

Programmazione Multicore

Alessio Marini, 2122855

Appunti presi durante il corso di **Programmazione Multicore** nell'anno **2025/2026** del professore Daniele De sensi.

Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni .

Contatti:

 alem1105

 marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1.	Parallel Computing	3
1.1.	Type of Parallel Systems	5
2.	Distributed memory programming with MPI	9
2.1.	Communicators	10
2.2.	Point-to-Point Communication Modes	15
2.3.	Non-Blocking Communication	16
3.	Parallel Program Design	18
3.1.	Foster's Methodology	18
4.	Parallel Design Patterns	19
4.1.	Single Program Multiple Data	19
4.2.	Multiple Program Multiple Data	19
4.3.	Master-Worker	20
4.4.	Map-Reduce	20
4.5.	Fork/Join	20
4.6.	Loop Parallelism	20
4.7.	Collective Communication	23
5.	Performance Evaluation	25
5.1.	Matrici	31
6.	Derived Datatypes	33
7.	Shared Memory Programming with OpenMP	35
7.1.	Reduction Clause	37
7.2.	Parallel For	41
7.3.	Nested Loops	44
8.	GPU Programming	45
8.1.	Cuda	46
8.1.1.	Scrivere Programmi Cuda	47
8.1.2.	Otttenere la posizione dei thread	48
8.1.3.	Thread Scheduling	49
8.1.4.	Warps	50

1. Parallel Computing

Dal 1986 al 2003 le velocità dei microprocessori aumentava di un 50% all'anno, ovvero un 60x in 10 anni. Dal 2003 in poi però questo incremento ha iniziato a rallentare, ad esempio dal 2015 al 2017 c'è stato soltanto un 4% all'anno ovvero un 1.5x in 10 anni.

Ci sono motivi fisici dietro a questo fenomeno e per questo, invece di continuare a costruire processori più potenti abbiamo iniziato ad inserire più processori all'interno dello stesso circuito.

I programmi seriali ovviamente non sfruttano questi benefici e continuano a venir eseguiti in un singolo processore, anche se ovviamente significa che possiamo eseguirli in più istanze contemporaneamente. Sono quindi i programmatori che devono sapere come utilizzare queste tecnologie per scrivere programmi che le sfruttano.

Ci serve tutta questa efficienza?

Per la maggior parte dei programmi no ma esistono dei campi di ricerca che dove c'è bisogno di eseguire tantissime operazioni in poco tempo o comunque su tantissimi dati, ad esempio:

- LLMs
- Decoding the human genome
- Climate modeling
- Protein folding
- Drug discovery
- Energy research
- Fundamental physics

Il motivo fisico che abbiamo accennato prima del perché costruiamo questi sistemi paralleli è dovuto al fatto che le performance di un processore aumentano con l'aumentare della densità di transistor che ha, questo comporta alcune cose:

- Transistor più piccoli -> Processori più veloci
- Processori più veloci -> Aumenta il loro consumo energetico
- Consumo più alto -> Aumenta la temperatura del processore
- Temperatura alta -> Comportamenti del processore inaspettati

Quindi anche se alcuni programmi possiamo eseguirli in più istanze e aumentare la loro efficienza spesso non è la strada migliore e dobbiamo imparare a scrivere del codice che usa la parallelizzazione. Con il tempo sono stati scoperti dei «pattern» facilmente convertibili in codice parallelo ma spesso la strada migliore è quella di fare un passo indietro e ripensare un nuovo algoritmo. Non sempre saremo in grado di parallelizzare completamente il codice.

Esempio

Codice Seriale - Compute n values and add them together

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

C

Codice Parallello - abbiamo p cores dove $p \ll n$ e ogni core calcola la somma di $\frac{n}{p}$ valori

```
1 my_sum = 0;
2 my_first_i = ... ;
3 my_last_i = ... ;
4 for (my_i = my_first_i; my_i < my_last_i; my_i++) {
5     my_x = Compute_next_value(...);
6     my_sum += my_x;
7 }
```

C

Quindi ogni core usa delle variabili per memorizzare il suo primo e ultimo valore da sommare, in questo modo ogni core può eseguire del codice indipendentemente dagli altri core.

