `int recv_count` analogo a `send_count`
- `MPI_Datatype recv_type` analogo a `send_type`
- `int dest_proc` il rank del processo che riceverà il vettore
- `MPI_Comm comm` il comunicatore in questione

Esempio di codice : Il seguente programma, fa sì che il processo di rank 0 generi due vettori con valori casuali, che verranno divisi fra i vari processi che eseguiranno delle somme parziali, trovando sezioni parziali del vettore somma, che verrà poi ri-unito con l'operazione collettiva.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <mpi.h>
5
6 void print_vector(int n, int *v, char name)
7 {
8     printf("%c = [ ", name);
9     for (int i = 0; i < n - 1; i++)
10    {
11        printf("%d, ", v[i]);
12    }
13    printf("%d ]\n", v[n - 1]);
14 }
15
16 int main(int argc, char **argv)
17 {
18     MPI_Init(NULL, NULL);
19
20     int my_rank;
21     int size;
22     int local_size = 3;
23     MPI_Comm_size(MPI_COMM_WORLD, &size);
24
25     int local_vector[size];
26     int local_vector2[size];
  
```

```

27 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
28
29 if (my_rank == 0)
30 {
31
32     int inputVector[local_size * size];
33     int inputVector2[local_size * size];
34     srand(time(NULL));
35     for (int i = 0; i < local_size * size; i++)
36     {
37         inputVector[i] = rand() % 100;
38         inputVector2[i] = rand() % 100;
39     }
40     print_vector(local_size * size, inputVector, 'A');
41     print_vector(local_size * size, inputVector2, 'B');
42     printf("\n");
43     MPI_Scatter(inputVector, local_size, MPI_INT, local_vector, local_size,
44                 MPI_INT, 0, MPI_COMM_WORLD);
45     MPI_Scatter(inputVector2, local_size, MPI_INT, local_vector2, local_size,
46                 MPI_INT, 0, MPI_COMM_WORLD);
47 }
48 else
49 {
50     MPI_Scatter(NULL, local_size, MPI_INT, local_vector, local_size,
51                 MPI_INT, 0, MPI_COMM_WORLD);
52     MPI_Scatter(NULL, local_size, MPI_INT, local_vector2, local_size,
53                 MPI_INT, 0, MPI_COMM_WORLD);
54 }
55 printf("process %d :\n      A%d = [ ", my_rank, my_rank);
56 for (int i = 0; i < local_size - 1; i++)
57 {
58     printf("%d, ", local_vector[i]);
59 }
60 printf("%d ]\n      B%d = [ ", local_vector[local_size - 1], my_rank);
61 for (int i = 0; i < local_size - 1; i++)
62 {
63     printf("%d, ", local_vector2[i]);
64 }
65 printf("%d ]\n      A%d+B%d = [ ", local_vector2[local_size - 1], my_rank, my_rank);
66 for (int i = 0; i < local_size - 1; i++)
67 {
68     printf("%d, ", local_vector2[i] + local_vector[i]);
69 }
70 printf("%d ]\n\n", local_vector2[local_size - 1] + local_vector[local_size - 1]);
71
72 int vec_sum[local_size];
73 for (int i = 0; i < local_size; i++)
74 {
75     vec_sum[i] = local_vector2[i] + local_vector[i];
76 }
77
78 if (my_rank == 0)
79 {
80     int outVector[local_size * size];
81     MPI_Gather(vec_sum, local_size, MPI_INT, outVector, local_size,
82                 MPI_INT, 0, MPI_COMM_WORLD);
83     print_vector(local_size * size, outVector, 'O');
84 }
85 else
86 {
87     MPI_Gather(vec_sum, local_size, MPI_INT, NULL, local_size,
88                 MPI_INT, 0, MPI_COMM_WORLD);
89 }
90
91 MPI_Finalize();
92
93 exit(0);
94 }
```

Le operazioni collettive sui vettori possono essere eseguite anche sulle matrici, anche se ciò dipende dall'allocazione di esse. Quando si dichiara una matrice di dimensioni statiche nel seguente modo

```
int matrix[n][n];
```

essa viene allocata in maniera contigua in memoria.

```
int matrix[3][3];
```

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

viene allocata :

0,0	0,1	0,2	1,0	1,1	1,2	2,0	2,1	2,2
-----	-----	-----	-----	-----	-----	-----	-----	-----

In tal caso, le operazioni collettive di *reduce* funzioneranno senza problemi dato che agiscono su blocchi di memoria contigui. Il punto è che una dichiarazione di questo tipo ha senso esclusivamente se è nota a priori la dimensione della matrice, nel caso più generale, una matrice si dichiara dinamicamente come un puntatore di puntatori di interi.

```
1 int** a;
2 a = (int**) malloc(sizeof(int*)*num_rows);
3 for(int i = 0; i < num_rows; i++){
4     a[i] = (int*) malloc(sizeof(int)*num_cols);
5 }
```

Le righe della matrice in questo caso *non* sono contigue in memoria, non è quindi possibile utilizzare operazioni di *reduce*, per far ciò, bisogna collassare la matrice in un array unidimensionale regolando l'accesso degli indici. Su una matrice statica, l'accesso avviene per mezzo di due indici `matrix[i][j]`, se la matrice è collassata in un array, un solo indice deve codificare entrambi gli indici della matrice, nella pratica, si fa così

```
matrix[i*num_col+j]
```

Una matrice allocata dinamicamente come un unico array può essere soggetta alle operazioni di *reduce*.

M =	<table border="1"> <tr><td>a</td><td>b</td><td>c</td></tr> <tr><td>d</td><td style="background-color: pink;">e</td><td>f</td></tr> <tr><td>g</td><td>h</td><td>i</td></tr> </table>	a	b	c	d	e	f	g	h	i	M' =	<table border="1"> <tr><td>a</td><td>b</td><td>c</td><td>d</td><td style="background-color: pink;">e</td><td>f</td><td>g</td><td>h</td><td>i</td></tr> </table>	a	b	c	d	e	f	g	h	i
a	b	c																			
d	e	f																			
g	h	i																			
a	b	c	d	e	f	g	h	i													

$$M[1][1] = e = M'[1*3+1] = M'[4]$$

Prodotto fra Vettori e Matrici

Il prodotto scalare fra due vettori è definito come segue

$$\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n$$

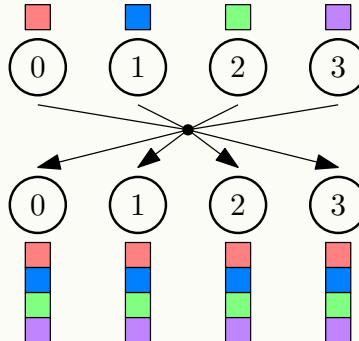
Il prodotto fra una matrice $n \times n$ ed un vettore \bar{b} di n componenti, è un vettore sempre di n componenti, di cui ogni i -esimo componente è il prodotto scalare fra la i -esima riga della matrice ed il vettore \bar{b} .

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + \dots + a_{1n}b_n \\ a_{21}b_1 + a_{22}b_2 + \dots + a_{2n}b_n \\ \vdots \\ a_{n1}b_1 + a_{n2}b_2 + \dots + a_{nn}b_n \end{bmatrix}$$

```
1 int A[n][n]; //matrice
2 int b[n]; //vettore input
3 int y[n]; //vettore risultato
4 for(i=0;i<n; i++){
5     for(j=0;j<n; j++){
6         y[i]=A[i][j]*b[j] //prodotto scalare implicitamente definito
7     }
8 }
```

2.6.2 Ultime Collettive di tipo "All"

La collettiva `int MPI_Allgather` si occupa di unire i dati di un vettore come la normale funzione di `gather`, per poi condividere i dati con tutti i processi di un comunicatore, logicamente, equivale ad una `gather` seguita da una `broadcast`, ma è implementata in maniera più efficiente.



- `void *send_buf_p`
- `int send_count`
- `MPI_Datatype send_type`
- `int recv_count`
- `MPI_Datatype recv_type`
- `MPI_Comm comm`

La collettiva `MPI_Allscatter` equivale ad una reduce seguita da una scatter.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \text{Allscatter} \Rightarrow \begin{bmatrix} a_1 + b_1 + c_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ a_2 + b_2 + c_2 \\ \vdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ a_n + b_n + c_n \end{bmatrix}$$

~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~

2.7 Tipi di Dato Custom

Abbiamo visto come MPI implementi il tipo `MPI_Datatype` per riferirsi ai tipi di dato presenti nel C. In C, è però possibile creare dei tipi di dato aggiuntivi (struct).

```

1 struct t{
2     float a,
3     float b,
4     int n
5 };

```

Cosa va definito nel campo `data_type` quando si intende inviare tramite MPI una struttura? MPI permette di creare dei tipi custom, è possibile definire **tipi derivati**, che equivalgano (in quanto disposizione in memoria) alla struct in questione. Un tipo derivato non è altro che una sequenza di datatipe di base alla quale è assegnata una certa disposizione in memoria.

`(MPI_DOUBLE,0), (MPI_DOUBLE,16),(MPI_INT,24)`

Un tipo derivato va definito tramite la funzione `int MPI_Type_create_struct`:

- `int count` il numero di elementi distinti nel tipo derivato

- `int array_of_blocklengths[]` ogni elemento potrebbe essere un vettore, va quindi specificata la lunghezza di ogni elemento della struct.
- `MPI_Aint array_of_displacements` la disposizione degli elementi della struct in memoria
- `MPI_Datatype *new_type_p` il tipo degli elementi della nuova struttura

Per ottenere la disposizione è possibile usare il puntatore del valore, è però auspicabile utilizzare la funzione di libreria `MPI_Get_address`, in quanto in alcuni sistemi (seppur pochi) i puntatori potrebbero non rappresentare un indirizzo di memoria.

```

1 MPI_Aint a_addr, b_addr, n_addr;
2 MPI_Get_address(&a, &a_addr);
3 array_of_displacements[0]=0;
4 MPI_Get_address(&b, &b_addr);
5 array_of_displacements[1]=b_addr-a_addr;
6 MPI_Get_address(&n, &n_addr);
7 array_of_displacements[2]=n_addr-a_addr;
```

Una volta creata tale struttura, bisogna eseguire la funzione `MPI_Type_commit` che, eseguendo opportune ottimizzazioni, decide il metodo di salvataggio della struttura in memoria. Quando non è più necessario quel datatype, si può chiamare la funzione `MPI_Type_free`.

CAPITOLO

3

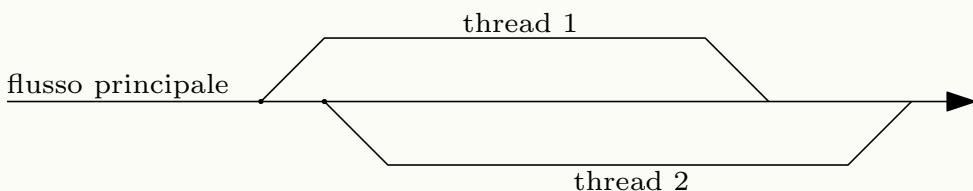
MEMORIA CONDIVISA : POSIX THREADS

3.1 Introduzione ai Thread

In MPI, un insieme di nodi comunica tramite lo scambio di messaggi su un apposito canale, all'interno di ogni nodo, sono presenti delle unità di calcolo con più cores, ossia più sotto-unità che comunicano ed operano contemporaneamente su una memoria condivisa.

Se MPI si occupa della comunicazione fra più nodi, è necessario un paradigma volto alla comunicazione dei differenti core di un'unità di calcolo che possono accedere ad una memoria comune.

Un processo è un'istanza di un programma (in esecuzione) che viene caricata sulla CPU, possiamo identificare i *singoli flussi indipendenti di computazione* di un processo, denominati **thread**, analoghi ai processi ma più "leggeri" nella loro creazione e gestione, due thread appartenenti ad uno stesso processo hanno differenti program counter e stack ma condividono lo stesso spazio di indirizzamento dinamico (heap), la comunicazione è semplice, in quanto basta scrivere sulla stessa area di memoria.



Lo standard **Posix Threads (Pthreads)** è implementato con una libreria in C e specifica l'API per l'interazione fra i thread, è necessario specificare nel codice la direttiva di inclusione

```
# include <pthread.h>
```

La libreria fornisce il tipo di dato `pthread_t`, non è importante la sua struttura interna, basti sapere che tale tipo identifica univocamente un thread all'interno di un processo, a volte viene denotato *thread handler*. La creazione di un thread avviene tramite la chiamata di libreria `int pthread_create`, i cui parametri sono

- `pthread_t* thread_p` verrà assegnato l'identificatore del thread creato.
- `const pthread_attr_t* attr_p` attributi in input che per adesso verranno ignorati.

- `void* (*start_routine) (void*)` la funzione che verrà eseguita dal thread creato.
- `void* arg_p` i parametri della funzione sopracitata.

All'avvio di un thread è quindi necessario specificarne la funzione da eseguire, una funzione può essere passata in input in quanto rappresenta un puntatore alla zona di memoria in cui è contenuta la procedura da eseguire.

```

1 void func(int a){
2     /*codice*/
3 }
4
5 void(*func_ptr)(int)=func; /*puntatore a funzione in cui se ne specifica anche il tipo
6 [int] del parametro*/

```

Convenzionalmente, la funzione del thread è di tipo puntatore a void e ha come parametro di ingresso un puntatore a void (caso più generale possibile) che verrà eventualmente castato.

```
void *thread_function(void *args_p);
```

Per attendere la terminazione di un thread da parte del processo principale è necessario utilizzare la funzione `pthread_join`, valida per un singolo thread, i parametri sono

- `pthread_t thread` l'identificatore del thread di cui attendere la terminazione
- `void **value_ptr` il valore di ritorno del thread

La funzione `pthread_self` ritorna l'identificatore del thread chiamante. Il seguente esempio mostra la creazione di diversi thread che stampano a schermo una stringa.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #define THREAD_COUNT 4
5
6 void *Hello(void *rank)
7 {
8     int my_rank = rank;
9     printf("hello from thread %d\n", my_rank);
10    return NULL;
11 }
12
13 int main()
14 {
15     pthread_t thread_handles[THREAD_COUNT];
16     for (int i = 0; i < THREAD_COUNT; i++) /*creazione dei thread*/
17         pthread_create(&thread_handles[i], NULL, Hello, (void *)i);
18     printf("hello from main thread\n");
19     for (int i = 0; i < THREAD_COUNT; i++) /*terminazione dei thread*/
20         pthread_join(thread_handles[i], NULL);
21     return 0;
22 }
```

Riguardo il numero dei thread, sarebbe corretto considerare tanti thread quanti sono i core fisici sulla macchina che esegue il programma. Se si vogliono passare *molti* parametri ad un thread è necessario definire un'apposita struct che li comprenda.

3.1.1 Prodotto Matrice-Vettore con Pthread

Si vuole eseguire il classico algoritmo di moltiplicazione di una matrice per un vettore, la cui computazione seriale si ricorda essere

```

1 /* m := righe di A*/
2 /* n := elementi di x*/
3 /* y := vettore output*/
4 for(int i = 0; i < m; i++){
5     y[i] = 0.0;
6     for(int j = 0; j < n; j++){
7         y[i] += A[i][j] * x[j];
8     }
9 }
```

Si vuole parallelizzare l'esecuzione fra più thread, come prima cosa, si supponga di avere tanti thread quante sono le righe della matrice in input (quindi, gli elementi del vettore in output), e di far calcolare ad ogni thread i il valore dell' i -esimo elemento del vettore in output.

$$\begin{vmatrix} a_{11} & \dots & \dots & a_{1n} \\ \vdots & \ddots & & \vdots \\ a_{m1} & \dots & \dots & a_{mn} \end{vmatrix} \times \begin{vmatrix} x_1 \\ \vdots \\ x_n \end{vmatrix} = \begin{vmatrix} y_1 & \text{thread 1} \\ \vdots & \vdots \\ y_m & \text{thread } m \end{vmatrix}$$

In generale, se il numero di thread è $t < m$, ogni q -esimo thread computerà $\frac{m}{t}$ righe, ossia

- dalla $q \cdot \frac{m}{t}$ -esima riga
- alla $(q + 1)\frac{m}{t} - 1$ -esima riga

Si ha il seguente codice

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 #define MAT_ORDER 6
6 #define THREAD_COUNT 6
7
8 /*Siano A la matrice, x il vettore in input ed y il vettore in output*/
9
10 void *mat_vec_product(void *rank)
11 {
12     int my_rank = rank;
13     int local_m = MAT_ORDER / THREAD_COUNT;
14     int first_row = my_rank * local_m;
15     int last_row = ((my_rank + 1) * local_m) - 1;
16
17     for (int i = first_row; i <= last_row; i++)
18     {
19         y[i] = 0;
20
21         for (int j = 0; j < MAT_ORDER; j++)
22         {
23             y[i] += A[i][j] * x[j];
24         }
25     }
26 }
27
28 int main()
29 {
30     pthread_t handler[THREAD_COUNT];
31     for (int i = 0; i < THREAD_COUNT; i++)
32         pthread_create(&handler[i], NULL, mat_vec_product, (void *)i);
33     for (int i = 0; i < THREAD_COUNT; i++)
34         pthread_join(handler[i], NULL);
35     return 0;
36 }
```



3.2 Sezioni Critiche

Si consideri il seguente scenario : Vi è una variabile (intera) x inizializzata a 0 condivisa da due thread, entrambi si adoperano per incrementare la variabile di 1, ci si aspetta, che dopo l'esecuzione la variabile x sia uguale a 2. Il punto è che l'incremento di una variabile (somma) in C, equivale ad una sequenza di istruzioni assembly che possono essere interfogliate dai due thread.

istanti di tempo	thread 0	thread 1
1	inizializzato dal main thread	
2	legge la variabile <code>x=0</code> e la carica nel registro <code>r1</code>	inizializzato dal main thread
3	incrementa il registro <code>r1=0</code> di 1, avendo <code>r1=1</code>	legge la variabile <code>x=0</code> e la carica nel registro <code>r2</code>
4	salva il valore del registro in <code>x</code> , si ha <code>x=1</code>	incrementa il registro <code>r2=0</code> di 1, avendo <code>r2=1</code>
5	termina	salva il valore del registro in <code>x</code> , si ha <code>x=1</code>
6		termina

Alla fine dell'esecuzione la variabile `x` sarà uguale ad 1 perché i due thread non si sono sincronizzati correttamente. Tale problema è noto come **race condition** e ha comportato la nascita di una vasta teoria riguardante la sincronizzazione dei processi paralleli che accedono a risorse condivise, in questa sezione verranno presentate delle possibili soluzioni.

3.2.1 Busy Waiting e Mutex

Il tema centrale è la mutua esclusività delle variabili, l'accesso ad esse non può venire contemporaneamente da parte di più thread, con **busy waiting**, si denota una metodologia che consiste nel far attendere un processo che tenta di accedere ad una variabile già in uso, lasciandolo bloccato in un ciclo in cui ad ogni iterazione controlla la disponibilità della risorsa.

Nel seguente esempio, ogni thread viene bloccato in un ciclo finché la variabile `flag` non diventa uguale al loro rank. A seguito dell'utilizzo dovranno incrementare la variabile per permettere al prossimo thread di accedere.