Ad esempio se abbiamo 8 core e 24 valori, ogni core somma 3 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

E quindi avremo come somme:

- 8, 19, 7, 15, 7, 13, 12, 14

Ci basta quindi sommare la somma di tutti i core e ottenere il risultato finale.

Però usando questa soluzione, nello step finale abbiamo che un core solo (il principale) effettua la somma dei risultati degli altri, mentre loro appunto non stanno facendo nulla.

Per risolvere questo possiamo ad esempio accoppiare il core 0 con il core 1 e fare in modo che il core 0 sommi al suo risultato quello del core 1, poi possiamo fare lo stesso lavoro con il 2 e il 3 ed avere quindi nel core 2 la somma di 2 e 3 ecc...

Ripetiamo tutto questo accoppiando 0-2, 4-6... e poi continuiamo a ripetere accoppiando 0-4...

Vediamo graficamente:

TODO: image

Se confrontiamo i due metodi:

TODO: image

Notiamo che con il primo metodo, quello a sinistra, se abbiamo 8 cores facciamo 7 somme aggiuntive, in generale $p - 1$ somme.

Con il secondo metodo se abbiamo 8 cores facciamo 3 somme aggiuntive (sono sempre 7 ma sono parallele e avvengono nello stesso momento), in generale abbiamo $\log_2(p)$ somme.

Quindi se ad esempio avessimo avuto $p = 1000$ con il primo metodo avremmo avuto 999 somme mentre con il secondo soltanto 10.

Come scriviamo codice parallelo?

- **Task Parallelism:** Dividere alcune task fra i cores. L'idea è quella di eseguire compiti diversi in parallelo.
- **Data Parallelism:** Partizionare i dati fra i cores, fargli risolvere operazioni simili e risolvere il problema raccogliendo i dati. In generale quando i cores svolgono la stessa operazione ma su pezzi di dati diversi.

Esempio

Devo valutare 300 esami da 15 domande ciascuno e ho 3 assistenti:

- Data Parallelism: Ogni assistente valuta 100 esami
- Task Parallelism:
 - L'assistente 1 valuta tutti gli esami ma soltanto le domande 1-5
 - L'assistente 2 valuta tutti gli esami ma soltanto le domande 6-10
 - L'assistente 3 valuta tutti gli esami ma soltanto le domande 11-15

L'esempio che abbiamo fatto precedentemente con le somme dei vari cores, è stato parallelizzato on Data o Task Parallelism?

TODO: image

Se ogni core può lavorare in modo indipendente dagli altri allora la scrittura del codice sarà molto simile a quella di un programma seriale. In generale dobbiamo coordinare i cores, questo perché:

- Communication: Ad esempio perché ogni core manda una somma parziale ad un altro
- Load Balancing: Nessun core deve svolgere troppo lavoro in più rispetto ad altri perché altrimenti qualche core dovrà aspettare che alcuni finiscano e questo significa perdita di risorse e potenza.
- Synchronization: Ogni core lavora al suo ritmo ma dobbiamo assicurarci che nessuno vada troppo avanti. Ad esempio se un core compila una lista dei file da comprimere ed i cores che comprimono partono troppo presto potrebbero perdersi qualche file.

Noi scriveremo codice esplicitamente parallelo usando 4 diverse estensioni delle API di C:

- Message-Passing Interface (MPI) [Library]
- Posix Threads (Pthreads) [Library]
- OpenMP [Library + Compiler]
- CUDA [Library + Compiler]

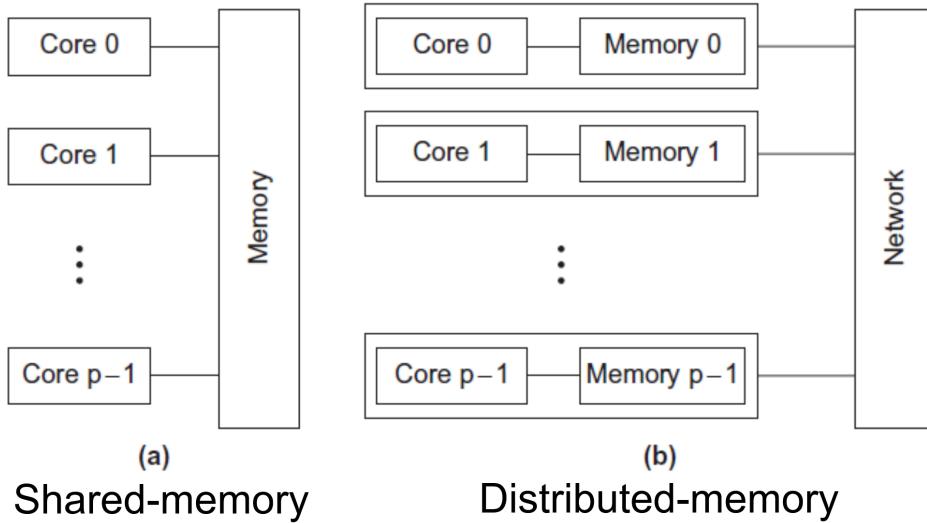
Useremo anche librerie ad alto livello già esistenti che però hanno un compresso per quanto riguarda facilità di utilizzo e performance.

1.1. Type of Parallel Systems

- **Shared Memory:** I core lavorano sulla stessa memoria e si coordinano leggendo una specifica zona di memoria.

- **Distributed Memory**: Ogni core ha la sua memoria dedicata e per coordinarsi si scambiano messaggi su una rete dedicata.

In generale quindi devono comunque avere un modo per coordinarsi.



- **Multiple-Instruction Multiple-Data (MIMD)**: Ogni core ha la sua unità di controllo e può lavorare indipendentemente dagli altri. Come ad esempio la CPU classica dei PC, ogni core può fare qualcosa di diverso.
- **Single-Instruction Multiple-Data (SIMD)**: Ogni core può lavorare su un pezzo di dato diverso, ma tutti devono lavorare per la stessa istruzione. Ad esempio possono lavorare tutti su un vettore ma ognuno su una parte di vettore diversa.

How will we write parallel programs?

	Shared Memory	Distributed Memory
SIMD	CUDA	
MIMD	Pthreads/ OpenMP/ CUDA	MPI

Concurrent

Diverse tasks possono essere in esecuzione in ogni momento. Questi possono essere anche seriali, ad esempio dei sistemi operativi su un singolo core.

Parallel

Diverse tasks possono **collaborare** per risolvere lo **stesso problema**.

Distributed

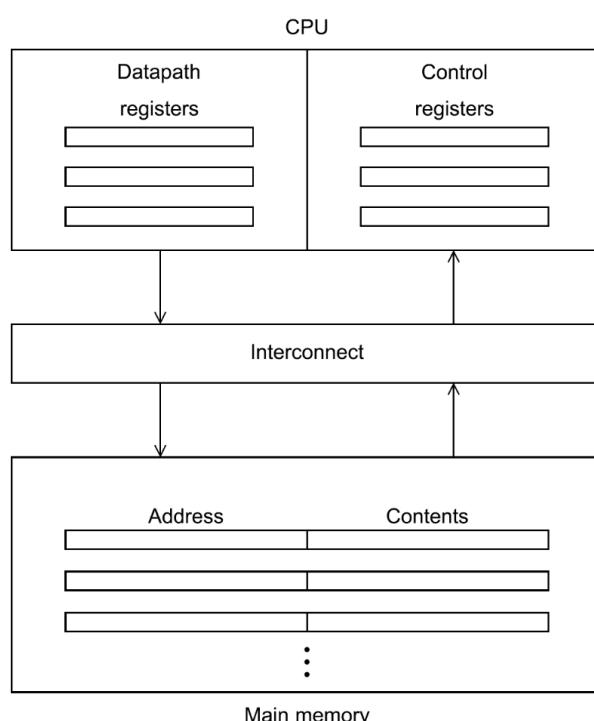
Il lavoro viene diviso su macchine separate che comunicano tramite una rete. Il sistema appare come sistema unico.