```
1 while( flag!=my_rank );
2 x+=1;
3 flag++;
```

Il busy waiting, seppur semplice nella sua implementazione, comporta svantaggi difficilmente trascurabili

- un processo in attesa occuperà inutilmente la CPU comportando uno spreco di computazione
- Il compilatore, se nota un'indipendenza apparente fra due istruzioni, potrebbe invertirne l'ordine, ad esempio, trasformando il codice come segue

```
1 x+=1;
2 while( flag!=my_rank); /* sequenza di istruzioni errata */
3 flag++;
```

Tale riarrangiamento è parte di una serie di ottimizzazioni che fa il compilatore. Disattivare tali ottimizzazioni comporterebbe un uso meno efficiente dei registri per tutte le altre istruzioni.

Le **Mutex** permettono una gestione più sofisticata degli accessi alle risorse mutualmente esclusive, una Mutex è una variabile utilizzata per restringere l'accesso ad una risorsa, per poi rilasciarla una volta utilizzata.

Una mutex viene gestita dal tipo di dato `pthread_mutex_t`, una volta definita, va inizializzata tramite la funzione `int pthread_mutex_init`, assegnandole la variabile il cui accesso deve essere regolamentato. La funzione ha i seguenti parametri

- `pthread_mutex_t *mutex_p` la mutex a cui si riferisce
- `const pthread_mutexattr_t *attr_p` degli attributi che per adesso saranno ignorati.

Quando si vuole accedere ad una variabile protetta da una mutex bisogna controllarne lo stato, che può essere *bloccato* o *libero*. Si richiede l'accesso alla variabile tramite la chiamata

```
int pthread_mutex_lock(pthread_mutex_t *mutex_p)
```

Se l'accesso è libero, il thread continuerà la sua esecuzione, altrimenti verrà *sospeso* (senza occupare tempo sulla CPU), e potrà continuare la sua esecuzione solo quando la lock sarà rilasciata.

Una volta che un thread ha terminato l'accesso alla variabile, deve rilasciare la lock con la chiamata

```
int pthread_mutex_unlock(pthread_mutex_t *mutex_p)
```

Una volta terminata l'utilità di una mutex, va eliminata con la chiamata

```
int pthread_mutex_destroy(pthread_mutex_t *mutex_p)
```

Il fenomeno della **Starvation** si verifica quando l'esecuzione di un thread o di un processo viene sospesa o impedita per un tempo indefinito, anche se è in grado di continuare l'esecuzione. È tipicamente associata all'imposizione di priorità o alla mancanza di equità nella pianificazione o nell'accesso alle risorse. Se una mutex è bloccata, il thread viene bloccato e inserito in una coda di thread in attesa. Se la coda è FIFO, non si verificherà starvation.

Un **Deadlock** invece identifica una situazione di attesa circolare in cui ogni elemento di un insieme di thread è sia in attesa del rilascio di una lock, sia in possesso di una lock necessaria agli altri thread.

<pre> 1 pthread_mutex_lock(&a) ; 2 pthread_mutex_unlock(&b) ; 3 /*in attesa di a per 4 rilasciare b*/ </pre>	<pre> 1 pthread_mutex_lock(&b) ; 2 pthread_mutex_unlock(&a) ; 3 /*in attesa di b per 4 rilasciare a*/ </pre>
----------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------

In termini di prestazioni il Mutex mostra performance più elevate rispetto il busy waiting quando il numero di thread è superiore al numero di core fisici, si osservi la seguente tabella che riporta i tempi impiegati (in secondi) per stimare il valore di π su una macchina con 8 core.

num. thread	busy waiting	mutex
1	2.9	2.9
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.5	0.38
32	0.8	0.4
64	3.56	0.38

3.2.2 Semafori, Barriere e Variabili di Condizione

Il busy waiting permette l'accesso esclusivo ai thread imponendo anche un certo ordine, ma presenta degli svantaggi considerevoli, che però non presenta l'utilizzo delle mutex, seppur quest'ultimo lascia l'ordine di accesso indeterminato e deciso dal sistema operativo, ci sono situazioni in cui si vuole un accesso ordinato senza dover subire i problemi del busy waiting.

Lo standard Posix definisce un costrutto simile al mutex ma più elaborato : i **Semafori**, che rappresentano delle mutex *non binarie*. Per utilizzarli, si include la direttiva

```
# include <semaphore.h>
```

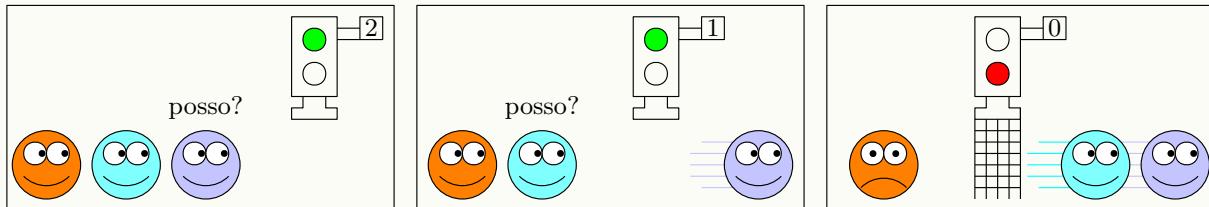
L'handler dei semafori è il tipo `sem_t`, un semaforo viene inizializzato con `int sem_init` i cui parametri sono

- `sem_t* semaphore_p`
- `int shared` se uguale ad 1, sarà condiviso anche fra diversi processi
- `unsigned initial_val` il numero intero con il quale è inizializzato un semaforo

Il funzionamento è il seguente, quando un thread vuole accedere ad una sezione, chiama la funzione `sem_wait(sem_t *sem)`, ad ogni semaforo è associato un numero intero, se tale numero è maggiore di zero al momento della chiamata, il processo potrà procedere, e decrementerà di 1 il valore del semaforo.

Se il valore è uguale a zero, il processo si arresterà entrando in una coda di attesa, attendendo il via libera per ripartire.

Quando un thread termina con l'utilizzo del semaforo, può chiamare la funzione `sem_post(sem_t * sem)` che farà procedere un thread in attesa del semaforo nell'esecuzione. Se la coda è vuota, il valore del semaforo verrà incrementato di 1.



Con **barriera** si intende un costrutto che viene inserito in un certo punto del codice, in cui ogni thread che lo esegue, per poter continuare l'esecuzione oltre tale punto deve attendere che ogni altro thread vi sia arrivato, serve a ri-sincronizzare l'esecuzione dei thread facendoli ripartire da un punto comune. Un tipico esempio può essere quello di voler far noto in fase di debug che tutti thread abbiano raggiunto un certo punto del codice.

```

1  /*punto comune del programma*/
2  barriera;
3  if (my_rank==0){
4      printf("tutti i thread hanno raggiunto questo punto \n");
5      fflush(stdout);
6  }

```

L'implementazione di una barriera tramite l'uso del busy waiting e delle mutex è intuitiva, si utilizza un contatore condiviso inizializzato a zero (il cui accesso è protetto da mutex) e si incrementa di 1 per ogni thread che arriva a tal punto. Ogni thread sarà in busy waiting finché il contatore non sarà uguale al numero dei thread.

```

1  int counter = 0;
2  int thread_number;
3  pthread_mutex barrier_mutex;
4
5  void *thread_function(void *arg_p){
6      ...
7      /*implementazione della barriera*/
8      pthread_mutex_lock(&barrier_mutex);
9      counter++;
10     pthread_mutex_unlock(&barrier_mutex);
11     while(counter<thread_number);
12     /*fine della barriera*/
13     ...
14 }

```

Una barriera può anche essere implementata con un semaforo. L'implementazione mostrata presenta un problema, se fosse necessario riutilizzare tale barriera, bisognerebbe resettare la variabile `counter` a zero. Tale reset però deve avvenire solo *in seguito* al passaggio di tutti i thread della barriera, potrebbe capitare che un thread non si sia reso conto che la variabile `counter` avesse raggiunto il valore sufficiente per proseguire, rimanendo bloccato nella barriera. Questo problema, nonostante possa sembrare banale, è in realtà un intralcio considerevole e non ha soluzioni immediate.

Può essere di aiuto l'uso di una **variabile di condizione**, queste ultime sono un *oggetto* associato ad una mutex, che permettono ad un thread di sospendere la sua esecuzione fino all'accadere di un determinato *evento*, si definiscono con il tipo `pthread_cond_t`. A seguito di tale evento, il thread può essere svegliato con un apposito segnale. Una variabile di condizione descrive un *evento*.

```

1  lock mutex;
2  if CONDIZIONE SI AVVERA : /* avvisa i thread che la condizione si e' avverata*/

```

```

3     signal thread(s);
4 else : /*se non e' verificata, si libera la lock e si attende la condizione*/
5     unlock mutex
6     SOSPENSIONE ESECUZIONE
7     unlock mutex;
8

```

In seguito, l'implementazione di una barriera con delle variabili di condizione.

```

1 int counter = 0; /*variabile condivisa*/
2 pthread_mutex_t mutex;
3 pthread_cond_t cond_var;
4 ...
5
6 void *thread_function( void *args_p){
7     ...
8     /*barriera*/
9     pthread_mutex_lock(&mutex);
10    counter++;
11    if(counter==thread_count){
12        counter=0;
13        pthread_cond_broadcast(&cond_var); /*avvisa tutti i thread che
14                                     la condizione si e' avverata*/
15    } else{
16        while(pthread_cond_wait(&cond_var,&mutex)!=0);
17    }
18    pthread_mutex_unlock(&mutex);
19 }

```

Si osservi la riga di codice 16

```
while(pthread_cond_wait(&cond_var,&mutex)!=0);
```

La chiamata `pthread_cond_wait` sospende il processo, perché allora è necessario inserirla all'interno di un while come se andasse verificato periodicamente? È buona norma dato che il sistema operativo potrebbe inaspettatamente svegliare un processo dallo stato di attesa prima che la condizione si avveri, queste *sveglie spurie* vanno quindi gestite.

- `pthread_cond_signal(pthread_cond_t* cond_var_p);` sveglia un thread in attesa
- `pthread_cond_broadcast(pthread_cond_t* cond_var_p);` sveglia tutti i thread in attesa
- `pthread_cond_init(pthread_cond_t* cond_p,pthread_condattr_t* cond_attr_p)` inizializza una variabile di condizione
- `pthread_cond_destroy(pthread_cond_t* cond_p)` termina una variabile di condizione
- La funzione

```
pthread_cond_wait(pthread_cond_t* cond_var_p, pthread_mutex_t mutex_p)
```

si occupa di

1. fare l'unlock della mutex
2. sospendere il thread finché non arriva un segnale di risveglio
3. fare la lock della mutex una volta svegliato il thread

3.2.3 Stima di π con Pthread

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <time.h>
5
6 unsigned precision = 100000; /* punti generati da ogni thread*/
7 const unsigned thread_number = 6;
8
9 unsigned total_tosses;
10 unsigned point_in_center = 0;
11
12 /* gestione accesso a variabili condivise */
13 pthread_mutex_t mutex;
14
15 void *thread_function(void *arg_p)
16 {
17     int local_circle_point = 0;
18
19     for (int i = 0; i <= precision; i++)
20     {
21         double x = (double)rand() / RAND_MAX * 2.0 - 1.0; /* Generazione punto casuale */
22         double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
23         local_circle_point += (x * x + y * y < 1); /* Controllo se e' nel cerchio */
24     }
25
26     /* L'accesso alla variabile condivisa deve essere mutualmente esclusivo */
27     pthread_mutex_lock(&mutex);
28     point_in_center += local_circle_point;
29     pthread_mutex_unlock(&mutex);
30 }
31
32 int main(int argc, char **argv)
33 {
34     srand(time(NULL));
35
36     if (argc > 1)
37     {
38         precision = atoi(argv[1]);
39     }
40
41     total_tosses = precision * thread_number;
42     pthread_mutex_init(&mutex, NULL);
43     pthread_t tids[thread_number];
44
45     for (int i = 0; i < thread_number; i++)
46     {
47         pthread_create(&tids[i], NULL, thread_function, NULL);
48     }
49
50     for (int i = 0; i < thread_number; i++)
51     {
52         pthread_join(tids[i], NULL);
53     }
54
55     double esteem = ((double)point_in_center / (double)(total_tosses)) * 4;
56     printf("valore di pi greco stimato : %lf\n", esteem);
57
58     return 0;
59 }
```

Si consideri adesso la seguente porzione di codice

```

1 void *fun( void *args){
2     int thread_id=*((int*)args);
3     printf("i am thread %d\n", thread_id);
4     return NULL;
5 }
6
7 int main(){
8     ...
9     for( int i = 0; i<num_threads; i++){
10         pthread_create(&thread_handles[ i ] , NULL, fun , (void*)&i );
11     }
12     ...
13 }
```

C'è un errore, essendo la variabile `i` condivisa da tutti i thread, potrebbe succedere che l'esecuzione della funzione `fun` avvenga parallelamente fra tutti i thread in seguito alla terminazione del ciclo `for`. In tal caso, ogni thread annuncerebbe di essere il $n - 1$ -esimo (dove n è il numero di thread).

Per ovviare a tale problema, è opportuno creare un array di interi in modo che ogni thread acceda ad una porzione di memoria diversa.

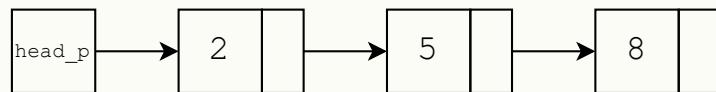
```

1 int main(){
2     ...
3     int *ids=(int*)malloc( num_threads* sizeof(int) );
4     for( int i = 0; i<num_threads; i++){
5         ids[ i]=i;
6         pthread_create(&thread_handles[ i ] , NULL, fun , (void*)&ids[ i ] );
7     }
8     ...
9 }
```

~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~ ~*~

3.3 Read-Write Lock

Questa sezione tratterà l'accesso a strutture dati condivise di grandi dimensioni, si consideri una lista puntata di interi, ordinata in maniera crescente, soggetta ad operazioni di inserimento, eliminazione, e controllo della presenza di un elemento (`member`).



```

1 struct list_node_s{
2     int data;
3     struct list_node_s* next;
4 }
```

Essendo la lista definita con i puntatori, ogni accesso ad uno dei suoi elementi sarà un accesso in memoria principale, non essendo questi contigui, il prefetching non potrà fare affidamento sulle ipotesi di vicinanza spaziale e temporale.

Si vuole mantenere una lista puntata *condivisa in memoria* fra i diversi thread, in tal caso è comodo definire la testa della lista `head_p` come una variabile globale. Questo semplificherà le operazioni sulla lista in quanto non sarà necessario passarla come parametro.

Se due o più thread eseguono un'operazione di `member` contemporaneamente, il programma funziona senza problemi, ma cosa succederebbe se contemporaneamente, un thread eseguisse un'operazione di `member` ed un altro un'operazione di modifica? L'accesso alla lista potrebbe non essere consistente.

	thread 1	thread 2
$t = 1$	chiama <code>member(8)</code>	
$t = 2$		chiama <code>delete(8)</code>
$t = 3$		esegue <code>delete(8)</code>
$t = 4$	esegue <code>member(8)</code>	
$t = 5$	non trova l'elemento 8 anche se nel momento della chiamata era presente	

Idea di Soluzione 1

Una soluzione immediata ricade nell'utilizzo di una lock, è possibile eseguire il lock dell'intera lista prima di accedervi in un qualsiasi modo (lettura o scrittura).

```

1 Pthread_mutex_lock(&list);
2 member(value);
3 Pthread_mutex_unlock(&list);

```

Nonostante funzioni, nell'effettivo si sta serializzando l'accesso alla lista: non ci saranno mai due thread che accedono contemporaneamente alla lista (anche su elementi diversi, anche per operazioni di `member`), quindi l'utilizzo del multithreading risulta inutile e deleterio per le prestazioni del programma.

Idea di Soluzione 2

Un approccio più raffinato consiste nel, piuttosto che fare il lock sull'intera lista, farlo sui singoli elementi (ridefinendo la struttura del nodo).

```

1 struct list_node_s{
2     int data;
3     struct list_node_s* next;
4     pthread_mutex_t mutex;
5 }

```

Questo approccio è più complesso ed appesantisce notevolmente la lista, in quanto ogni nodo occuperà più byte in memoria, inoltre, ad ogni singolo accesso ci sarà un `lock` seguito da un `unlock`, rallentando significativamente le operazioni, gestire tali sequenze di accessi non è semplice e si può facilmente incombere in deadlock.

Nessuna delle soluzioni presentate è ottimale, e sfrutta al meglio i vantaggi del multithreading per l'accesso alle strutture dati condivise. A tal proposito, Pthreads definisce una metodologia di accesso favorevole alla situazione.

Definizione : Un **read-write lock** è una struttura simile ad una mutex, che mette a disposizione *due funzioni* di lock.

La prima, esegue il lock di una variabile in lettura, la seconda, esegue il lock in scrittura, ciò, permette a più thread contemporaneamente di ottenere il lock in lettura di una variabile, finché non c'è alcun nodo che richiede una lock in scrittura.

- più thread possono accedere contemporaneamente in lettura
- solo un thread alla volta può accedere in scrittura

Se un thread ha eseguito il lock in lettura di una variabile, ogni altro thread potrà accedervi in lettura, ma un qualsiasi accesso in scrittura dovrà attendere il rilascio della lock.

Se un thread ha eseguito il lock in scrittura di una variabile, ogni altro thread dovrà attendere il rilascio della lock per accedervi in un qualsiasi modo.

Il tipo di dato fornito da Pthreads è

`pthread_rwlock_t`

Come le altre funzioni, gode di una funzione `init` per l'inizializzazione ed una funzione `destroy` per lo svuotamento della memoria.

valore sulla variabile	azione richiesta	risultato
libera	lock in lettura	ottiene il lock in lettura
ha un lock in lettura	lock in lettura	ottiene il lock in lettura
ha un lock in scrittura	lock in lettura	attende il rilascio della variabile
libera	lock in scrittura	ottiene il lock in scrittura
ha un lock in lettura	lock in scrittura	attende il rilascio della variabile
ha un lock in scrittura	lock in scrittura	attende il rilascio della variabile

Il problema della `rwlock`, è che la prioritizzazione dei lock in lettura può causare starvation dei thread che provano ad accedere in scrittura. Quando si esegue una lock su una variabile va specificato se è in lettura o scrittura, le funzioni fornite per il lock e l'unlock sono

- `pthread_rwlock_rdlock(&rlock)` ;
- `pthread_rwlock_wrlock(&rlock)` ;
- `pthread_rwlock_unlock(&rlock)` ;

Si consideri il seguente esempio riguardante 100.000 operazioni eseguite su una lista puntata, di cui il 99% delle operazioni è in sola lettura `member`, mentre il restante 1% riguarda `insert` e `delete`.

	numero di thread			
	1	2	4	8
read-write lock	0.213	0.123	0.098	0.115
lock sull'intera lista	0.211	0.45	0.385	0.457
lock sui singoli nodi	1.68	5.7	3.45	2.7

tempo misurato in secondi

Si osservi la prima riga, relativa alle esecuzioni con 1 thread, è chiaro che l'utilizzo delle rwlock non comporti alcun vantaggio rispetto il lock sull'intera lista, mentre l'esecuzione con i lock sui singoli nodi risulta lenta dato l'overhead nel chiamare le funzioni di lock ad ogni accesso.

	numero di thread			
	1	2	4	8
read-write lock	0.213	0.123	0.098	0.115
lock sull'intera lista	0.211	0.45	0.385	0.457
lock sui singoli nodi	1.68	5.7	3.45	2.7

tempo misurato in secondi

Si osservi ora l'ultima riga, il costo aumenta in quanto ci sono più thread che tentano l'accesso ai nodi. Le operazioni sulla lista sono relativamente semplici e veloci, quindi l'utilizzo delle lock ad ogni accesso su ogni singolo nodo non è giustificato e l'overhead "uccide" i tempi del programma, rendendolo inefficiente.

	numero di thread			
	1	2	4	8
read-write lock	0.213	0.123	0.098	0.115
lock sull'intera lista	0.211	0.45	0.385	0.457
lock sui singoli nodi	1.68	5.7	3.45	2.7

tempo misurato in secondi

Si osservi come le rwlock garantiscono uno speed up notevole rispetto l'implementazione con la lock sull'intera lista, si ricorda che in questo contesto la maggior parte delle operazioni sono di `member`, è quindi immediato il fatto che le rwlock siano vantaggiose, in quanto la maggior parte delle lock, essendo in lettura, non bloccano gli altri thread.

Si consideri ora un altro esempio, sempre riguardante 100.000 operazioni eseguite su una lista puntata, ma questa volta, l'80% delle operazioni è in sola lettura `member`, mentre le operazioni di `insert` e `delete` coprono il 20% restante. Essendo più frequenti i lock in scrittura, ci si aspetta un peggioramento dello speed up per il programma che fa uso di `rwlock` rispetto il caso precedente.

	numero di thread			
	1	2	4	8
read-write lock	2.48	4.97	4.69	4.71
lock sull'intera lista	2.5	5.13	5.04	5.11
lock sui singoli nodi	12	29.6	17	12

tempo misurato in secondi

Le read-write lock garantiscono un vantaggio nelle performance finché le operazioni di scrittura sono una parte relativamente piccola del totale delle operazioni da eseguire sulla struttura dati.

3.4 Funzioni Thread-Safe

Definizione : Un blocco di codice si dice **thread safe** se può essere acceduto contemporaneamente da più thread senza che l'accesso simultaneo violi la correttezza del programma.

Si consideri il seguente esempio, si vuole "tokenizzare" un file contenente parole inglesi, ossia dividere il file in unità più piccole, chiamate "token", che hanno un significato linguistico o grammaticale.

- **Tokenizzare un file:** analizzare il testo contenuto nel file e identificare tutti i token che lo compongono.

È possibile suddividere il file in righe ed assegnare ad ogni thread un insieme di righe da tokenizzare, l'assegnamento può avvenire in maniera circolare, sia n il numero di thread

- al thread 1 si assegna la riga 1
 - al thread 2 si assegna la riga 2
 - \vdots
 - al thread n si assegna la riga n
 - al thread 1 si assegna la riga $n + 1$
 - al thread 2 si assegna la riga $n + 2$
 - ecc..