Da notare che i sistemi paralleli e distribuiti sono comunque concorrenti, infatti molte attività vengono svolte nello stesso momento e si contendono le risorse del sistema.

- **Von Neumann architecture**

Di solito quando programmiamo non ci preoccupiamo di come è fatto l'hardware su cui stiamo lavorando, di solito possiamo astrarre. Però se vogliamo scrivere codice efficiente ci tornerà utile capire come funziona l'hardware e scrivere codice ottimizzato per quell'hardware.

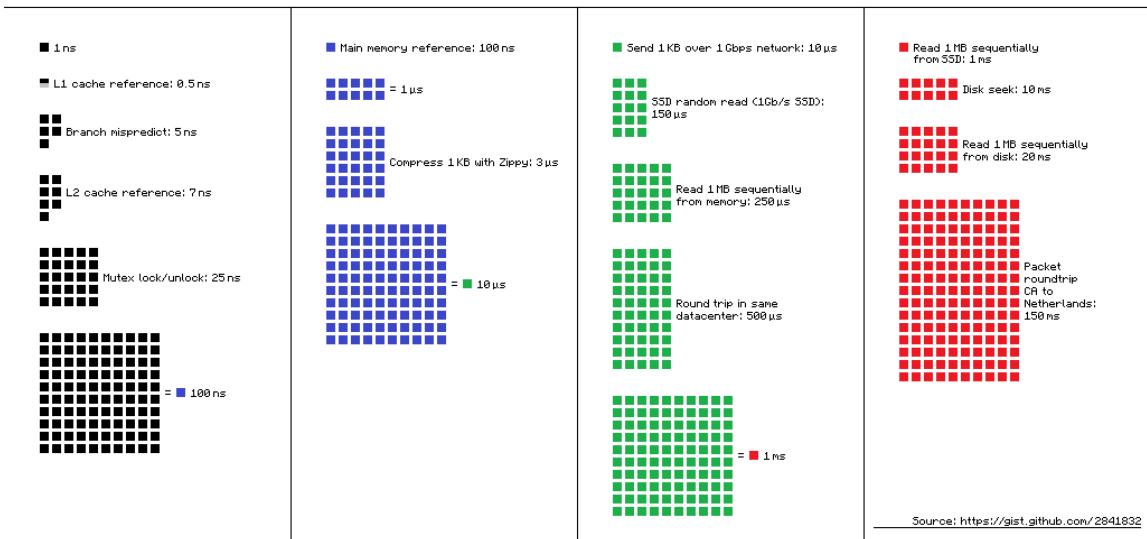
Vediamo la **Von Neumann Architecture**



- **Main Memory:** È formata da tante locazioni di memoria che contengono dei dati, ciascuna è identificata da un **indirizzo**
- **CPU:** Esegue le istruzioni e decide quali eseguire. I registri sono delle memorie estremamente veloci ma anche molto piccole che servono a memorizzare dati importanti come lo stato d'esecuzione dei processi. Ad esempio uno dei più importanti è il **Program Counter (PC)** che contiene l'indirizzo della prossima istruzione da eseguire.
- **Interconnect:** Serve a mettere in comunicazione la memoria e la CPU, di solito è un **bus di sistema** ma in alcuni casi potrebbe essere più complesso.

Una macchina che segue il modello di Von Neumann esegue un'istruzione alla volta, ogni istruzione opera su una piccola parte di dati che vengono memorizzati nei registri. La CPU può leggere e scrivere dati nella memoria ma la separazione di queste due componenti causa quello che viene chiamato **Von Neumann Bottleneck**, infatti il bus di sistema o in generale la connessione fra i componenti determinate la velocità di trasferimento dei dati, di solito è più lenta sia della memoria che della CPU.