È possibile serializzare l'accesso alle righe dell'input tramite i semafori, questi ultimi sono preferibili rispetto le semplici mutex dato che l'accesso alle linee deve seguire uno specifico ordine. Una volta che un thread è in possesso della riga che gli appartiene, può chiamare la funzione di libreria `strtok`.

```
1 char *strtok(
2     char* string,
3     const char* separators );
```

La funzione `strtok` ha un funzionamento insolito

1. la prima volta che viene chiamata, va passata in input la stringa da tokenizzare ed il separatore.
 2. va poi richiamata ogni volta (passando NULL come separatore) per ottenere i token successivi

La funzione mantiene uno *stato interno* sul puntatore del separatore corrente.

Esempio di utilizzo : Si chiama `strtok` con separatore " " (spazio vuoto) su input

live and learn

- Prima chiamata `strtok(string, " ")` ritorna "live".
- Seconda chiamata `strtok(string, NULL)` ritorna "and".
- Terza chiamata `strtok(string, NULL)` ritorna "learn".

Ad ogni chiamata utilizza lo stato del puntatore per ottenere il token successivo. La funzione **non è thread safe**, lo stato che mantiene è globale (il puntatore è condiviso e non privato) e tutti i thread che la chiamano operano sullo stesso puntatore, se più thread la chiamano simultaneamente, potrebbe non funzionare correttamente.

Esistono altre funzioni della libreria standard del C che non sono thread safe

- `random` in `stdlib.h`
- `localtime` in `time.h`

In alcuni casi, il C fornisce delle versioni alternative di alcune funzioni dette *rientranti*, che risultano thread safe, ad esempio `strtok_r`.

Definizione : Una funzione **rientrante** in C è una funzione che può essere interrotta in qualsiasi momento della sua esecuzione e richiamata nuovamente, anche da più thread contemporaneamente, senza causare comportamenti imprevisti o corruzione dei dati (lo stato interno non viene manomesso).

♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

3.5 Combinazione di Thread ed MPI

È possibile utilizzare il multithreading anche in un programma che utilizza `MPI`, se viene eseguita una `MPI_Send`, il programmatore richiede la garanzia che tale funzione sia chiamata contemporaneamente su ogni thread, a tal proposito, è necessario inizializzare la sezione di codice che utilizza MPI con la chiamata

```
MPI_Init_thread(int *argc, char **argv, int required, int *provided)
```

Il parametro `required` deve specificare uno fra i 4 possibili **threading level**, ossia una modalità di gestione del multithreading su MPI, in particolare

- `MPI_THREAD_SINGLE`: i processi non possono usare i thread, è come se fosse stata chiamata una normale `MPI_Init`.
- `MPI_THREAD_FUNNELED`: I processi possono essere multi-thread, ma solo il thread principale può effettuare chiamate MPI. Questo livello è utile quando si vogliono utilizzare le direttive OpenMP per parallelizzare le computazioni all'interno di un processo MPI.
- `MPI_THREAD_SERIALIZED`: I processi possono essere multithread, ma solo un thread alla volta può effettuare chiamate MPI. Le chiamate MPI quindi, devono essere serializzate, anche se ci sono più thread. Questo richiede una sincronizzazione tra i thread per evitare conflitti.
- `MPI_THREAD_MULTIPLE`: Questo è il livello più flessibile, ma anche il più complesso. Permette a tutti i thread di effettuare chiamate MPI in modo concorrente. Tuttavia, comporta un overhead maggiore e richiede una cura particolare nella sincronizzazione.

Non tutti i livelli di threading sono supportati da tutte le implementazioni di MPI. Ad esempio, alcune implementazioni potrebbero non supportare `MPI_THREAD_MULTIPLE`.

3.5.1 Thread Pinning

Con *thread pinning*, si identifica la costrizione da imporre ad un thread sul venire eseguito in uno specifico core. In alcuni casi, un sistema operativo potrebbe schedulare tutti i thread su un unico core fisico, in tal caso sono definite apposite chiamate nella libreria `sched.h`.

```
1 #define _GNU_SOURCE /* definizione necessaria per il pinning */
2 #include <pthread.h>
3 #include <sched.h>
4
5 void* thread_func(void *args){
6     cpu_set_t cpuset; /* bitmask per l'assegnazione*/
7     pthread_t thread = pthread_self();
8     CPU_ZERO(&cpuset); /* imposta a zero tutti i bit */
9     CPU_SET(3, &cpuset); /* imposta ad 1 il terzo bit */
10    /* si assegna la bitmask al thread, verrà schedulato solo sul terzo core.*/
11    s = pthread_setaffinity_np(thread, sizeof(cpu_set), &cpuset);
12 }
```

CAPITOLO

4

RICHIAMO DI ARCHITETTURE

Questo capitolo (decisamente ridotto rispetto agli altri) richiama alcuni concetti base delle architetture degli elaboratori, espandendo il campo alle CPU multicore.

4.1 Caching

La *Cache* è una memoria aggiuntiva posta "in vicinanza" del processore in modo che gli accessi ad essa siano più rapidi rispetto gli accessi alla memoria principale. Tipicamente, la cache è saldata sullo stesso chip della CPU, ed è costruita su una tecnologia più rapida (SRAM) rispetto la memoria principale (DRAM), è de facto anche più costosa e ha molti più transistor, quindi le sue dimensioni sono ridotte.

La cache si basa sul principio di località spaziale e temporale, è un principio empirico, riguarda l'assunzione che

- Se si accede ad un'area di memoria, nel futuro prossimo si accederà ad aree di memoria adiacenti
- Se si accede ad un'area di memoria, nel futuro prossimo si accederà nuovamente ad essa

Quando si accede ad un blocco di memoria quest'ultimo (ed eventualmente qualche blocco a lui adiacente) viene trasferito nella cache, per il principio di località, un accesso prossimo ridurrà notevolmente i tempi in quanto verrà eseguito sulla cache piuttosto che sulla memoria principale.

L'accesso in cache richiede un tempo di alcuni ordini di grandezza inferiore rispetto l'accesso in memoria principale

Esempio: In un array `z=malloc(sizeof(int)*16)`, quando si accede all'elemento `z[0]`, anche gli altri elementi fino a `z[15]` saranno trasferiti nella cache.

4.1.1 Livelli della Cache

Spesso su un chip non vi è un'unica cache, ma diverse (solitamente 3), e si identificano quindi cache di diverso livello. La cache più "vicina" alla CPU è più veloce negli accessi, ma ha anche meno spazio a disposizione, la cache più lontana invece è la più capiente, ma anche più lenta.

Quando bisogna accedere alla memoria, la CPU controlla prima la cache più vicina, se i dati richiesti non sono presenti, passa a quella successiva, fino ad (eventualmente) accedere alla memoria principale.

Quando la CPU richiede un dato e lo trova accedendo in cache, si dice che è avvenuto un **cache hit**, altrimenti, se non dovesse essere presente, obbligandola a cercare in memoria principale, si dice che è

avvenuto un **cache miss**. Quando si scrive del codice, bisogna sfruttare il principio di località spaziale per ridurre al minimo i cache miss.

La cache su più livelli può causare problemi di inconsistenza, de facto se un certo dato x si trova su più cache, questo deve mantenere il suo valore identico in tutte quante: All'aggiornamento di x , sarà necessario aggiornarlo sia sulla memoria principale, che su ogni cache in cui è presente.

Quando la CPU modifica un dato nella cache, quest'ultimo potrebbe essere inconsistente con il valore nella memoria principale

Si identificano due soluzioni

- **write-through** : ad ogni aggiornamento della cache, viene aggiornato il dato anche in memoria principale.
- **write-back** : quando un dato viene aggiornato sulla cache viene marcato come *sporco*, una volta che il blocco di memoria in cui è presente viene rimosso dalla cache, il dato sporco viene aggiornato nella memoria principale.

♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪ ♪

4.2 La Cache nei Sistemi Multicore

Nei sistemi multicore, ogni core è in possesso di una cache privata di livello 1, mentre le cache di livello 2 e 3 potrebbero essere condivise. Il programmatore, non ha alcun controllo sulla gestione della cache e sulle frequenze di aggiornamento. Il problema dell'inconsistenza si fa più complesso in quanto una stessa variabile deve essere consistente nelle cache di tutti i core.

- *istante 0* : il core A prende x dalla RAM e lo mette nella sua cache.
- *istante 1* : il core B prende x dalla RAM e lo mette nella sua cache.
- *istante 2* : il core A modifica x , comando ad esso 1.
- *istante 3* : il core B modifica x , sottraendo ad esso 1.
- *istante 4* : x ha un valore diverso in ogni unità di memoria (cache di A , cache di B , RAM).

Ci sono due meccanismi per mantenere la cache coerente

- **Snooping Cache Coherence**
 - Condivisione del bus: Tutti i core (processori) sono connessi a un bus comune, un canale di comunicazione attraverso il quale vengono trasmesse tutte le informazioni.
 - Visibilità universale: Qualsiasi segnale inviato sul bus può essere "visto" da tutti i core collegati.
 - Aggiornamento e broadcast: Quando un core modifica il valore di un dato (ad esempio, la variabile x) presente nella sua cache, trasmette questa informazione a tutti gli altri core attraverso il bus.
 - Invalidazione della cache: Se un altro core sta utilizzando il bus e nota che il valore di x è stato modificato, invalida la propria copia di x nella cache, assicurandosi così di avere sempre il valore più aggiornato.
 - Limiti: Questo metodo, sebbene semplice, diventa inefficiente in sistemi multi-core con un gran numero di processori, poiché il broadcast su un bus condiviso può fare un collo di bottiglia.

- **Directory Based Cache Coherence**

- Struttura dati directory: Viene utilizzata una struttura dati chiamata directory per tenere traccia dello stato di ogni linea di cache. Questa directory contiene informazioni su quali core hanno una copia valida di una particolare linea di cache.
- Aggiornamento e invalidazione: Quando un valore viene modificato, la directory viene consultata per identificare tutti i core che hanno una copia di quel valore nella loro cache. I controller di cache di questi core vengono quindi informati di invalidare la loro copia, garantendo così la coerenza.

- Vantaggi: Questo metodo è più scalabile rispetto allo snooping, in quanto evita il broadcast indiscriminato su un bus condiviso. È particolarmente adatto per sistemi multi-core con un gran numero di processori.

4.2.1 False Sharing

Il **false sharing** è una problematica comune che degrada le prestazioni, può sorgere nelle architetture multicore quando più thread, eseguiti in parallelo, accedono a dati che, pur essendo logicamente distinti, risiedono fisicamente sullo stesso blocco della cache.

Il problema sottopone i dati a **invalidazioni inutili**. Se due thread accedono a dati diversi che si trovano sulla stessa linea (o blocco) di cache, ogni modifica di un dato da parte di un thread causerà l'invalidazione dell'intera linea di cache per tutti gli altri thread che condividono quella linea. Questo comporta un overhead in termini di prestazioni, poiché i dati validi vengono scartati inutilmente.

Esempio : Si considerino due thread che incrementano due contatori distinti. Se questi contatori sono allocati in modo tale da risiedere sulla stessa linea di cache, ogni incremento di un contatore comporterà l'invalidazione dell'intera linea, anche se l'altro contatore non è stato modificato

Per evitare il false sharing, è possibile adottare alcune strategie:

- **Padding:** Aggiungere byte "vuoti" ai dati in modo da farli iniziare su una nuova linea di cache. È necessario conoscere le dimensioni delle linee, è possibile tramite la chiamata

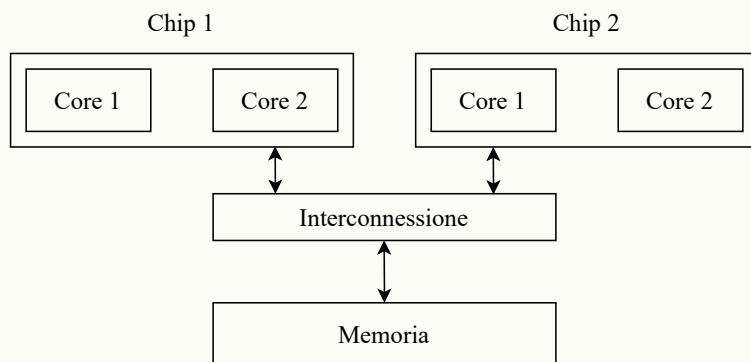
```
sysconf(_SC_LEVEL1_DCACHE_LINESIZE)
```

- **Strutture dati :** Progettare le strutture dati in modo che i dati acceduti da thread diversi siano allocati su linee di cache diverse.
- **Località :** Mantenere i dati il più possibile locali al thread, ad esempio sullo stack, e scriverli in memoria condivisa solo quando è strettamente necessario.

4.2.2 Organizzazione della Memoria

La memoria può essere organizzata in due modi diversi nei sistemi multicore.

Uniform Memory Access



In un sistema **UMA**, tutti i processori hanno accesso diretto alla memoria. Questo significa che qualsiasi processore può accedere a qualsiasi dato in memoria allo stesso tempo e con la stessa velocità, indipendentemente da dove si trovi fisicamente il dato.

- I programmati non devono preoccuparsi di gestire l'accesso alla memoria in modo complesso, poiché tutti i processori vedono la stessa memoria.
- In molte applicazioni, l'accesso uniforme alla memoria può portare a prestazioni migliori, poiché i processori possono accedere ai dati di cui hanno bisogno rapidamente.

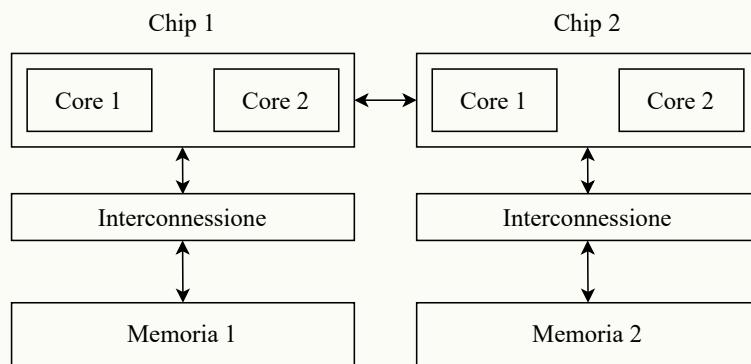
Differentemente, nei sistemi **NUMA**, i diversi core vengono partizionati in *gruppi*, ed ogni gruppo dispone di una memoria principale distinta e disconnessa dalle altre, tale suddivisione è *trasparente* al programmatore, in quanto funzionalmente la memoria viene vista come un'unica unità.

I nodi sono interconnessi tra loro tramite un'interconnessione ad alta velocità, che consente ai processori di accedere alla memoria di altri nodi

Nonostante ciò, i core che accedono alla memoria del proprio gruppo saranno più rapidi rispetto quelli che accedono alla memoria di un gruppo diverso, per questioni di vicinanza fisica dei chip.

È possibile forzare l'allocazione della memoria su una specifica unità tramite l'inclusione della libreria `numa.h`.

Non Uniform Memory Access



CAPITOLO

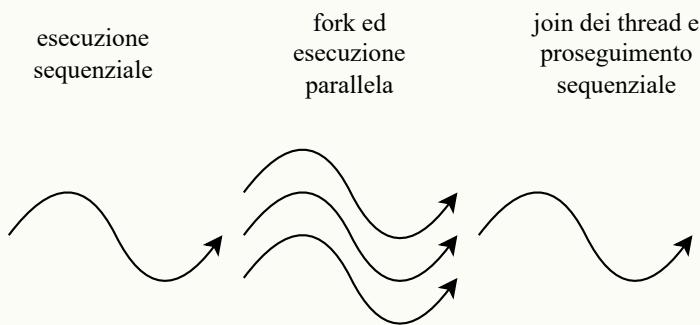
5

GESTIONE DEI THREAD : OPENMP

OpenMP è un framework di programmazione per i sistemi multicore a memoria condivisa, in particolare, costituisce un *interfaccia* di gestione dei thread ad alto livello rispetto quella fornita da PThread. OpenMP vede il sistema come un’insieme di più CPU o core che possono accedere ad una memoria condivisa. In pratica, OpenMP permette di parallelizzare un codice sequenziale in maniera semplice tramite dei micro-interventi sul codice e tramite l’assistenza del compilatore.

Lo scopo di OpenMP è la decomposizione di alcune porzioni del programma al fine di renderle parallele (diramazione) per svolgere un determinato compito, alla fine di esso i thread possono essere eliminati ed il programma può tornare ad un esecuzione sequenziale.

il codice è globalmente sequenziale e localmente parallelo



Fa utilizzo di alcune direttive da dare al compilatore, per utilizzare OpenMP è quindi necessario sia importare la libreria `omp.h`, sia avere un compilatore che supporti le direttive, che in caso contrario verranno ignorate.

5.1 Direttive pragma

Le *pragma* sono delle direttive speciali che elabora il pre processore, come anticipato, se openMP non è supportato dal compilatore, queste ultime verranno ignorate. Servono per conferire al sistema funzionalità non incluse nel comportamento standard del linguaggio C.

La direttiva più semplice ma anche più importante è

```
# pragma omp parallel
```

Una volta dichiarata nel codice, il blocco successivo verrà eseguito in parallelo (se e solo se il compilatore supporta OpenMP), è possibile anche specificare il numero di thread che dovranno eseguire in parallelo il blocco. Sono anche fondamentali le seguenti funzioni

- `omp_get_thread_num()` : ritorna l'identificatore del thread chiamante.
- `omp_get_num_threads()` : ritorna il numero di thread attivi. Se chiamata in una porzione sequenziale, ritornerà 1.

Si consideri il seguente esempio

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 void Hello (void);
6 int main(int argc, char** argv){
7     int thread_count = atoi(argv[1]);
8     #pragma omp parallel num_threads(thread_count)
9     Hello(); /* verrà eseguita in parallelo */
10    return 0;
11 }
12
13 void Hello(){
14     int my_rank = omp_get_thread_num();
15     int thread_count = omp_get_num_threads();
16     printf("Hello from thread %d of %d",my_rank,thread_count);
17 }
```

La funzione `Hello()` essendo preceduta dalla direttiva pragma verrà eseguita in parallelo. OpenMP funziona come Pthreads, vi è un thread main che si occupa di eseguire i fork generando gli altri thread, per poi ri-congiungerli (join) ad una sola esecuzione. Per motivi di ottimizzazione, OpenMP potrebbe mantenere i thread sempre attivi in uno stato di "idle" (attesa), per poi richiamarli ed utilizzarli quando necessario, evitando di crearne di nuovi ogni volta.

Attenzione : Se OpenMP non è supportata la direttiva pragma sarà ignorata, però le funzioni `omp_get_thread_num()`, `omp_get_num_threads()` genereranno un errore di compilazione in quanto non sono definite. Quando si compila un codice che utilizza OpenMP va inclusa l'opzione `-fopenmp`.

Il numero dei thread da eseguire può essere controllato in diversi modi

- È possibile impostare l'apposita variabile d'ambiente `OMP_NUM_THREADS` tramite la bash, ad esempio

```
export OMP_NUM_THREADS=4
```

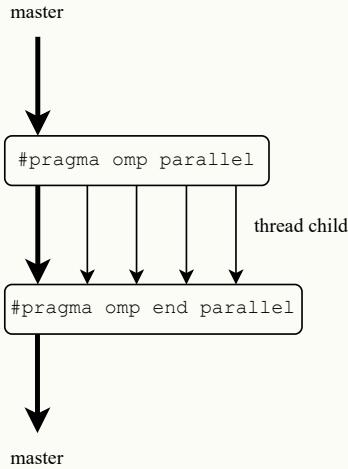
Questa specifica varrà per tutti i programmi che faranno uso di OpenMP in quell'ambiente.

- Si può cambiare il numero di thread da eseguire tramite l'apposita funzione `omp_set_num_threads()` all'interno del codice.
- L'ultimo modo, mostrato nell'esempio superiore, è tramite la specifica nella direttiva pragma con la clausola `num_threads`.

Come `num_threads` esistono altre clausole in grado di modificare le direttive pragma, specificando dettagli aggiuntivi sul comportamento.

In base al sistema, potrebbero essere definite delle limitazioni riguardo il numero di thread che un programma può avviare, lo standard OpenMP non garantisce sempre che il numero di thread specificato nella clausola sia avviato, ma finché il numero di thread è ridotto, si può assumere che siano eseguiti quelli specificati. È importante sapere che al termine di ogni blocco parallelo vi è la presenza di una *barriera implicita*.

L'insieme dei thread che vengono eseguiti in parallelo su uno stesso blocco è denominato *team*, all'interno di esso si identificano



- **master** : il thread originale dell'esecuzione (che avvia la funzione `main()`)
- **parent** : il thread che incontra la direttiva pragma sull'avvio dell'esecuzione in parallelo e si occupa di generare gli altri thread (spesso coincide con il master)
- **child** : ogni thread che viene avviato da un thread parent

Quando si include OpenMP nel proprio codice, si vuole far sì che quest'ultimo sia flessibile anche nei sistemi il cui compilatore non supporta lo standard. Si è già accennato al fatto che le direttive pragma verranno ignorate in tal caso, non è però la stessa cosa per le funzioni di libreria come `omp_set_num_threads()` o altre. È possibile utilizzare la variabile d'ambiente `_OPENMP`, quest'ultima è definita nei sistemi che supportano OpenMP, quindi può essere argomento della direttiva `#ifdef` in modo da rendere flessibile il codice permettendogli di evitare di chiamare funzioni di libreria che risulterebbero non dichiarate.

```

1 #ifdef _OPENMP
2     int my_rank = omp_get_thread_num();
3     int thread_count = omp_get_num_threads();
4 #else
5     int my_rank = 0;
6     int thread_count = 1;
7 #endif
  
```

♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

5.2 Mutua Esclusione

Nonostante OpenMP fornisca un interfaccia di alto livello, è comunque possibile far cadere i thread in una situazione di race condition quando questi ultimi accedono a variabili condivise simultaneamente. A tal proposito, esiste una direttiva che permette di simulare una lock, in particolare, il blocco che segue la direttiva, si assume "chiuso" da una `lock` prima di esso, ed un `unlock` dopo. La direttiva in questione è

```
#pragma omp critical
```

Garantisce la mutua esclusione del blocco che segue.

Grazie alle direttive pragma è possibile trasformare un codice sequenziale in parallelo con l'aggiunta di pochissime linee di codice. Tale semplicità ha permesso ad OpenMP di essere adottato in moltissime applicazioni, ed essere utilizzato da coloro che non hanno una conoscenza approfondita dei sistemi multicore.

5.2.1 Integrazione Numerica con OpenMP

Il codice parallelizzato con OpenMP che esegue la regola del trapezoido per calcolare l'integrale definito di una funzione $f : \mathbb{R} \rightarrow \mathbb{R}$ è identico al sequenziale, ad eccezione dell'aggiunta di sole due linee di codice.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 void Trap(double a, double b, int n, double *global_result_p)
6 {
7     double h, x, my_result;
8     double local_a, local_b;
9     int i, local_n;
10    #ifdef _OPENMP
11        int my_rank = omp_get_thread_num();
12        int thread_count = omp_get_num_threads();
13    #else
14        int my_rank = 0;
15        int thread_count = 1;
16    #endif
17
18    h = (b - a) / n;
19    local_n = n / thread_count;
20    local_a = a + my_rank * local_n * h;
21    my_result = (f(local_a) + f(local_b)) / 2.0; /* f è la funzione integranda*/
22    for (i = 0; i <= local_n - 1; i++)
23    {
24        x = local_a + i * h;
25        my_result += f(x);
26    }
27    my_result = my_result * h;
28    #pragma omp critical
29        *global_result_p += my_result;
30 }
31
32 void main(int argc, char **argv)
33 {
34     double global_result = 0.0;
35     double a, b;
36     int n;
37     int thread_count = atoi(argv[1]);
38     printf("Enter a,b and n\n", &a, &b, &n);
39     #pragma omp parallel num_threads(thread_count);
40         Trap(a, b, n, &global_result);
41
42     printf("result : %f", global_result);
43 }
```

Alcuni processori moderni implementano le operazioni di incremento come un'unica istruzione atomica

load + sum + store

In tal caso è possibile utilizzare la direttiva pragma

#pragma omp atomic

che risulta più efficiente della direttiva precedente in quanto copre la mutua esclusione solo dell'aggiornamento della variabile e non di tutto il blocco.

OpenMP è un'interfaccia di alto livello che garantisce meno flessibilità e controllo rispetto PThread ma permette una parallelizzazione del codice più semplice, è inoltre portatile e può funzionare anche su Windows. Con OpenMP è possibile programmare la GPU.

5.2.2 Riduzione

OpenMP permette di eseguire le collettive di riduzione, come, ad esempio la chiamata **MPI_Reduce**. Nell'esempio del trapezoido, ogni thread, piuttosto che restituire la somma parziale, si occupava di aggiornare la variabile condivisa, è possibile far sì che i thread ritornino le somme parziali, e che queste ultime vengano utilizzate in un'operazione di riduzione.

Le riduzioni in OpenMP sono operazioni binarie (come la somma o la moltiplicazione). È un calcolo che applica ripetutamente lo stesso operatore di riduzione a una sequenza di valori, fino a ottenere un unico risultato finale. Tutti i risultati intermedi di questa operazione vengono accumulati in una stessa

variabile, chiamata variabile di riduzione.

La clausola di riduzione viene aggiunta ad una direttiva pragma ed è definita come segue

```
reduction(<operator>:<variabile list>)
```

Bisogna specificare l'operatore da eseguire fra quelli disponibili

```
1 + * - & | ^ && ||
```

Va poi specificata la variabile in questione che conterrà il risultato finale

```
1 global_result = 0.0;
2 #pragma omp parallel num_threads(thread_count) reduction(+:global_result)
3     global_result += Local_trap(double a, double b, int n);
```

♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

5.3 Scoping

Con *scope di una variabile*, si intendono le porzioni di codice in cui quella variabile è accessibile e può essere utilizzata, definisce il ciclo di vita/visibilità delle variabili. Nei programmi paralleli con OpenMP, con scope ci si riferisce all'insieme di thread che può accedere la specifica variabile nel blocco parallelo. Una variabile può essere

- **shared** : accessibile da ogni thread del blocco. Le variabili dichiarate prima di un blocco parallelo rientrano in questa categoria. Inoltre anche le zone di memoria allocate sullo stack del thread master sono accessibili dai suoi figli.
- **private** : accessibile da un singolo thread, le variabili dichiarate all'interno di un blocco parallelo sono private, ed ogni thread ne avrà una sua copia distinta dalle altre.

È possibile specificare la visibilità di ogni singola variabile tramite delle clausole apposite, con la clausola

```
default(None)
```

si sta dicendo al compilatore che, per default, nessuna variabile dichiarata al di fuori del blocco sarà considerata automaticamente privata o condivisa all'interno di quel blocco. Ciò significa che il programmatore dovrà specificare esplicitamente lo scope di ogni variabile che vuole utilizzare, evitando ambiguità. Data una variabile, le clausole di specifica disponibili sono le seguenti

- **shared** : è il comportamento di default delle variabili al di fuori del blocco parallelo, va quindi specificata per queste ultime solo se presente anche la clausola `default(None)`
- **reduction** : si specifica che una variabile dovrà essere soggetta ad un'operazione di riduzione. Ci saranno delle copie di quest'ultima private per ogni thread, ma il risultato finale sarà condiviso
- **private** : la variabile avrà una copia privata per ogni thread nel blocco. Le variabili private non sono inizializzate.

```
1 int x = 5;
2 #pragma omp parallel private(x)
3 {
4     x = x + 1;
5 }
6 printf("x is %d", x);
```

Nel blocco di codice mostrato, la chiamata di funzione `printf` stamperà in output "x is 5". Inoltre, l'incremento di `x` all'interno del blocco parallelo può causare errore in quanto `x` è privata e non inizializzata.

Esistono altre 4 clausole per specificare lo scope di una variabile

- **firstprivate** : Come la clausola `private`, crea una copia locale della variabile per ogni thread all'interno del blocco parallelo. A differenza di `private`, la copia locale viene inizializzata con il valore della variabile originale (quella "esterna" al blocco) prima che il blocco venga eseguito.

- `lastprivate` : Anche questa clausola crea copie locali delle variabili, come `private`. Al termine dell'esecuzione del blocco parallelo, il valore della copia locale dell'ultimo thread che ha completato il ciclo viene copiato nella variabile originale (quella "esterna" al blocco).

Quando dichiariamo una variabile come `private` in un contesto di programmazione parallela, questa diventa inaccessibile al di fuori del blocco di codice in cui è stata dichiarata. Questo è utile per evitare conflitti tra i thread, ma a volte può essere necessario avere una variabile privata che persista tra diverse sezioni parallele.

- `threadprivate` : Crea una copia locale e persistente di una variabile globale o statica per ogni thread. Questa copia è accessibile solo al thread corrispondente e mantiene il suo valore per tutta la durata del programma. È ideale quando è necessario conservare dati privati che devono essere mantenuti tra diverse sezioni parallele, ma che non devono essere condivisi tra i thread.
- `copyin` : Viene utilizzata in combinazione con `threadprivate` per inizializzare le copie locali di ogni thread con il valore della variabile originale (quella definita nel thread master). È utile quando si vuole che tutti i thread inizino con lo stesso valore iniziale per una variabile persistente.

Esempio riassuntivo

```

1 #include <omp.h>
2 int x, y, z[1000];
3 #pragma omp threadprivate(x)
4
5 void function(int a){
6     default(none);
7     const int c = 1;
8     int i = 0;
9 #pragma omp parallel default(none) private(a) shared(z)
10 {
11     int j = omp_get_num_threads();
12     a = z[j]; /* OK : 'j' è dichiarata nella sezione parallela */
13     x = c; /* OK : 'x' è threadprivate */
14     z[i] = y; /* Errore : non si possono referenziare 'i' o 'y', data
15                 la clausola default(none) */
16 }
17 }
```

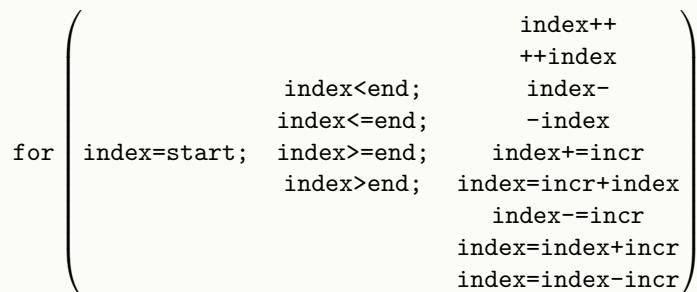
♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

5.4 Cicli For Parallelvi

Si vogliono parallelizzare le iterazioni (indipendenti) dei blocchi ciclici nel C, queste ultime sono spesso le operazioni più dispendiose, rendendo favorevole una loro parallelizzazione. OpenMP è in grado di parallelizzare esclusivamente i cicli `for`, è assolutamente necessario che il numero di iterazioni totali sia noto a priori. La direttiva

```
#pragma omp parallel for
```

permette di parallelizzare un ciclo for, facendo sì che ogni thread si occuperà di un sottoinsieme delle iterazioni totali. Un ciclo for per essere parallelizzato si deve attenere alla seguente struttura



solo in questo modo il numero di iterazioni sarà noto a priori. È quindi importante che

- la variabile `index` sia un tipo intero o puntatore.
- `start`, `end`, `incr` devono avere tipi compatibili.
- `incr` deve essere costante.
- l'indice del ciclo `index` non può essere modificato all'interno del ciclo, ma solo tramite l'espressione di incremento.

Il seguente codice riporta un esempio di for parallelizzato nell'ambito dell'integrazione con la regola del trapezoide.

```

1 h=(b-a)/n;
2 approx=(f(a)+f(b))/2.;
3 #pragma omp parallel for num_threads(thread_count) reduction(+:approx)
4 for(int i = 1; i<n; i++){
5     approx+=f(a+i*h);
6 }
7 approx=h*approx;

```

Il seguente ciclo

```

1 for (i=0;i<n;i++){
2     if ([condizione])
3         break;
4 }

```

non può essere parallelizzato perché il numero di iterazioni non è noto a priori. Anche il seguente

```

1 for (i=0;i<n;i++){
2     if ([condizione])
3         i++;
4 }

```

non può essere parallelizzato perché l'indice viene incrementato dentro il ciclo. Il seguente

```

1 for (i=0;i<n;i++){
2     if ([condizione])
3         exit();
4 }

```

nonostante il numero di iterazioni non sia noto a priori, può essere parallelizzato perché la clausola `exit()` termina l'intero programma e non è quindi necessario preoccuparsi di un eventuale comportamento anomalo.

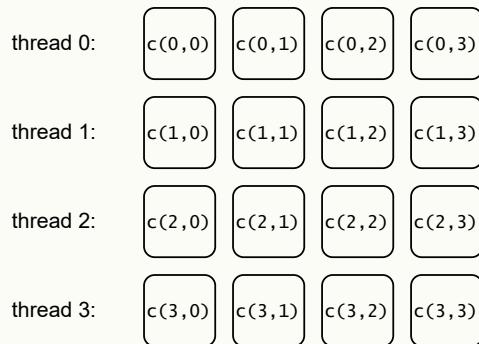
5.4.1 Cicli Annidati

Nella maggior parte dei casi in cui ci sono due cicli for annidati, è sufficiente parallelizzare solo il primo ciclo. È importante sapere che alla fine di ogni ciclo for parallelizzato è presente una *barriera implicita*. Il seguente esempio mostra come vengono suddivise le iterazioni ai vari thread in un ciclo for parallelizzato.

```

1 #pragma omp parallel for num_threads(thread_count)
2 for(i=0;i<4;i++){
3     for(j=0;j<4;j++){
4         c(i,j);
5     }
6 }

```

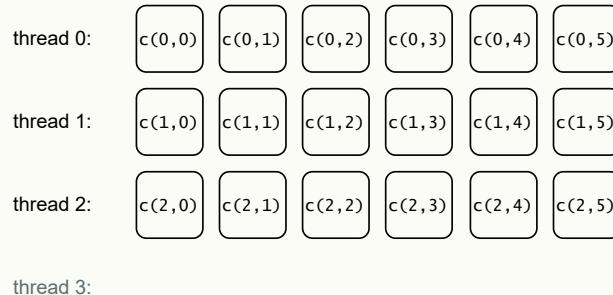


Ogni thread (dei 4) esegue una iterazione del ciclo for esterno (parallelizzato), e per ognuna di questa, esegue le 4 iterazioni del ciclo for interno non parallelizzato. Si consideri la seguente situazione, in cui ci sono 4 thread, ed un ciclo esterno di 3 iterazioni.

```

1 #pragma omp parallel for num_threads(thread_count)
2 for (i=0;i<3;i++){
3     for (i=0;i<6;i++){
4         c(i,j);
5     }
6 }
```

le iterazioni sono assegnate in questo modo

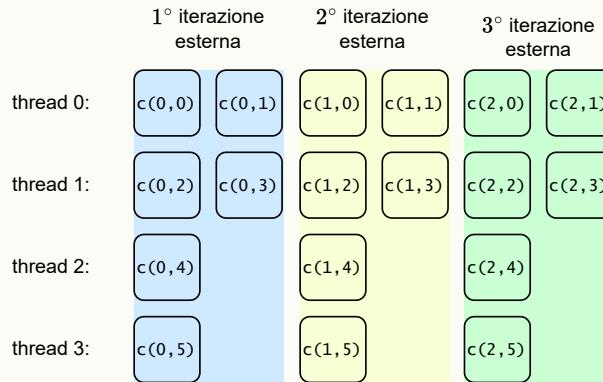


I 3 cicli esterni verranno assegnati ai primi 3 thread, il quarto thread quindi non verrà considerato nel calcolo parallelo, in tal modo la distribuzione del carico di lavoro è poco uniforme e certamente non efficiente. È sicuramente meglio parallelizzare il for interno

- ad ogni iterazione esterna, il ciclo interno (di 6 iterazioni) eseguito da 4 thread
- i primi due thread eseguiranno 2 iterazioni, gli altri due 1 iterazione.

```

1 for ( i=0;i<3;i++){
2 #pragma omp parallel for num_threads(thread_count)
3     for ( i=0;i<6;i++){
4         c ( i , j ) ;
5     }
6 }
```

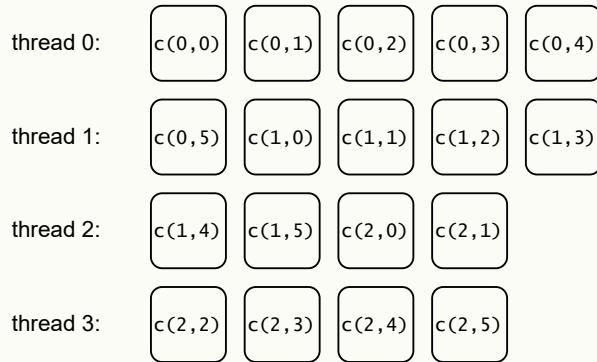


Nonostante questa soluzione sia migliore della precedente, non è ancora ottimale. La soluzione migliore (nei casi in cui i cicli annidati sono molto semplici) è di collassare i due cicli in un unico ciclo tramite degli accorgimenti sugli indici, ad esempio:

```

1 #pragma omp parallel for num_threads(thread_count)
2 for (ij=0;ij<3*6;ij++){
3     c ( ij /6 , ij%6 );
4 }
```

In questo caso la distribuzione dei cicli è ottimale (il più uniforme possibile).



Si noti come in quest'ultima soluzione sono necessarie 5 iterazioni totali per completare il lavoro, dove invece nelle soluzioni precedenti (meno uniformi) ne erano necessarie 6. OpenMP fornisce una clausola per collassare automaticamente due cicli for annidati in uno singolo, ossia

`collapse(f)`

Dove `f` è il numero di cicli da collassare. Risulta piuttosto utile nei casi come il precedente, quando il numero di iterazioni del ciclo esterno è basso (paragonabile, se non minore, dei core sulla macchina).

```

1 #pragma omp parallel for num_threads(thread_count) collapse(2)
2 for(i=0;i<3;i++){
3     for(i=0;i<6;i++){
4         c(i,j);
5     }
6 }
```

Di default, è vietato parallelizzare due cicli annidati con la direttiva `parallel for`.

```

1 #pragma omp parallel for num_threads(thread_count)
2 for(i=0;i<3;i++){
3     #pragma omp parallel for num_threads(thread_count) /* VIETATO */
4         for(i=0;i<6;i++){
5             c(i,j);
6         }
7 }
```

5.4.2 Iterazioni Dipendenti

La parallelizzazione di un ciclo for non impone alcun ordinamento sulle iterazioni, è possibile che un thread esegua l'iterazione `i+1` prima di un thread che esegue la `i`, è quindi ovvio che è possibile parallelizzare un ciclo finché le iterazioni sono indipendenti fra loro.

Il seguente esempio è il più esplicativo, per calcolare l' i -esimo numero di fibonacci è necessario conoscerne l' $i-1$ -esimo, non è quindi assicurata la correttezza di un ciclo parallelizzato che deve calcolare la sequenza dei primi n numeri di fibonacci.

```

1 fibo[0]= fibo[1]=1;
2 #pragma omp parallel for num_threads(thread_count)
3 for(i=2;i<n;i++){
4     fibo[i]= fibo[i-1] + fibo[i-2];
5 }
```

Perché non funziona?

- l'iterazione `i=3` potrebbe essere calcolata prima dell'iterazione `i=2`, supponiamo sia questo il caso
- l'iterazione `i=3` considerà l'elemento `fibo[2]`, questo dovrebbe essere uguale a 2, ma visto che l'iterazione `i=2` non è stata eseguita, il valore dentro l'array è nullo,
- l'iterazione `i=3` calcolerà `fibo[3]=fibo[2]+fibo[1]=0+1=1`

- il risultato è sbagliato in quanto il terzo numero di fibonacci è 3.

Quando si parallelizza un ciclo for è obbligatorio assicurarsi che non ci siano dipendenze fra i dati tra le varie iterazioni, dipendenze di questo tipo sono dette **loop-carried dependence**. Bisogna modificare il codice dei cicli per rendere le iterazioni indipendenti.

Dato un loop della seguente forma

```

1 for ( i =... ) {
2   ...
3   S1 : operazione sulla locazione di memoria x
4   ...
5   S2 : operazione sulla locazione di memoria x
6   ...
7 }
```

ci sono due operazioni su una stessa locazione di memoria, essendo queste contenute in un'unica iterazione, si ha la certezza che verranno eseguite da uno stesso thread nell'ordine in cui sono state stabilite. Il problema si verifica quando due operazioni che vanno eseguite sequenzialmente, si trovano in due iterazioni differenti, ad esempio

```

1 for ( i =... ) {
2   ...
3   S2 : operazione sul valore x[i-1]
4   ...
5   S1 : operazione sul valore x[i]
6   ...
7 }
```

in questo caso l'operazione **S2** potrebbe essere eseguita prima dell'operazione **S1** se un'iterazione **i** viene eseguita prima dell'iterazione **i-1**.

Ci sono 4 differenti tipi di dipendenze che coinvolgono due operazioni **S1,S2** che non possono essere eseguite in due iterazioni differenti:

- **RAW** : Read After Write (una lettura dopo una scrittura)

```

1 x=10;           //S1
2 y=2*x+5;       //S2
```

se **S2** fosse eseguita per prima, potrebbe leggere un valore di **x** non corretto.