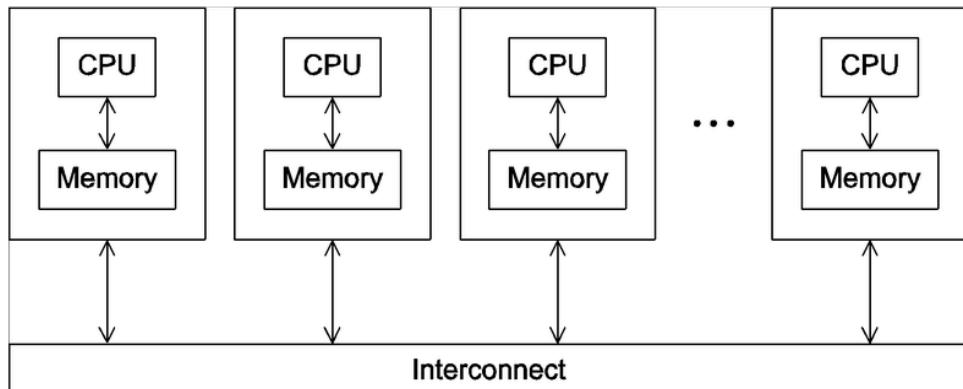
Una comparazione



Assumendo 1GB/sec SSD

2. Distributed memory programming with MPI

Possiamo astrarre un **distributed memory system** in questo modo:



Single-Program Multiple-Data

Noi **compiliamo un solo programma** ma poi questo verrà eseguito da più processi.

Useremo un **if-else** per cambiare il comportamento dei processi quindi, ad esempio:

- Se sono il processo 0 faccio X
- Altrimenti faccio Y

Ricordiamo che i processi non condividono memoria quindi la comunicazione avviene tramite **message passing**.

Per programmare con la libreria **MPI** abbiamo bisogno di aggiungere l'header `mpi.h` e usare degli identificatori, di solito iniziano con `MPI_`.

Esempio

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(void) {
5     MPI_Init(NULL, NULL);
6     printf("hello, world\n");
7     MPI_Finalize();
8     return 0;
9 }
```



- `MPI_Init` inizializza `mpi` per tutto il necessario, possiamo passare dei puntatori ai parametri del main, se non ci servono possiamo anche passare `NULL`, come l'esempio sopra.
- `MPI_Finalize` fa capire a `mpi` che il programma è finito e può pulire tutta la memoria allocata.

Il programma non deve trovarsi tutto all'interno di questo costrutto ma è importante che ci sia la parte che vogliamo venga eseguita da più cores.

Compilazione

```
1 mpicc -g -Wall -o mpi_hello mpi_hello.c
```

Bash

- `mpicc` è il compilatore
- `-g` fa visualizzare delle informazioni di debug
- `-Wall` attiva tutti gli avvisi
- `-o` specifica il nome del file di output

Esecuzione

```
1 mpiexec -n <number of processes> <executable>
```

Bash

Serve per eseguire il programma e ci permette di specificare con quanti core lanciarlo.

Debugging

```
1 mpiexec -n 4 ./test : -n 1 ddd ./test : -n 1 ./test
```

Bash

Debuggare programmi non seriali è più complicato, ad esempio se eseguiamo lo stesso programma con un solo processo il problema potrebbe non verificarsi mentre se lo eseguiamo con un certo numero di core si.

Con il comando sopra possiamo lanciare 5 processi, il quinto verrà eseguito con il debugger, che si chiama `ddd`.

2.1. Communicators

Sono dei processi che possono mandarsi dei messaggi fra di loro, quando chiamiamo `MPI_Init` viene inizializzato anche un comunicatore per tutti i processi, questo è chiamato `MPI_COMM_WORLD`.

Dopo un `INIT` possiamo «catturare» l'identificatore di ogni processo e anche il numero totale, attraverso:

```
1 int MPI_Comm_size(
2     MPI_Comm    comm      /* in */,
3     int*        comm_sz_p /* out */
4 );
```

C C