- **WAR** : Write After Read (una scrittura dopo una lettura)

```

1 y=x+3;          //S1
2 x++;            //S2
```

se **S2** fosse eseguita per prima, la variabile **y** leggerebbe un valore errato di **x**.

- **WAW** : Write After Write (una scrittura dopo una scrittura)

```

1 x=10;           //S1
2 x=x+c;          //S2
```

se **S2** fosse eseguita per prima, verrebbe incrementata anche se **x** dovesse essere diverso da 10, anche se ogni volta che si esegue **S2**, dovrebbe sempre essere uguale a 10.

- **RAR** : Read After Read (una lettura dopo una lettura)

```

1 y=x+c;          //S1
2 z=2*x+1;        //S2
```

non è una vera dipendenza e non causa problemi.

5.4.3 Risoluzione delle Dipendenze RAW

In *alcuni casi* è possibile rimuovere le dipendenze di tipo RAW nei cicli, in particolare, si identificano 6 metodologie di risoluzione.

1) Sistemare le induction/reduction variable

Si consideri il seguente ciclo:

```

1 double v = start;
2 double sum = 0;
3 for(int i = 0; i < N; i++){
4     sum = sum + f(v);    //S1
5     v = v + step;        //S2
6 }
```

Definizione : Una *induction variable*, o *variabile induttiva*, è una variabile che all'interno di un ciclo viene incrementata di un valore costante ad ogni iterazione.

Le dipendenze sono

- una dipendenza di tipo RAW per l'operazione **S1** sulla variabile di riduzione **sum**, che ad ogni iterazione, utilizza il valore dell'iterazione precedente.
- una dipendenza di tipo RAW per l'operazione **S2** sulla variabile induttiva **v**, che ad ogni iterazione, utilizza il valore dell'iterazione precedente.
- una dipendenza di tipo RAW fra le due operazioni, dato che **S1** utilizza il valore di **v** calcolato all'iterazione precedente.

Essendo che **v** è una variabile induttiva, il suo valore può essere calcolato a prescindere per ogni iterazione **i** in maniera indipendente.

$$\begin{aligned}
 i = 0 &\implies v = \text{start} \\
 i = 1 &\implies v = \text{start} + \text{step} \\
 i = 2 &\implies v = (\text{start} + \text{step}) + \text{step} \\
 \vdots & \\
 i = k &\implies v = \text{start} + \text{step}*k
 \end{aligned}$$

è quindi possibile rimuovere due dipendenze calcolando **v** a partire dall'indice **i** senza dover utilizzare il valore dell'iterazione precedente.

```

1 double v = start;
2 double sum = 0;
3 for(int i = 0; i < N; i++){
4     v = start + i*step;
5     sum = sum + f(v);
6 }
```

L'unica dipendenza da risolvere riguarda il valore di **sum** che è dipendente dalle precedenti iterazioni, ma per ovviare a ciò, è sufficiente specificare che **sum** è una variabile di riduzione nella direttiva pragma, facendo sì che se ne occupi openMP.

```

1 double v = start;
2 double sum = 0;
3 #pragma omp parallel for reduction(+:sum) private(v)
4 for(int i = 0; i < N; i++){
5     v = start + i*step;
6     sum = sum + f(v);
7 }
```

2) Loop skewing

Questa tecnica consiste nel riarrangiare il loop eseguendo una sottospecie di "shift" delle operazioni (l'esempio renderà chiaro il concetto). Si consideri la seguente porzione di codice:

```

1 for(int i = 1; i < N; i++){
2     y[i] = f(x[i-1]);
3     x[i] = x[i] + c[i];
4 }
```

Il ciclo **i** è dipendente dal ciclo **i-1**, in quanto **y[i]** per essere aggiornata necessita di **x[i-1]**. In questo metodo, si "srotola" il loop, considerando le diverse iterazioni in sequenza, ad esempio:

```

i=1: y[1]=f(x[0]);
      x[1]=x[1]+c[1];
i=2: y[2]=f(x[1]);
      x[2]=x[2]+c[2];
i=3: y[3]=f(x[2]);
      x[3]=x[3]+c[3];
      :
i=N-2: y[N-2]=f(x[N-3]);
      x[N-2]=x[N-2]+c[N-2];
i=N-1: y[N-1]=f(x[N-2]);
      x[N-1]=x[N-1]+c[N-1];
  
```

dipendenti
dipendenti
dipendenti

Basta raggruppare in un'unica iterazione le operazioni dipendenti, "shiftando" di un'operazione il ciclo for, come segue

```

1 y[1]=f(x[0]);
2 for( int i = 1; i < N-1; i++){
3     x[i]=x[i]+c[i];
4     y[i+1]= f(x[i]);
5 }
6 x[N-1]=x[N-1]+c[N-1];
  
```

Riscrivendo il ciclo in questo modo si sono rimosse le dipendenze fra diverse iterazioni, ed il ciclo può essere parallelizzato.

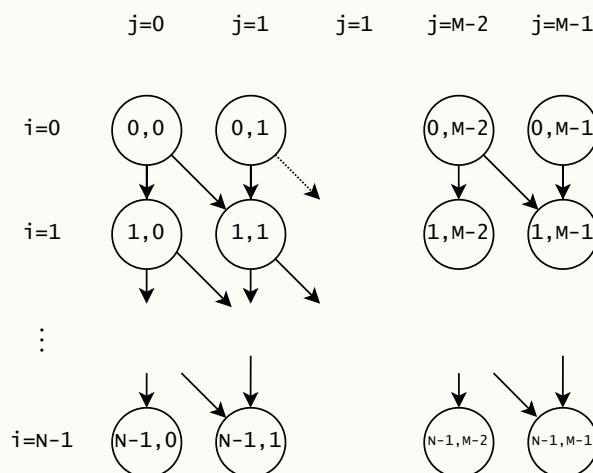
3) Parallelizzazione parziale

Questa metodologia è la più semplice, consiste nell'individuare le iterazioni indipendenti tramite la costruzione di un grafo, si consideri il seguente ciclo annidato

```

1 for( int i = 1; i < N; i++){
2     for( int j = 1; j < M; j++){
3         data[i][j]=data[i-1][j]+data[i-1][j-1];
4     }
5 }
  
```

si disegna un grafo in cui ogni nodo rappresenta un'iterazione, e vi è un arco dal nodo q al nodo q' se l'iterazione q' è dipendente dall'iterazione q . Nell'esempio del ciclo riportato qua sopra, si ha il seguente grafo:



Si osserva che non ci sono archi che collegano due nodi sulla stessa riga, quindi le righe possono essere parallelizzate senza problemi, in quanto le iterazioni su ogni riga (ossia, fissato i e al variare di j) non sono dipendenti fra loro.

```

1 for (int i = 1; i < N; i++){
2     #pragma omp parallel for
3         for (int j = 1; j < M; j++){
4             data [i] [j]=data [i-1][j]+data [i-1][j-1];
5         }
6 }
```

Anche se apparentemente l'iterazione j dipende dall'iterazione $j-1$, si ha che j è eseguita all'iterazione esterna i , e $j-1$ è eseguita all'iterazione esterna $i-1$, essendo garantita la sequenzialità dell'ordinamento esterno, questa dipendenza apparente su j non causa problemi.

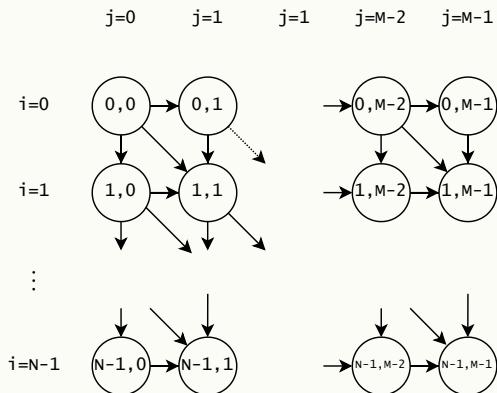
4) Refactoring

Questo metodo consiste nella riscrittura dei cicli in modo da renderli parallelizzabili, si consideri il seguente esempio

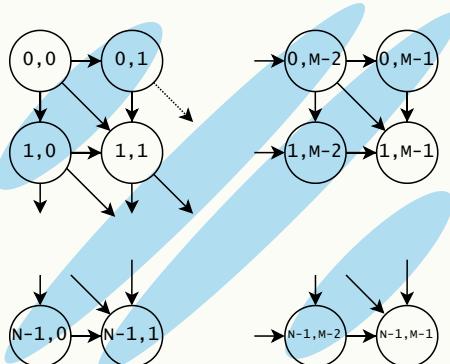
```

1 for (int i = 1; i < N; i++){
2     for (int j = 1; j < M; j++){
3         data [i] [j]=data [i-1][j]+data [i-1][j-1]+data [i] [j-1];
4     }
5 }
```

si ha il seguente grafo:



In questo caso anche le righe sono dipendenti fra loro, ed il ciclo sembra non parallelizzabile. Che succede se si considerano i nodi sulle diagonali del grafo?



non ci sono dipendenze fra le iterazioni sulla diagonale, queste si definiscono *onde*, e possono essere parallelizzate, anche se è difficile individuarle nel codice, ed il ciclo va completamente riscritto. Un idea di implementazione può essere la seguente:

```

1 for (wave = 0; wave<NumWaves; wave++){
2     diag=F(wave);
3     #pragma omp parallel for
4     for(k=0; k < diag; k++){
5         int i = get_i(diag,k);
6         int j = get_j(diag,k);
7         data[i][j]=data[i-1][j]+data[i-1][j-1]+data[i][j-1];
8     }
9 }
```

5) Fissione del Loop

Si divide semplicemente il loop in una parte parallelizzabile ed in una parte sequenziale. Il ciclo

```

1 s=b[0];
2 for (int i = 1; i < N; i++){
3     a[i]=a[i]+a[i-1]; //S1
4     s=s+b[i];          //S2
5 }
```

ha l'operazione **S1** che non si può parallelizzare, e l'operazione **S2** che può essere parallelizzata, quindi si divide in due cicli:

```

1 for (int i = 1; i < N; i++){
2     a[i]=a[i]+a[i-1]; //S1
3 }
4 s=b[0];
5 #pragma omp parallel for
6 for (int i = 1; i < N; i++){
7     s=s+b[i];          //S2
8 }
```

6) Cambio dell'algoritmo

Se nessun metodo precedente è applicabile, è necessario cambiare totalmente l'algoritmo, ad esempio, la sequenza di fibonacci può essere parallelizzata se calcolata tramite la formula di Binet:

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \quad \text{dove} \quad \frac{1}{\varphi} \simeq 0.618$$

```

1 fibo[0]=fibo[1]=1;
2 #pragma omp parallel for
3 for (int i=2; i < N; i++){
4     fibo[i]=(pow(phi,n)-pow(1-phi,n))/sqrt(5);
5 }
```

5.4.4 Rimozione di dipendenze WAR e WAW

Le dipendenze WAR possono essere risolte semplicemente, si consideri il seguente ciclo

```

1 for (int i=0; i < N-1; i++){
2     a[i]=a[i+1]+c;
3 }
```

Nell'iterazione **i** si scrive un valore che verrà letto nell'iterazione successiva **i+1**. È sufficiente creare una copia dell'array **a[]** da utilizzare nel ciclo da parallelizzare.

```

1 for (int i=0; i < N-1; i++){
2     a2[i]=a[i+1];
3 }
4
5 #pragma omp parallel for
6 for (int i=0; i < N-1; i++){
7     a[i]=a2[i]+c;
8 }
```

Bisogna valutare se il tempo risparmiato nel parallelizzare il ciclo sia minore o maggiore del tempo utilizzato per eseguire la copia dell'array.

Anche le dipendenze WAW possono essere risolte, spesso è sufficiente usare un costrutto di openMP, si consideri il seguente ciclo

```

1 for( int i = 0; i < N; i++){
2     y[ i]=a*x[ i]+c;    //S1
3     d=fabs(y[ i ]);    //S2
4 }
```

la dipendenza è sulla variabile `d`, il valore finale di questa, una volta che termina la sezione parallela, deve essere quello calcolato nell'ultima iterazione, ossia `fabs(y[N-1])`. È sufficiente utilizzare la clausole `lastprivate` per propagare all'esterno del for il valore di `d` calcolato dal thread che ha eseguito l'ultima iterazione.

```

1 #pragma omp parallel for shared(a, c) lastprivate(d)
2 for( int i = 0; i < N; i++){
3     y[ i]=a*x[ i]+c;    //S1
4     d=fabs(y[ i ]);    //S2
5 }
```

♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

5.5 Schedluing Loops

Consideriamo la seguente funzione

```

1 double f(int i){
2     int j, start = i*(i+1)/2, finish = start + i;
3     double return_val = 0.0;
4     for(j=start; j<=finish; j++)
5         return_val+=sin(j);
6     return return_val;
7 }
```

La complessità della funzione è lineare nelle dimensioni dell'input. Consideriamo il ciclo for

```

1 double sum = 0.0;
2 for( i=0;i<=n; i++)
3     sum+=f( i );
```

se dovessimo parallelizzare le `n` iterazioni, come verrebbero assegnate ai differenti thread nell'effettivo?

Thread	Iterations
0	0, 1, 2, ..., n/t - 1
1	n/t, n/t + 1, ..., 2n/t
...	...
t-1	n(t-1)/t, ..., n-1

Thread	Iterations
0	0, n/t, 2n/t, ...
1	1, n/t + 1, 2n/t + 1, ...
:	:
t - 1	t - 1, n/t + t - 1, 2n/t + t - 1, ...

Default partitioning.

Cyclic partitioning.

Se venissero partizionati come nel caso di sinistra (che è lo scheduling di default adottato da openMP), il carico di lavoro sarebbe sbilanciato, dato che i thread con numero di rank maggiore eseguiranno le ultime iterazioni, che data la struttura della funzione `f(i)`, sono più dispendiose dal punto di vista computazionale, è quindi preferibile una *partizione ciclica* come nel caso mostrato a destra. È possibile specificare ad openMP la modalità di partizionamento con la clausola `schedule`

```

1 double sum = 0.0;
2 #pragma omp parallel for reduction(+:sum) schedule( static , 1)
3 for( i=0;i<=n; i++)
4     sum+=f( i );
```

La modalità di `schedule` può essere

- **statica** : le iterazioni sono assegnate ai thread prima dell'esecuzione del ciclo
- **dinamica/guidata** : l'assegnazione avviene durante l'esecuzione del ciclo

- **auto** : l'assegnazione è determinata dal compilatore/sistema operativo

Il numero specificato nella clausola `schedule`, detto **chunk size** indica il numero di iterazioni contenute in un gruppo che verrà poi partizionato in maniera ciclica, ad esempio, con 3 thread e 12 iterazioni si ha:

```
schedule(static,1)
    Thread 0 : 0,3,6,9
    Thread 1 : 1,4,7,10
    Thread 2 : 2,5,8,11

schedule(static,2)
    Thread 0 : 0,1,6,7
    Thread 1 : 2,3,8,9
    Thread 2 : 3,4,10,11

schedule(static,4)
    Thread 0 : 0,1,2,3
    Thread 1 : 4,5,6,7
    Thread 2 : 8,9,10,11
```

Nello scheduling **dinamico**, ad ogni thread viene assegnato un gruppo (o chunk) di iterazioni, una volta che lo ha terminato, richiederà il prossimo al sistema operativo, la pianificazione dell'assegnazione è quindi più dispendiosa in quanto avviene in fase di esecuzione, ma comporta un miglior bilanciamento del carico. La variante **guidata** è identica, ma ogni volta che un thread richiede un nuovo chunk, il numero di iterazioni per chunk viene diminuito.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Lo scheduling può anche essere deciso in fase di esecuzione modificando la variabile d'ambiente `OMP_SCHEDULE`, ad esempio

```
export OMP_SCHEDULE = "static, 1"
```

Riassumendo:

- se il costo di ogni iterazione è omogeneo è sufficiente schedulare le iterazioni in maniera statica
- se il costo di ogni iterazione può variare è opportuno impostare uno scheduling dinamico/guidato
- in ogni caso è opportuno provare differenti soluzioni

Riguardo l'ordine delle iterazioni, esiste la clausola `ordered` che può essere specificata all'interno di un `for` parallelo, questa fa sì che il blocco di codice che la segue verrà eseguito in ordine sequenziale (dalla prima all'ultima iterazione).

```
1 #pragma omp parallel for ordered schedule(static,1)
2 for(int i = 0; i < N; i++){
3     //processamento dei dati
4     //...
5     //print dei risultati in ordine
```

```

6 #pragma omp ordered
7 cout<< data[ i ];
8 }
```

♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

5.6 Argomenti Extra sulle Sezioni Critiche

È possibile *denominare* una sezione critica in openMP

```
# pragma omp critical (nome)
```

Tale direttiva crea una sezione critica specifica con il nome fornito. Ogni sezione critica con nome ha il suo blocco. Questo permette ai thread di entrare in sezioni critiche con nomi diversi contemporaneamente senza bloccarsi a vicenda, aumentando il parallelismo.

Vantaggi delle Sezioni Critiche con Nome:

- Maggiore Parallelismo: Permette l'accesso concorrente a diverse risorse protette, migliorando le prestazioni complessive.
- Contesa Ridotta : Evita che i thread si blocchino a vicenda inutilmente quando accedono a risorse indipendenti.
- Migliore Organizzazione del Codice: Rende più chiaro quale sezione critica protegge quale risorsa, migliorando la leggibilità e la manutenibilità del codice.

OpenMP fornisce un costrutto di lock per accedere in maniera sequenziale ad una sezione di codice

```

1 omp_lock_t writelock ;
2 omp_init_lock(&writelock) ;
3 #pragma omp parallel for
4 for ( i = 0; i < x; i++) {
5     // some stuff
6
7     omp_set_lock(&writelock) ;
8     // one thread at a time stuff
9
10    omp_unset_lock(&writelock) ;
11    // some stuff
12 }
13 omp_destroy_lock(&writelock) ;
```

Per le sezioni critiche è meglio utilizzare **critical**, **atomic** oppure le lock?

1. In generale, la direttiva **atomic** è potenzialmente il metodo più veloce per ottenere la mutua esclusione
2. Se un blocco di codice consiste in più istruzioni macchina è meglio utilizzare la direttiva **critical** (potrebbe dipendere dall'implementazione)
3. L'uso dei lock dovrebbe probabilmente essere riservato a situazioni in cui la mutua esclusione è necessaria per una struttura di dati piuttosto che per un blocco di codice

5.6.1 Sezioni Parallele

Le direttive **master** e **single** in OpenMP sono strumenti utili per controllare l'esecuzione di specifiche sezioni di codice all'interno di un costrutto parallelo. Entrambe limitano l'esecuzione di un blocco di codice a un singolo thread, ma lo fanno in modi leggermente diversi.

- La direttiva **master** specifica che il blocco di codice associato deve essere eseguito solo dal thread master del team di thread. Il thread master è il thread che ha iniziato la regione parallela.
- La direttiva **single** specifica che il blocco di codice associato deve essere eseguito da uno qualsiasi dei thread del team, ma solo uno. OpenMP sceglie quale thread eseguirà il blocco single. Non è garantito che sia il thread master. In questo caso inoltre, vi è una barriera implicita al termine del blocco considerato.

Esiste anche la direttiva `#pragma omp barrier` per far sì che i thread si risincronizzino in un punto comune.

Le direttive `master` e `single` precludono ad alcuni thread l'esecuzione di una sezione, esiste anche la direttiva `#pragma omp parallel section` che equivale ad uno `switch` parallelo, ossia permette di specificare ai thread un comportamento diverso da eseguire in parallelo agli altri.

```

1 #pragma omp parallel sections
2 {
3     #pragma omp section
4     {
5         // concurrent block 0
6     }
7
8     // ... (altre sezioni)
9
10    #pragma omp section
11    {
12        // concurrent block M-1
13    }
14 }
```

♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪

5.7 Impatto del False Sharing

Nella sezione 4.2.1 è stata introdotta la problematica del *false sharing* relativa alle invalidazioni inutili delle linee di cache quando dati logicamente distinti sono vicini in memoria fisica. Una possibile soluzione vista è quella di fare del *padding* sui dati
senza padding

```

1 double x[N];
2 #pragma omp parallel for schedule(static , 1)
3 for(int i = 0; i < N; i++)
4     x[i] = someFunc(x[i]);
```

con padding

```

1 double x[N][8];
2 #pragma omp parallel for schedule(static , 1)
3 for(int i = 0; i < N; i++)
4     x[i][0] = someFunc(x[i][0]);
```

È una soluzione apparentemente elegante ma comporta uno spreco di memoria e potrebbe rendere inutile l'utilizzo della cache, una possibile soluzione alternativa è quella di cambiare l'accesso ai dati (ad esempio, cambiando lo scheduling) in modo che non ci sia accesso parallelo a sezioni di memoria adiacenti.

```

1 double x[N];
2 #pragma omp parallel for schedule(static , 8)
3 for(int i = 0; i < N; i++)
4     x[i] = someFunc(x[i]);
```

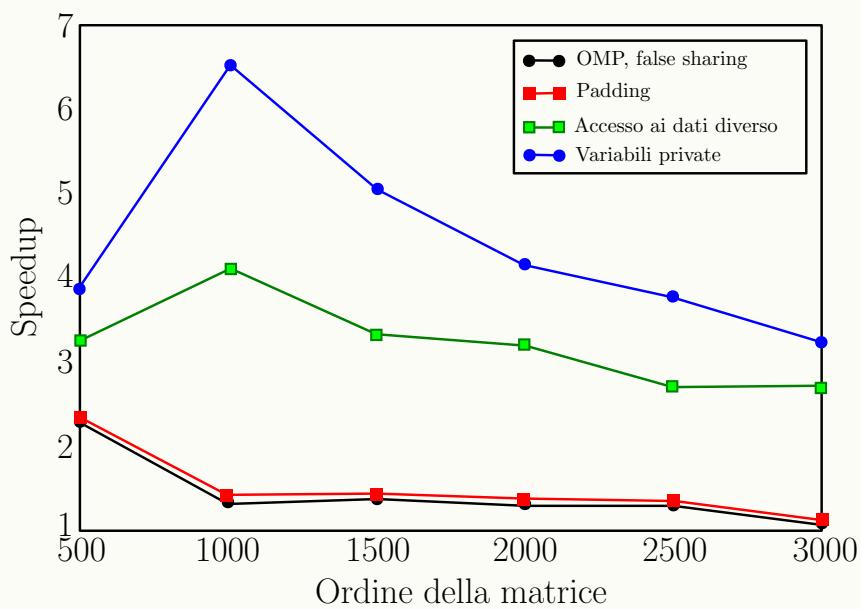
Una soluzione altrettanto auspicabile riguarda l'utilizzo di variabili private:

```

1 double x[N]; //Si assume che N sia un multiplo di 8
2 #pragma omp parallel for schedule(static , 1)
3 for(int i = 0; i < N; i++){
4     double temp[8];
5     for(int j = 0; j < 8; j++){
6         temp[j] = someFunc(x[i+j]);
7     }
8     memcpy(x+i, temp, 8*sizeof(double));
9 }
```

In questo modo ogni thread parallelamente definirà un'array locale `temp` fisicamente disgiunto dalle copie private degli altri thread.

Il seguente grafico riporta lo speed up di un programma che esegue moltiplicazione fra matrici in funzione dell'ordine della matrice, mostra chiaramente come il false sharing sia deleterio per le prestazioni, va quindi aggrirato.



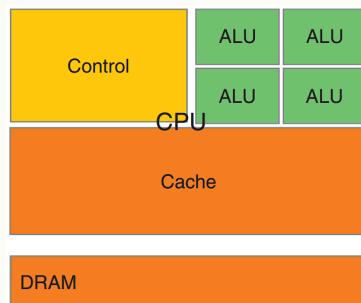
CAPITOLO

6

PROGRAMMAZIONE DI GPU : CUDA

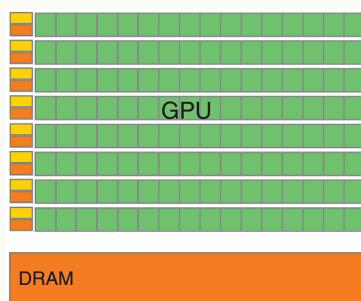
6.1 Architettura della GPU

I processori ordinari, hanno un architettura dedicata alla *riduzione della latenza* per le singole operazioni.



Lo scopo della CPU è quello di eseguire codice sequenziale il più rapidamente possibile, ed è caratterizzata da un alta frequenza di clock. Gran parte del chip della CPU, è occupata dalla cache, utile per ridurre gli accessi in memoria, che causano un alta latenza, inoltre, una grossa parte è composta dall'unità di controllo, che si occupa di gestire le istruzioni, eseguire il prefetching, branch predictor ed altre operazioni simili.

Le componenti dedicate alla computazione effettiva costituiscono una piccola parte di tutto il processore, inoltre le ALU sono molto performanti. Differente è la situazione per le GPU (anche dette schede video).



Queste presentano un'architettura differente, la frequenza di clock è moderata, le unità di controllo sono più semplici (no branch prediction) e la cache è ridotta. Una GPU presenta molte più unità di calcolo rispetto alla CPU, la differenza sostanziale risiede nel fatto che queste sono progettate per **massimizzare**

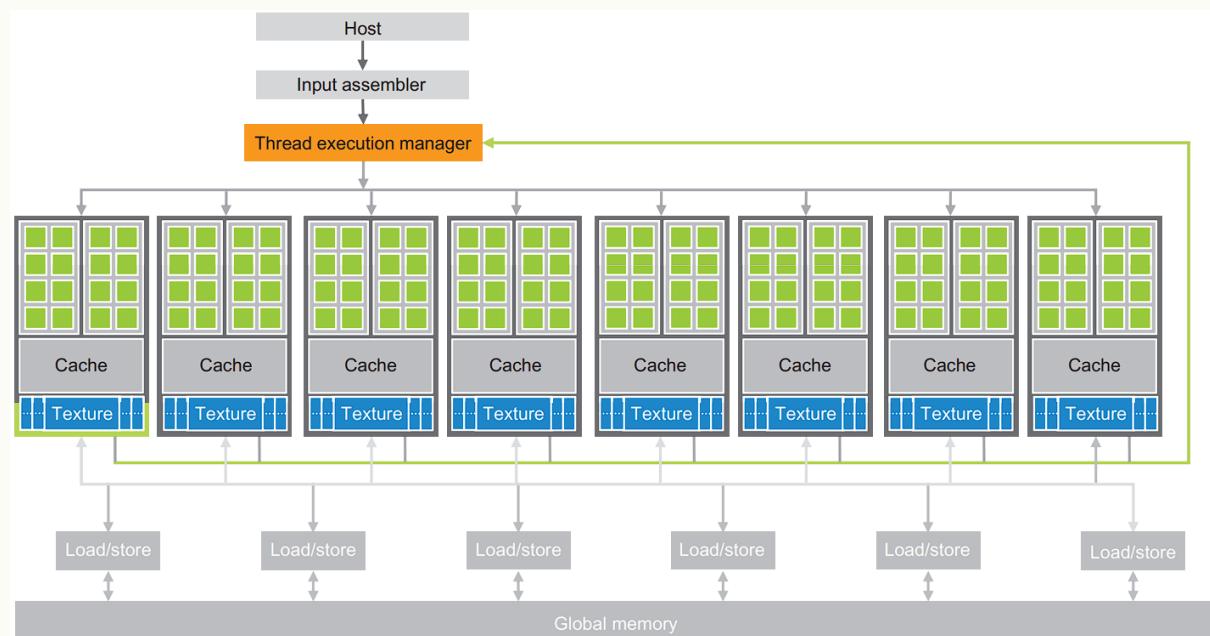
il **throughput** (numero di task processati per unità di tempo), piuttosto che ridurre la latenza delle singole operazioni.

Una particolarità delle GPU è che, gruppi di core condividono la stessa unità di controllo, questi ultimi saranno costretti ad eseguire quindi le stesse istruzioni contemporaneamente (seppur su dati differenti).

GPU è l'acronimo di *Graphics Processing Unit*, dato che queste sono nate per il rendering grafico. Ad oggi il loro utilizzo è ampliamente più vasto, vengono applicate nel calcolo parallelo, in quanto si prestano ad eseguire un gran numero di operazioni semplici alla volta (un esempio calzante di operazione in cui le GPU risultano efficienti, è il prodotto fra matrici).

È importante sapere che il cambio di contesto sui core della GPU è notevolmente rapido rispetto a quello eseguito sulla CPU. La memoria della scheda video, detta VRAM o HBM (High Bandit Memory) è separata dalla RAM, ha un'alta banda e permette a molti thread di accedervi in parallelo.

La seguente immagine descrive ad alto livello l'architettura di una scheda video:



Con **Host**, si definisce la coppia CPU-RAM su cui viene eseguito classicamente il codice, essendo che i core della GPU non possono accedere alla RAM, solitamente il ciclo di esecuzione dei programmi consiste nei seguenti passi

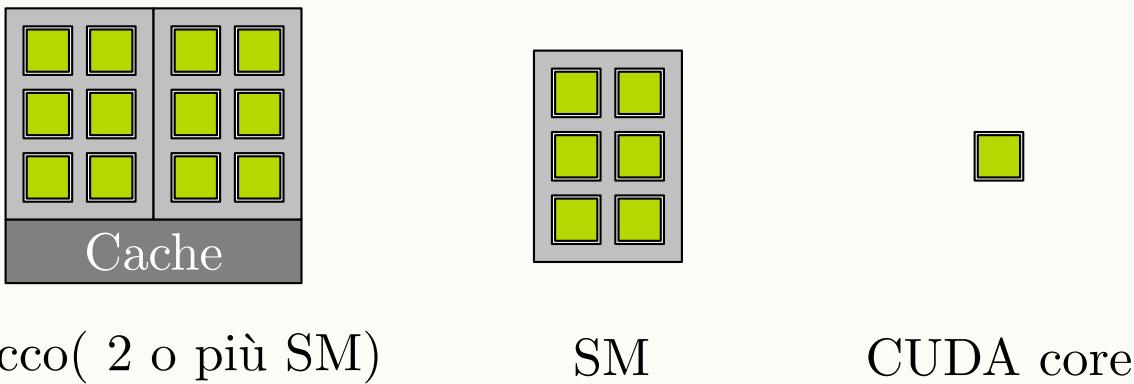
- I dati su cui vanno eseguite le operazioni vengono passati dall'host alla VRAM (memoria della GPU)
- La GPU esegue il calcolo parallelo sui dati
- Una volta ottenuto il risultato, lo trasferisce nella memoria principale dell'host

Attenzione : La GPU non dovrebbe mai fare I/O

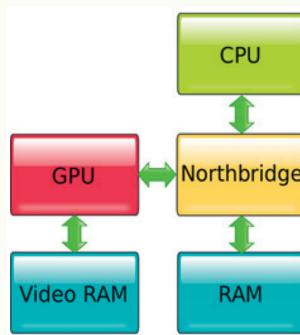
I singoli blocchi verdi mostrati in figura rappresentano i singoli core, anche detti **CUDA core** (ogni core possiede dei registri privati, come per la classica CPU). I core sono divisi in **Streaming Multiprocessor (SM)**, un SM racchiude più core che condividono la stessa unità di controllo

i core di uno stesso SM non possono eseguire istruzioni differenti contemporaneamente

Due o più SM costituiscono un **blocco**. Tutti i core possono accedere alla memoria globale condivisa (HBM), anche se per i blocchi è riservata una cache (come nel caso della CPU, questa è trasparente al programmatore).



L'host e la GPU sono connesse da un bus detto *Northbridge*.



Idealmente, bisogna ridurre al minimo la frequenza con la quale si trasferiscono i dati dalla RAM alla VRAM. Le due memorie sono separate ed il trasferimento dei dati fra queste va gestito in maniera esplicita, vedremo come CUDA si occupa di trasportare i dati fra le due memorie in maniera trasparente.

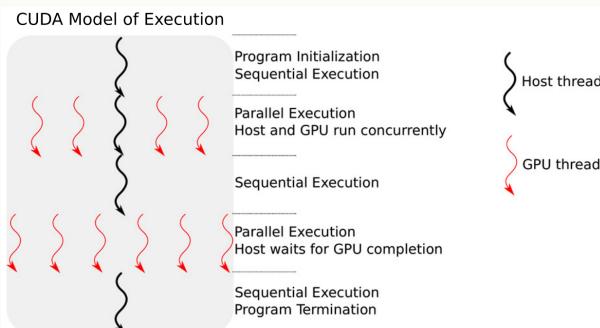
♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪

6.2 Introduzione a CUDA

CUDA (Compute Unified Device Architecture) è una piattaforma di programmazione parallela sviluppata da NVIDIA che permette di utilizzare direttamente la scheda video, si può dire che è un modello di programmazione *general-purpose*, e permette di gestire in modo esplicito la memoria. La GPU è vista come un *acceleratore* da utilizzare per dei calcoli specifici in un programma.

CUDA funziona solo su GPU NVIDIA

Le funzioni definite nel codice che verranno eseguite dai core della GPU sono chiamate **CUDA kernel** (o solamente kernel).



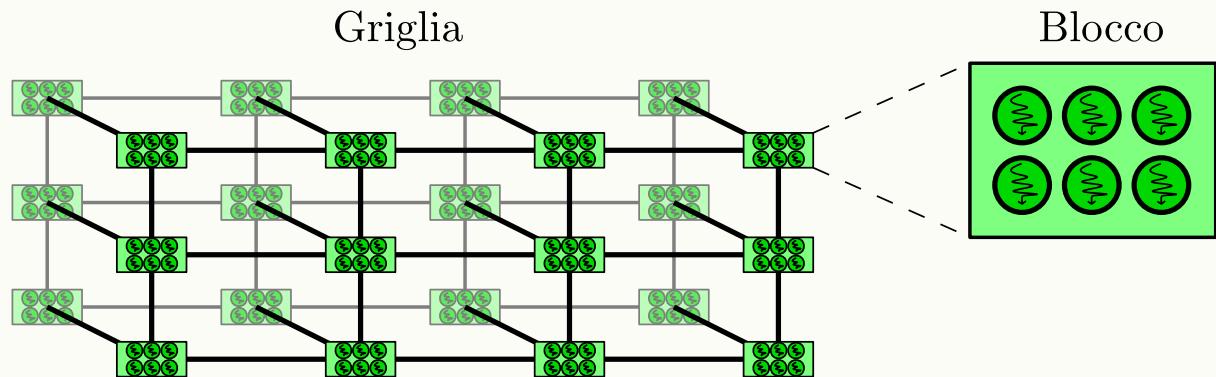
6.2.1 Modello di Esecuzione

Quando si scrive del codice da eseguire sulla GPU, si definisce un elevato numero di thread (anche superiore ai core fisici presenti), in CUDA i thread vengono rappresentati (o meglio dire, organizzati) in uno spazio 6-dimensionale : ogni thread è *identificabile univocamente* da due vettori con 3 coordinate ciascuno.

Come accennato, ad ogni thread sarà associato

- un vettore 3-dimensionale che ne indica le coordinate nel **blocco**.
- un vettore 3-dimensionale che ne indica le coordinate nella **griglia**.
- un thread può conoscere la sua posizione (x, y, z, x', y', z') chiamando apposite funzioni di libreria.

La struttura ammette anche blocchi o griglie di dimensioni inferiori a 3. La seguente immagine riporta una struttura composta da una griglia 3-dimensionale $4 \times 3 \times 2$, composta da blocchi 2-dimensional 3 \times 2, per un totale di $4 \times 3 \times 2 \times 3 \times 2 = 144$ thread.



La **capability** è una proprietà associata ad ogni GPU, e definisce il numero massimo di thread per blocco, blocchi per griglia e dimensioni della struttura che può implementare una GPU, è rappresentata da un *version number* (a volte chiamato SM version), identifica le funzionalità supportate dall'hardware e le istruzioni disponibili sulla GPU.

Item	Compute Capability			
	1.x	2.x	3.x	5.x
Max. number of grid dimensions	2		3	
Grid maximum x-dimension	$2^{16} - 1$		$2^{31} - 1$	
Grid maximum y/z-dimension		$2^{16} - 1$		
Max. number of block dimensions		3		
Block max. x/y-dimension	512		1024	
Block max. z-dimension		64		
Max. threads per block	512		1024	
GPU example (GTX family chips)	8800	480	780	980

♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪ ♪♪♪

6.3 Struttura del Codice

In questa sezione si introduce la programmazione delle GPU in C utilizzando CUDA. Come prima cosa, è necessario specificare una funzione *kernel* che verrà eseguita da tutti i thread definiti. Si definisce il numero di thread, specificando le dimensioni della griglia e del blocco.

Sia `foo` la funzione kernel:

```

1 dim3 block(3,2);
2 dim3 grid(4,3,2);
3 foo<<<grid , block>>>();

```

`dim3` è un tipo di dato (3-vettore di interi) che permette di specificare le dimensioni di blocco e griglia (ogni dimensione non specificata si assume essere 1), nell'esempio riportato, si definiscono blocchi $3 \times 2 \times 1 \equiv 3 \times 2$, ed una griglia avrà dimensione $4 \times 3 \times 2$, ci saranno in totale 144 thread, che verranno distribuiti sui core ed eseguiranno in parallelo la funzione `foo`.

la decisione della struttura (assegnamento delle dimensioni) dipende dal tipo di problema

```

1 dim3 b(3,3,3);
2 dim3 g(20,100);
3 foo<<<g , b>>>(); /* si esegue il codice su una griglia 20x100 composta
4                               da blocchi 3x3x3, per un totale di 50000 thread*/
5
6 foo<<<g , 256>>>(); /*e' possibile passare un valore costante per
7                               avviare la funzione kernel, in questo caso,
8                               i blocchi saranno unidimensionali e
9                               composti da 256 thread, per un totale
10                             di 2000x256 thread. */
11
12 foo<<<g , 2048>>>(); /*questo comando e' invalido in quanto per
13                               nessuna capability permette di definire
14                               blocchi da piu' di 1024 thread*/
15
16 foo<<<5, g>>>(); /* anche questo esempio e' invalido, in quanto
17                               si specifica un blocco da 20x100>1024 thread.*/

```

i file in cui è presente codice CUDA devono avere l'estensione `.cu` e devono essere compilati dal comando `nvcc`, specificando anche il numero di versione con `-arch=sm_[versione]`

Il seguente esempio riporta una semplice funzione da eseguire sulla GPU (nonostante nell'esempio venga fatto, è sconsigliato fare I/O sulla GPU).

```

1 #include <stdio.h>
2 #include <cuda.h>
3
4 __global__ void hello(){
5     printf("Hello world\n");
6 }
7
8 int main(){
9     hello<<<1,10>>>();
10    cudaDeviceSynchronize();
11    return 1;
12 }

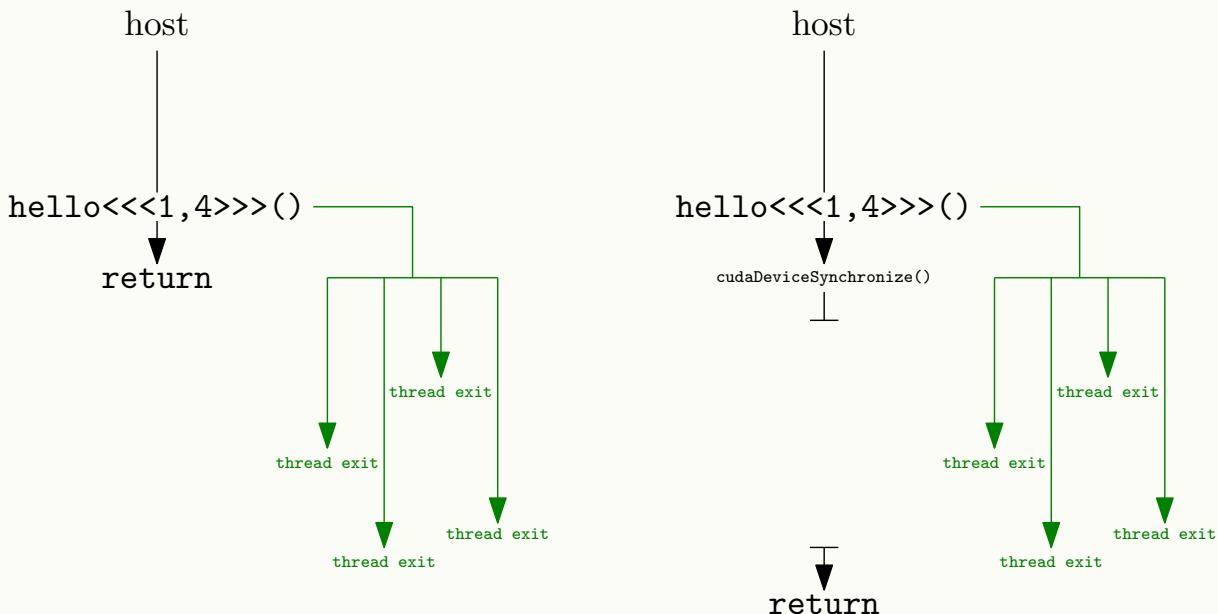
```

- le funzioni kernel devono essere di tipo `void`
- il decoratore `__global__` che precede la funzione kernel specifica che la funzione può essere chiamata sia dall'host che dalla GPU, ma può essere eseguita solamente sulla GPU

Esempio di compilazione:

```
nvcc -arch=sm_20 hello.cu -o hello
```

Nota importante : Le chiamate delle funzioni kernel sono asincrone, il codice host dopo aver chiamato una funzione da eseguire sulla GPU continuerà la sua esecuzione senza preoccuparsi dell'andamento del codice parallelo della scheda video, la funzione `cudaDeviceSynchronize()`, blocca il codice host ed attende la terminazione dell'esecuzione parallela.



Come `__global__`, ci sono altri decoratori

- `__device__` : la funzione può essere chiamata esclusivamente da un altro kernel all'interno della GPU
- `__host__` : la funzione può essere eseguita esclusivamente sull'host (può essere omesso).

Esistono diverse variabili che possono essere accedute (esclusivamente nella funzione kernel eseguita sulla GPU) che descrivono la posizione del thread all'interno della struttura 6-dimensionale.

- `blockDim` : restituisce la dimensione di ogni blocco tramite un vettore a 3 componenti.
- `gridDim` : restituisce la dimensione della griglia tramite un vettore a 3 componenti.
- `threadIdx` : restituisce le coordinate (x, y, z) del thread all'interno del blocco. Due thread possono avere stessi valori per questa variabile se sono in blocchi diversi.
- `blockIdx` : restituisce le coordinate (x', y', z') del blocco (in cui è contenuto il thread) nella griglia, due thread nello stesso blocco condivideranno tale variabile.

È possibile generare un ID univoco per ogni thread in funzione delle sue coordinate.

```

1 int myID = (blockIdx.z * gridDim.x * gridDim.y +
2             blockIdx.y * gridDim.x +
3             blockIdx.x) * blockDim.x * blockDim.y * blockDim.z +
4             threadIdx.z * blockDim.x * blockDim.y +
5             threadIdx.y * blockDim.x +
6             threadIdx.x;

```

6.3.1 Disposizione dei Thread

Abbiamo visto come

- Una GPU è composta da più core disposti su un SM, e più SM formano un blocco condividendo la cache
- I thread di CUDA sono organizzati logicamente in una struttura 6-dimensionale composta da blocchi su una griglia

Attenzione : I blocchi fisici della GPU (2 o più SM) ed i blocchi logici di CUDA (elementi della griglia) *non sono la stessa cosa*, seppur condividano il nome "blocco".

Ci interessa sapere : come viene mappata la griglia 6-dimensionale nei core fisici della GPU?

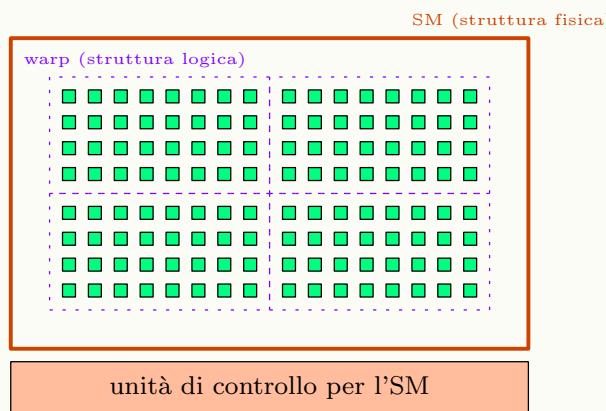
- ogni thread viene eseguito su un core fisico contenuto in un SM
- i core di uno stesso SM condividono la stessa unità di controllo e sono costretti ad eseguire in sincronia le stesse istruzioni
- differenti SM possono eseguire differenti funzioni kernel
- *Importante* : i thread di uno stesso blocco logico (della struttura di CUDA) vengono assegnati ai core di uno stesso SM, i thread di un blocco non possono essere divisi su CUDA core di differenti SM.
- Quando vanno eseguiti più thread di un blocco, CUDA li schedula tutti su un SM, una volta finito, schedulerà su questa i thread del blocco successivo (se questi vi entrano)

I thread di uno stesso blocco logico sono ulteriormente raggruppati in *gruppi* chiamati **warps** (solitamente, un warp contiene 32 thread). I thread di un unico blocco logico vengono divisi su più warps in accordo con i loro id del blocco.

tutti i thread di un warp eseguono simultaneamente la stessa istruzione

- più warp possono essere su un unico SM
- due warp possono eseguire istruzioni diverse
- i core di un SM non possono eseguire istruzioni diverse simultaneamente
- ne consegue che due warp su un'unica SM non potranno essere eseguiti in contemporanea se fanno istruzioni diverse (anche se ci sono core disponibili nell'SM)

Ad esempio, un SM con 128 core, potrebbe includere 4 warp da 32 thread ciascuno. Nonostante possono essere eseguiti 128 core in parallelo, ne verranno eseguiti solo 32 alla volta, dato che i thread di uno stesso warp possono eseguire le stesse istruzioni, thread di warp diversi (se hanno istruzioni differenti) non potranno essere eseguiti sui core fisici.



Il seguente esempio può rendere chiaro il concetto

```

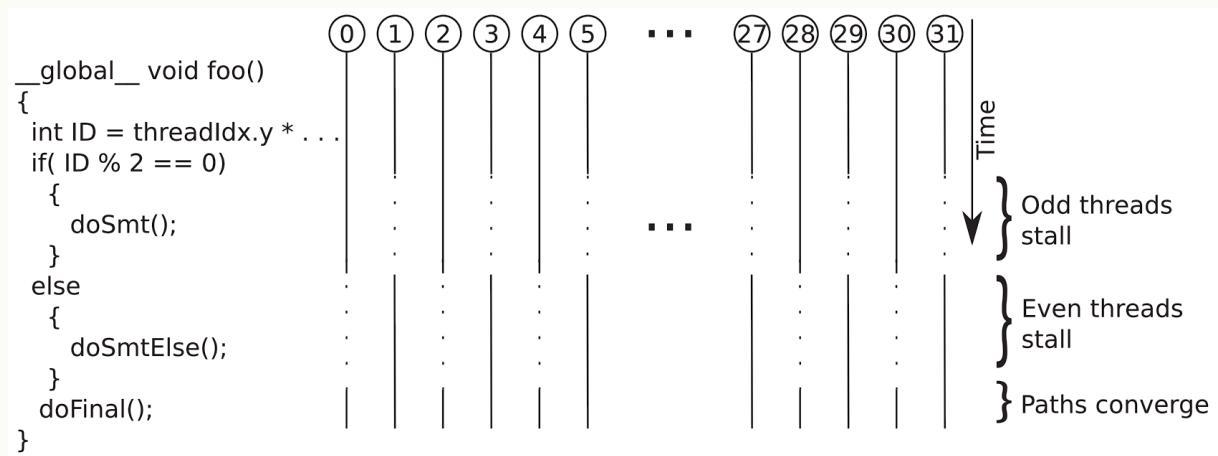
1  __global__ void foo(){
2      int ID = getThreadUniqueId(); //funzione mostrata precedentemente
3      if (ID%2==0){
4          funcA();
5      }
6      else{
7          funcB();
8      }
9      funcC();
10 }
```

supponiamo che tale funzione `foo()` sia eseguita da 32 thread in un'unico warp, la situazione è la seguente

- i thread dispari eseguiranno la funzione `funcA()`, i thread pari sono costretti ad eseguire le stesse istruzioni (dato che sono su uno stesso warp), quindi non potranno eseguire in parallelo la loro funzione

- una volta che i thread dispari hanno finito, i thread pari potranno eseguire `funcB()`. Finché questi non hanno finito, i thread dispari dovranno attenderli prima di continuare.
- una volta che i pari hanno finito con `funcB()`, tutti e 32 i thread potranno eseguire in contemporanea `funcC()`, eseguendo le stesse istruzioni.

Tale divergenza nell'esecuzione è detta **warp divergence**, ed costituisce un problema dal punto di vista computazionale.



6.3.2 Cambio di Contesto e Proprietà del Dispositivo

Soltanamente un SM supporta più warp di quanti ne possa eseguire simultaneamente

ad esempio, un SM potrebbe avere un totale di 64 core, ma supportare 4 warp da 32 thread ciascuno (128 thread in totale)

La particolarità delle GPU rispetto alle CPU, è che possono eseguire il cambio di contesto da un warp ad un altro in maniera estremamente rapida, senza perdita di tempo, è quindi *favorevole saturare* un SM con più thread di quanti ne possa eseguire in simultanea:

Se un gruppo di thread in un warp deve aspettare che un'operazione complessa termini (come leggere dati dalla memoria), l'SM passa ad eseguire un altro warp di thread che non è in attesa. Mentre un gruppo di thread aspetta, l'SM trova un altro gruppo pronto per essere eseguito.

È quindi utile mettere più thread su un SM di quanti ne possa eseguire in parallelo, in modo tale che, quando un warp viene interrotto (ad esempio, per attendere dati dalla memoria) con alta probabilità ne verrà schedulato un altro warp.

Esempio 1

Una scheda video (che supporta CUDA) è in grado di avere un massimo di 8 blocchi (logici), e 1024 thread su un'unico SM, e supporta al più 512 thread per blocco.

- **8×8** : Se considerassimo dei blocchi 8×8 , ogni blocco conterebbe 64 thread, essendo che ne possiamo avere 1024, avremo un totale di $\frac{1024}{64} = 16$ blocchi sull'SM. Nonostante ciò, un SM può avere *al massimo* 8 blocchi, quindi, ogni SM avrà $8 \times 64 = 512$ thread (la metà di quanti se ne potrebbero avere), e ci sarebbe uno spreco delle risorse. È fondamentale cercare di massimizzare il numero di thread scegliendo accuratamente le dimensioni del blocco.
- **16×16** : Se considerassimo dei blocchi 16×16 , ogni blocco conterebbe 256 thread, essendo che ne possiamo avere 1024, avremo un totale di $\frac{1024}{256} = 4$ blocchi sull'SM. Con questa configurazione, l'utilizzo di ogni SM è massimizzato.
- **32×32** : Se considerassimo dei blocchi 32×32 , ogni blocco conterebbe 1024 thread, ma il massimo numero di thread consentito per blocco è 512, quindi, questa configurazione non è valida.

Esempio 2

C'è una griglia $4 \times 5 \times 3$, questa contiene blocchi che possono avere al più 100 thread, in totale, ci sono 60 blocchi. La GPU, ha 18 SM. Bisogna distribuire i 60 blocchi totali sui 18 SM.

- supponiamo che i blocchi siano distribuiti in maniera round robin
- 12 SM riceveranno 4 blocchi, e 6 SM riceveranno 3 blocchi
- è poco efficiente, gli ultimi 12 blocchi saranno processati in simultanea da 12 SM, mentre ci saranno 6 SM che non stanno facendo computazione

Assumiamo che ogni warp è composto da 32 thread

- ogni blocco ha 100 thread
- in un blocco ci sono 4 warp [32 | 32 | 32 | 4]
- Supponiamo che ogni SM abbia 32 CUDA core \Rightarrow un solo warp alla volta può essere schedulato su ogni SM
- L'ultimo warp avrà 4 thread che schedulerà su un SM da 32 core, utilizzando solo il 12.5% delle risorse computazionali disponibili

Questo esempio mostra come la scelta di una buona struttura blocchi-griglia di CUDA è centrale quando si vuole massimizzare l'uso delle risorse.

il numero di thread in un blocco (scelto dall'utente) dovrebbe essere multiplo del numero di thread per warp (dato dalle specifiche della GPU)

In un programma che usa CUDA è possibile visualizzare tutte le schede video (che supportano CUDA) disponibili:

```

1 int deviceCount = 0;
2 cudaGetDeviceCount(&deviceCount); //per il numero di GPU disponibili
3 if(deviceCount==0)
4     printf("no CUDA compatible GPU exists.\n");
5 else{
6     cudaDeviceProp pr;
7     for(int i=0; i<deviceCount; i++){
8         cudaGetDeviceProperties(&pr, i); //per le proprieta' dell'i-esima GPU
9         printf("Device #%d is %s\n", i, pr.name);
10    }
11 }
```

Il tipo di dato `cudaDeviceProp` è una struttura che contiene diverse informazioni sulle specifiche della GPU, quali

```

1 struct cudaDeviceProp{
2     char name[256];
3     int major; //numero della versione (compute capability)
4     int minor; //numero della versione (compute capability)
5     int maxGridSize[3];
6     int maxThreadsDim[3];
7     int maxThreadsPerBlock;
8     int maxThreadsPerMultiProcessor; //thread massimi per SM
9     int multiProcessorCount; //numero di SM
10    int regsPerBlock; //numero di registri per ogni blocco
11    size_t sharedMemPerBlock;
12    size_t totalGlobalMem;
13    int warpSize;
14};
```



6.4 Gestione della Memoria

La memoria della GPU (VRAM) è separata dalla memoria dell'host (RAM), e i dati allocati sulla RAM non sono direttamente accessibili dai CUDA core. I dati su cui deve lavorare la GPU, vanno copiati sulla VRAM, il seguente frammento di codice:

```
1 int* mydata = (int*)malloc(sizeof(int)*N);
2 foo<<<grid , block>>>(mydata , N);
```

è errato, in quanto il puntatore `mydata` si riferisce ad una zona di memoria allocata sulla RAM, per i CUDA core quindi, tale puntatore non avrà significato. L'host può allocare (e gestire) la memoria sulla VRAM tramite le seguenti funzioni di libreria:

```
1 // allocare memoria
2 cudaError_t cudaMalloc(void** devPtr , size_t size);
3
4 // liberare memoria
5 cudaError_t cudaFree(void** devPtr);
6
7 // copiare memoria
8 cudaMemcpy(dst , const void* src , size_t count , cudaMemcpyKind kind);
```

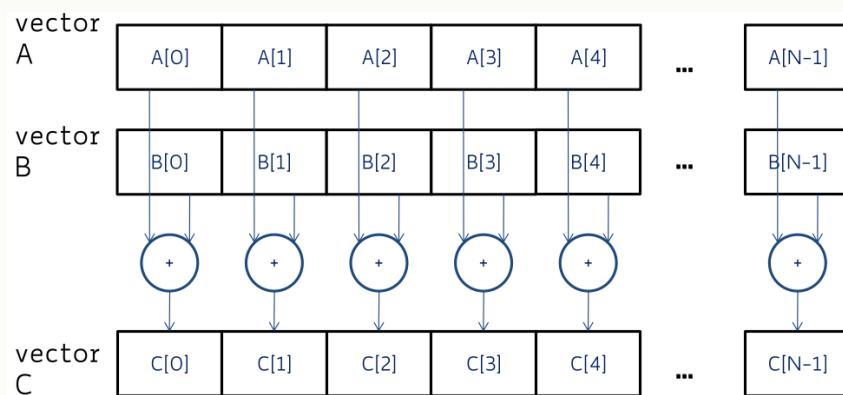
il tipo di queste funzioni è `cudaError_t`, serve ad identificare lo status della chiamata, se è 0, la funzione ha avuto successo. Nella `cudaMemcpy` è possibile specificare un parametro di tipo `cudaMemcpyKind`, questo, definisce la locazione del destinatario e del mittente:

- `kind=cudaMemcpyHostToHost=0` : copiare dati dalla RAM alla RAM
- `kind=cudaMemcpyHostToDevice=1` : copiare dati dalla RAM alla VRAM
- `kind=cudaMemcpyDeviceToHost=2` : copiare dati dalla VRAM alla RAM
- `kind=cudaMemcpyDeviceToDevice=3` : copiare dati dalla VRAM alla VRAM (nei sistemi con più GPU)

Dalla versione 6 di CUDA, è possibile utilizzare un unico spazio logico di indirizzamento sia per l'host che per la GPU, CUDA si occuperà di rendere trasparente il trasferimento.

6.4.1 Somma fra Vettori in CUDA

Si vuole fare la somma di due array di interi composti da n elementi, verrà creato un thread per ogni elemento, che si occuperà di tale somma.



Nota : avviare tanti thread quanti sono gli elementi da sommare sarebbe impensabile su una CPU, e renderebbe la computazione estremamente lenta dovuta ai molteplici cambi di contesto. D'altro canto, per le motivazioni precedentemente date, ha senso creare tanti thread quando questi dovranno essere eseguiti sulla GPU, inoltre, ogni thread dovrà eseguire le stesse istruzioni, quindi non dovrebbe esserci warp divergence.

```

1 //h_A, h_B, h_C sono i vettori definiti sulla RAM
2 void vecAdd(float* h_A, float* h_B, *float h_C, int n){
3     float *d_A, *d_B, *d_C; //definisco i puntatori per la VRAM
4
5     //alloco e copio i dati dalla RAM alla VRAM
6     cudaMalloc((void**) &d_A, n*sizeof(float));
7     cudaMemcpy(d_A,h_A, n*sizeof(float), cudaMemcpyHostToDevice);
8
9     cudaMalloc((void**) &d_B, n*sizeof(float));
10    cudaMemcpy(d_B,h_B, n*sizeof(float), cudaMemcpyHostToDevice);
11
12    cudaMalloc((void**) &d_C, n*sizeof(float));
13
14    //Chiamata funzione kernel
15    vecAddKernel<<<ceil(n/256.0) , 256>>>(d_A,d_B,d_C,n);
16    //ceil ritorna l'intero successivo
17
18    cudaDeviceSynchronize();
19
20    //copio il risultato dalla VRAM alla RAM
21    cudaMemcpy(h_C,d_C, n*sizeof(float), cudaMemcpyDeviceToHost);
22
23    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
24 }

```

Si avvia la funzione kernel `vecAddKernel` con $\frac{n}{256}$ blocchi, e 256 thread per blocco, per un totale di n thread. Ogni thread, dovrà identificarsi tramite il suo id, ed in base a questo, capire su quale elemento del vettore fare la somma.

```

1 __global__
2 void vecAddKernel( float* A, float* B, float *C, int n){
3     int index = blockDim.x*blockIdx.x+threadIdx.x; //id univoco del thread
4     if(index<n) C[index]=A[index]+B[index];
5 }

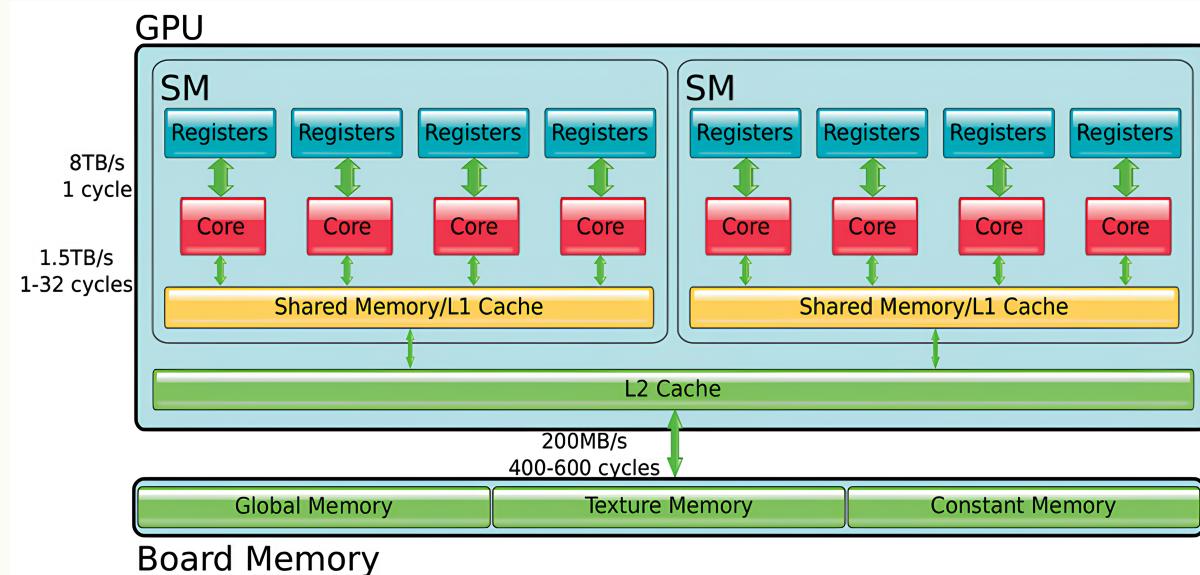
```

Perché controllare che l'indice sia minore di n ? Se il numero di thread non è divisibile per 256, ci saranno più di $\frac{n}{256}$ blocchi, quindi ci saranno più di n thread, ed alcuni (quelli con indice maggiore o uguale ad n) non devono partecipare alla computazione.

- Esempio : array da 1281 elementi
- si creeranno $\lceil \frac{1281}{256} \rceil = 6$ blocchi
- ci saranno 256 thread per blocco, in totale ci saranno $6 \times 256 = 1536$ thread
- 255 thread più del necessario, che non saranno utilizzati

6.4.2 Tipi di Memoria

Ci sono differenti tipi di memoria in una GPU



- **Registri** : classici registri come quelli della CPU, sono privati per ogni thread e servono per contenere le variabili locali sulla quale si eseguono le istruzioni.
- **Shared Memory** : è un tipo di memoria separata dalla VRAM (globale), va gestita dal programmatore in maniera esplicita, e non è trasparente come la cache, nonostante si trovi sullo stesso chip. La particolarità, è che è condivisa fra i core di uno stesso SM, quindi, due processori di un SM dovrebbero comunicare scrivendo sulla shared memory piuttosto che sulla memoria globale.
- **Global Memory** : la VRAM di cui si è parlato fin'ora, è la memoria più capiente, condivisa fra tutti i core.

Le variabili, per essere allocate su una memoria piuttosto che su un'altra, vanno precedute da appositi decoratori.

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
<code>_device_ __shared__ int SharedVar;</code>	Shared	Block	Kernel
<code>_device_ int GlobalVar;</code>	Global	Grid	Application
<code>_device_ __constant__ int ConstVar;</code>	Constant	Grid	Application

I registri vengono usati (localmente) dai thread per salvare le variabili, i registri su un core sono condivisi dai thread che vi saranno schedulati. La compute capability determina il massimo numero di registri disponibili per thread, quando si eccede tale numero nell'allocazione, le variabili verranno allocate nella memoria globale (ed il loro accesso sarà più lento). La decisione di quali variabili vanno nei registri, e quali vanno nella memoria globale, spetta al compilatore.

Occupancy

Il numero di registri disponibili per ogni thread influenza il numero di thread massimi che possono essere schedulati su un'unico SM, si consideri il seguente esempio

- una funzione kernel, utilizza 48 registri
- viene eseguita sulla GPU, i blocchi logici sono composti da 256 thread
- ogni thread (che esegue la funzione kernel) necessita di 48 registri, quindi ogni blocco (che ha 256 thread) necessita di $48 \times 256 = 12\,288$ registri.
- si assuma che la GPU in questione ha 32 000 registri per ogni SM.
- si assuma che la GPU in questione supporta al più 1536 thread per ogni SM.
- ogni SM potrebbe potenzialmente ospitare due blocchi, dato che per due blocchi sono necessari $12\,288 \times 2 = 24\,576$ registri, che sono meno dei 32 000 totali.
- quindi, in base ai limiti dei registri disponibili, ogni SM ospiterebbe al più 2 blocchi. Essendo che un blocco ha 256 thread, ogni SM ospiterebbe al più 512 thread.
- potenzialmente però, un SM supporta 1536 thread, quindi c'è uno spreco di efficienza.

NVIDIA definisce l'*occupancy* come il rapporto fra i warp schedulati su un SM ed il massimo numero possibile di warp schedulabili su un SM.

$$\text{occupancy} = \frac{\text{numero di warp attuali}}{\text{max numero di warp}}$$

nell'esempio riportato, assumendo che un warp contenga 32 thread:

$$\text{occupancy} = \frac{2^{\frac{256}{32}}}{\frac{1536}{32}} = \frac{16}{48} = 33.3\%$$

quasi due terzi dei thread disponibili non sono utilizzati. L'ideale sarebbe un *occupancy* del 100%.

Shared Memory

La shared memory è una memoria on-chip presente su ogni SM, come già accennato, il suo contenuto è condiviso fra tutti i thread che risiedono su uno stesso SM. L'accesso è rapido quanto quello su una cache L1, in quanto sono costruiti sullo stesso chip, la shared memory è però gestita dall'utente che scrive il programma, ed è logicamente distinta dalla cache. È utilizzabile per

- salvare dati che sono frequentemente utilizzati, per risparmiare tempo negli accessi
- comunicare fra thread di uno stesso SM

Quando si dichiara una variabile, per specificare che essa va salvata sulla shared memory, deve essere preceduta dal decoratore `__shared__`. Nonostante la cache si occupa già di rendere più rapido l'accesso frequente ad uno stesso dato, la gestione manuale con la shared memory può risultare più efficiente.

Esempio (stencil)

Vogliamo prendere un array in input e creare un array di output in cui ogni elemento con indice i nell'array di output sia la somma di tutti gli elementi con indice da $i - 3$ a $i + 3$. Per fare ciò in modo efficiente conviene che ogni blocco carichi in memoria condivisa la parte di array che andrà a utilizzare, inoltre i thread ai bordi dovranno caricare anche gli elementi ai lati il cui indice viene calcolato da un altro blocco.

```

1  __global__ void stencil_1d ( int * in , int * out ){
2      __shared__ int temp [ BLOCK_SIZE + 2*3];
3      int gindex = threadIdx . x + blockIdx . x * blockDim . x ;
4      int lindex = threadIdx . x + 3;
5      temp [ lindex ] = in [ gindex ];
6      if( threadIdx . x <3){
7          temp [ lindex - 3] = in [ gindex - 3];
8          temp [ lindex + BLOCK_SIZE ] = in [ gindex + BLOCK_SIZE ];
9      }
10     int result = 0;
11     for( int offset = -3; offset <= 3; offset ++){
12         result += temp [ lindex + offset ];
13     }
14     out [ gindex ]= result ;
15 }
```

Con questo codice potrebbe comunque esserci un errore nel caso un warp all'interno del blocco inizi ad eseguire il for prima che tutti i warp nel blocco abbiano caricato i valori nella memoria condivisa, per risolvere bisogna usare una barriera, che si scrive:

```
__syncthreads();
```

La shared memory è divisa in banchi con gli indirizzi assegnati ai banchi in modo ciclico, ad esempio con 16 banchi:

Address	Bank															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
	128	132	136	140	144	148	152	156	160	164	168	172	176	180	184	188

6.4.3 Constant Memory

La **memoria costante** è una memoria on-chip che può contenere esclusivamente valori costanti definiti dall'host, su cui i cuda CORE possono accedere esclusivamente in lettura, supporta inoltre una specifica funzione di condivisione di un valore a tutti i thread di un warp. Si considerino i seguenti scenari:

- dei dati sono salvati in memoria globale
 - tutti i warp richiedono lo stesso segmento dalla memoria globale

- il segmento viene (quando richiesto per la prima volta) caricato in cache L2
- con alta probabilità altri dati passeranno per la cache, e gli stessi dati richiesti dai warp dovranno essere trasferiti dalla memoria globale più volte
- dei dati sono salvati nella memoria costante
 - durante la prima richiesta di warp, i dati vengono copiati in una cache speciale della memoria costante
 - poiché c'è meno traffico nella constant-cache, ci sono buone probabilità che tutti gli altri warp troveranno i dati già nella cache

6.4.4 Global Memory

Se due thread accedono a due banchi di memoria diversi (nella memoria globale) l'accesso è istantaneo, se invece accedono allo stesso banco gli accessi saranno serializzati, obbligando uno dei due thread ad aspettare che il primo abbia finito. Nel caso venisse però fatto un accesso allo stesso indirizzo da parte di più di un thread verrebbe fatta una broadcast del valore.

Quando viene caricato un valore dalla memoria globale, viene caricato quello che viene chiamato un burst, cioè un insieme di indirizzi consecutivi a quello caricato e se tutti thread in un warp accedono a locazioni di memoria consecutive l'hardware esegue degli accessi **coalizzati**, cioè li tratta come un unico accesso.

In base a se l'accesso alla memoria globale è cachato o no cambia anche la dimensione del burst che viene caricato (128 byte per cache L1, 32 byte per L2 o memoria globale), per questo in alcuni casi potrebbe essere conveniente disattivare la cache L1 se i nostri thread eseguono tutti accessi in locazioni lontane tra loro per evitare di caricare molti valori inutili.

6.4.5 Memoria Pinnata

I trasferimenti di dati tra CPU e GPU vengono fatti attraverso un **DMA (Direct Memory Access)** che è una parte di hardware specializzata nel mandare dati richiesti dal sistema operativo e che permette di liberare la CPU per fare altri task. I sistemi moderni utilizzano la memoria virtuale, in cui più indirizzi virtuali sono mappati sullo stesso indirizzo fisico. Ogni spazio degli indirizzi virtuale è diviso in pagine che vengono caricate nella memoria fisica ed eventualmente tolte per far spazio ad altre pagine nello stesso indirizzo fisico.

Il DMA utilizza un indirizzo fisico, questo implica che non sia sicuro che la determinata pagina virtuale che stavamo cercando inviare alla GPU sia ancora in quell'indirizzo fisico, ma potrebbe essere stata sostituita da un'altra. Per evitare che quindi vengano copiati dati sbagliati la `cudaMemcpy` copia prima i dati nella **memoria pinnata**, cioè un insieme di pagine virtuali che non possono essere tolte dalla memoria fisica. Questa operazione è molto lenta, quindi per ottimizzarla converrebbe salvare prima di eseguire la `cudaMemcpy` i dati nella memoria pinnata, evitando un'ulteriore copia. È possibile allocare un determinato indirizzo di memoria nella memoria pinnata tramite la funzione `mlock` dopo aver eseguito una `malloc`, poi per liberare la memoria bisognerà usare la funzione `munlock` seguito da `free`.

♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪ ♪♫♪

6.5 Stima delle Performance

Le performance di una GPU si misurano in FLOP/s, cioè operazioni a virgola mobile al secondo. Possiamo fare una stima di quanto stiamo saturando l'hardware conoscendo la banda della memoria globale e il massimo numero di FLOP/s della GPU.

Esempio

Se abbiamo una GPU con memoria globale con banda di 200GB/s e carichiamo degli interi (4 bytes), possiamo caricare al massimo $50G = 50 \cdot 10^9$ operandi al secondo. Se eseguissimo una sola operazione

con ogni operando, potremmo eseguire 50GFLOP/s, che paragonata al massimo della GPU (ad esempio 1500GFLOP/s) ci dà una stima di quanto stiamo saturando l'hardware, in questo caso:

$$\frac{50\text{GFLOP/s}}{1500\text{GFLOP/s}} = 3.3\%$$

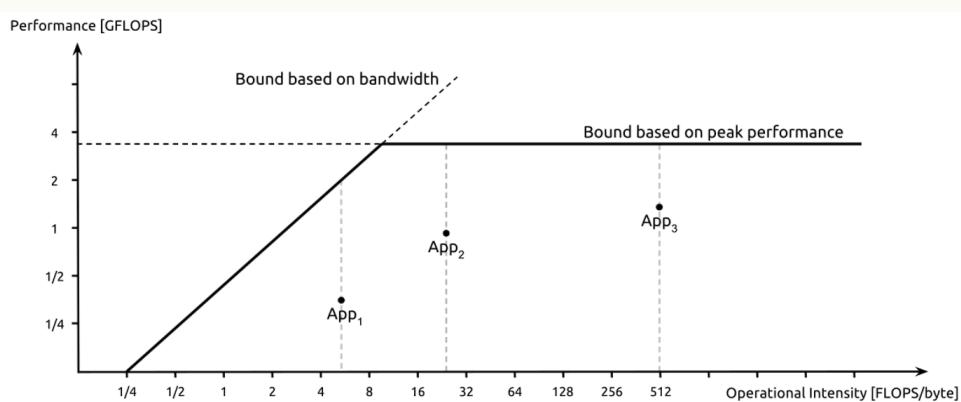
Per riuscire a saturare al massimo la GPU dovremmo fare:

$$\frac{1500\text{GFLOP/s}}{50\text{GFLOP/s}} = 30$$

quindi 30 operazioni per ogni operando caricato.

6.5.1 Roofline Model

Il Roofline model è un grafico che mostra le performance massime di una GPU basandosi sulla banda della memoria globale e il massimo di FLOP/s della GPU. Il grafico ha questa forma:



Nel grafico viene mostrato il bound dovuto alla banda della memoria globale (la riga obliqua) e il bound dovuto alla massima potenza della GPU stessa, ogni programma che viene eseguito su questa GPU sarà sempre al di sotto della riga.

6.5.2 MPI e CUDA

Se abbiamo più di una GPU che lavorano insieme, si utilizza solitamente MPI, associando ad ogni GPU un processo MPI, per poter passare i dati da una GPU all'altra. Come venga fatto dipende dal fatto se MPI sia **GPU-aware**:

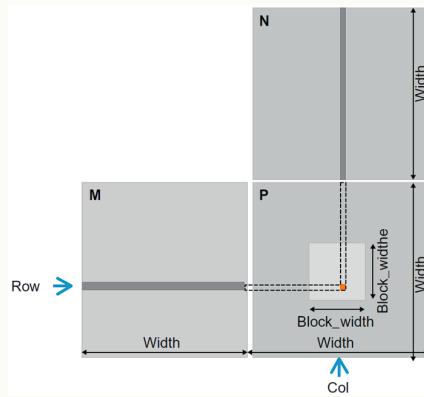
- MPI GPU-aware: si può accedere al buffer della GPU direttamente, quindi i puntatori ad una locazione di memoria della GPU possono essere usati nelle chiamate MPI.
- MPI non GPU-aware: bisogna trasferire i dati dalla memoria della GPU a quella dell'host prima di essere mandati al processo MPI associato all'altra GPU, che li copierà nella memoria della propria GPU.

☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞ ☞☞☞

6.6 Tiling

Si consideri il problema della moltiplicazione fra matrici, dove l'elemento (i, j) della matrice risultante è dato dal prodotto scalare del vettore riga i della prima matrice con il vettore colonna j della seconda matrice.

Si può rappresentare graficamente come mostrato nella seguente figura



Una semplice implementazione è la seguente

```

1  __global__ void MatrixMulKernel ( float* M, float* N, float* P, int Width ) {
2      // Calculate the row index of the P element and M
3      int Row=blockIdx.y*blockDim.y+threadIdx.y;
4      // Calculate the column index of P and N
5      int Col=blockIdx.x*blockDim.x+threadIdx.x;
6      if ((Row < Width) && (Col < Width)) {
7          float Pvalue=0;
8          // each thread computes one element of the block sub-matrix
9          for ( int k=0; k < Width; ++k)
10             Pvalue += M[Row*Width+k] *N[k*Width+Col];
11             P[Row*Width+Col]=Pvalue;
12     }
13 }
```

In questo modo ogni thread si occupa di un elemento della matrice risultante. Si consideri la decima linea di codice

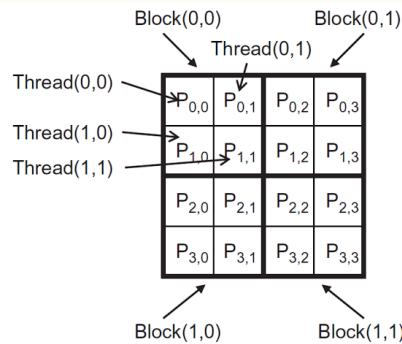
`Pvalue += M[Row*Width+k] *N[k*Width+Col];`

Essa coinvolge due accessi in memoria globale, uno per la matrice M ed uno per la matrice N, inoltre coinvolge due operazioni, un'addizione ed una moltiplicazione.

- per ogni operando coinvolto si esegue un'operazione in virgola mobile
- si accede a due valori per le due matrici composte da `float` (8 byte) quindi l'intensità di operazioni aritmetiche è di $8 \frac{\text{FLOP}}{\text{byte}}$.

Si vuole sfruttare la shared memory per aumentare il numero di operazioni in virgola mobile per byte.

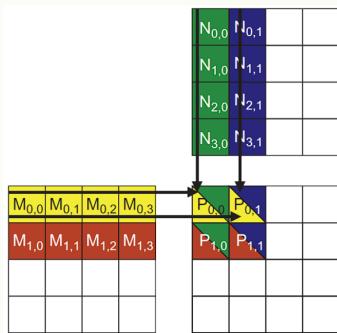
Assumiamo che la dimensione dei blocchi logici della struttura di CUDA sia 2×2 , quindi un blocco contiene 4 thread, ed assumiamo che le matrici coinvolte siano 4×4



Ci sono 4 blocchi = 16 thread, un thread per ogni elemento della matrice. È necessaria un'osservazione

- il thread che si occupa dell'elemento $(0, 0)$ accede alla riga 0 di M e alla colonna 0 di N
- il thread che si occupa dell'elemento $(1, 0)$, è contenuto nello stesso blocco del thread precedente, ed accede alla riga 1 di M e alla colonna 0 di N
- il thread che si occupa dell'elemento $(0, 1)$, è contenuto nello stesso blocco del thread precedente, ed accede alla riga 0 di M e alla colonna 1 di N

- il thread che si occupa dell'elemento (1, 1), è contenuto nello stesso blocco del thread precedente, ed accede alla riga 1 di M e alla colonna 1 di N

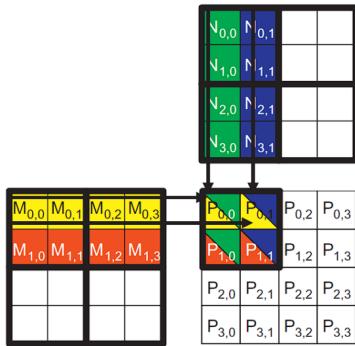


I thread di uno stesso blocco **accedono contemporaneamente ad elementi in comune**, in particolare, una stessa riga (o una stessa colonna) deve sempre essere acceduta da due thread di uno stesso blocco. Essendo che i thread di uno stesso blocco condividono la shared memory, è opportuno far sì che un thread si occupi di caricare una riga (o colonna) sulla shared memory, mentre l'altro può limitarsi a leggerla da quest'ultima, riducendo il traffico sulla memoria globale. Più i blocchi sono grandi, più il traffico può essere ridotto.

Access order →

thread _{0,0}	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread _{0,1}	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread _{1,0}	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread _{1,1}	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

Il **tiling** è una tecnica fondamentale nell'elaborazione di matrici, specialmente quando si tratta di operazioni complesse come la moltiplicazione. L'idea di base è dividere le matrici in blocchi più piccoli, chiamati "tiles" (o "mattoni"), per semplificare l'elaborazione e ottimizzare le prestazioni.



Per la prima tile della matrice, ogni thread carica una riga dalla prima tile di M ed una colonna dalla prima tile di N dalla memoria globale alla shared memory, si esegue poi una `__syncthreads()`, a questo punto i thread eseguiranno il prodotto riga per colonna avendo tutti i valori necessari nella shared memory.

time →

	Phase 1		Phase 2		
thread _{0,0}	$M_{0,0}$ ↓ $Mds_{0,0}$	$N_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{0,1} * Nds_{1,0}$	$M_{0,2}$ ↓ $Mds_{0,0}$	$N_{2,0}$ ↓ $Nds_{0,0}$
thread _{0,1}	$M_{0,1}$ ↓ $Mds_{0,1}$	$N_{0,1}$ ↓ $Nds_{0,1}$	$PValue_{0,1} +=$ $Mds_{0,0} * Nds_{0,1} +$ $Mds_{0,1} * Nds_{1,1}$	$M_{0,3}$ ↓ $Mds_{0,1}$	$N_{2,1}$ ↓ $Nds_{0,1}$
thread _{1,0}	$M_{1,0}$ ↓ $Mds_{1,0}$	$N_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{1,0} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{1,0}$	$M_{1,2}$ ↓ $Mds_{1,0}$	$N_{3,0}$ ↓ $Nds_{1,0}$
thread _{1,1}	$M_{1,1}$ ↓ $Mds_{1,1}$	$N_{1,1}$ ↓ $Nds_{1,1}$	$PValue_{1,1} +=$ $Mds_{1,0} * Nds_{0,1} +$ $Mds_{1,1} * Nds_{1,1}$	$M_{1,3}$ ↓ $Mds_{1,1}$	$N_{3,1}$ ↓ $Nds_{1,1}$

In seguito è riportata l'implementazione della moltiplicazione fra matrici che incorpora il tiling per ridurre gli accessi in memoria.

```

1  __global__ void MatrixMulKernel( float* M, float* N, float* P, int Width){
2      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
3      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
4
5      int bx = blockIdx.x; int by = blockIdx.y;
6      int tx = threadIdx.x; int ty = threadIdx.y;
7
8      // Identifica la riga e la colonna dell'elemento d_P su cui lavorare
9      int Row = by * TILE_WIDTH + ty;
10     int Col = bx * TILE_WIDTH + tx;
11
12     float Pvalue = 0;
13
14     // Ciclo sui tile d_M e d_N necessari per calcolare l'elemento d_P
15     for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {
16         // Caricamento collaborativo dei tile d_M e d_N nella memoria condivisa
17         Mds[ty][tx] = d_M[Row * Width + ph * TILE_WIDTH + tx];
18         Nds[ty][tx] = d_N[(ph * TILE_WIDTH + ty) * Width + Col];
19
20         __syncthreads(); // Sincronizzazione dei thread del blocco
21
22         // Moltiplicazione dei tile e accumulo del risultato parziale
23         for (int k = 0; k < TILE_WIDTH; ++k)
24             Pvalue += Mds[ty][k] * Nds[k][tx];
25         __syncthreads(); // Sincronizzazione dei thread del blocco
26     }
27
28     // Scrittura del risultato finale nella memoria globale
29     d_P[Row * Width + Col] = Pvalue;
30 }
```