```
1 int MPI_Comm_rank(
2     MPI_Comm    comm      /* in */,
3     int*        my_rank_p /* out */
4 );
```

C C

Prendono in input il comunicatore e un puntatore ad interno e salvano in questi interi i valori di, rispettivamente, quanti processi ci sono e il grado del processo attuale.

Esempio di utilizzo

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(void) {
5     int comm_sz, my_rank;
6     MPI_Init(NULL, NULL);
7     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     printf("hello, world from process %d out of %d\n", my_rank, comm_sz);
10    MPI_Finalize();
11    return 0;
12 }

```

C C

Se eseguiamo questo programma noteremo che ad ogni esecuzione l'ordine dei print cambia, questo perchè non sappiamo con esattezza quale processo finirà per primo quindi otterremo sempre un ordine diverso. Possiamo cambiare questo comportamento facendo comunicare i processi.

Per far comunicare i processi utilizziamo due funzioni `MPI_Send` e `MPI_Recv`:

```

1 int MPI_Send(
2     void*           msg_buf_p      /* in */,
3     int             msg_size       /* in */,
4     MPI_Datatype   msg_type       /* in */,
5     int             dest          /* in */,
6     int             tag           /* in */,
7     MPI_Comm        communicator /* in */
8 );
9
10 int MPI_Recv(
11    void*           msg_buf_p      /* out */,
12    int             buf_size       /* in */,
13    MPI_Datatype   buf_type       /* in */,
14    int             source         /* in */,
15    int             tag           /* in */,
16    MPI_Comm        communicator /* in */,
17    MPI_Status*     status_p       /* in */
18 );

```

C C

Da notare che `msg_size` e `buf_size` indicano il numero di elementi e non il numero di byte.

Il campo `status_p` di tipo `MPI_Status*` è un puntatore ad una struttura composta in questo modo:

```

1 typedef struct MPI_Status {
2     int MPI_SOURCE; // rank del mittente
3     int MPI_TAG;   // tag del messaggio
4     int MPI_ERROR; // eventuale codice d'errore
5 } MPI_Status;

```

C C

Serve a ricavare dei dati dal messaggio appena ricevuto, tramite una funzione aggiuntiva chiamata `MPI_Get_count` definita:

```
1 int MPI_Get_count(
2     MPI_Status*      status_p /* in */,
3     MPI_Datatype     type      /* in */,
4     int*             count_p   /* out */
5 );
```

C C

Possiamo capire quanti elementi stiamo ricevendo.

Proviamo adesso a scrivere una nuova versione di `hello world` che rispetta l'ordine delle stampa, facendo in modo che un processo padre riceva gli output dei processi figli.

È importante anche sapere che in MPI si usano dei tipi di dato specifici e non quelli classici di C:

MPI datatype	C datatype
<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_SHORT</code>	<code>signed short int</code>
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_LONG</code>	<code>signed long int</code>
<code>MPI_LONG_LONG</code>	<code>signed long long int</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned short int</code>
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <mpi.h>
4
5 const int MAX_STRING = 100;
6
7 int main(void) {
8     char greeting[MAX_STRING];
9     int comm_sz;
10    int my_rank;
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
```

C C

```

18     MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
19 } else {
20     printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
21     for (int q = 1; q < comm_sz; q++) {
22         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD,
23         MPI_STATUS_IGNORE);
24         printf("%s\n", greeting);
25     }
26
27     MPI_Finalize();
28     return 0;
29 }
```

Sending Order

MPI garantisce che i messaggi che si inviano da una coppia di processi arrivino nell'ordine di invio, quindi se ad esempio il processo p1 invia a p2 3 messaggi m1, m2 ed m3 questi arriveranno sempre nell'ordine di invio. Questo ordine però non è garantito quando ci sono mittenti diversi:

- Un processo p1 invia il messaggio m1 a p0 al tempo t0
- Un processo p2 invia il messaggio m2 a p0 al tempo t1

Ci aspettiamo che arrivi prima il messaggio m1 ma potrebbe non accadere sempre.

Oltre al comunicatore visto prima `MPI_COMM_WORLD` possiamo anche crearne di nostri per avere più libertà ed implementare sistemi più complessi.

Quindi per fare comunicare un insieme di processi possiamo o creare un comunicatore per ogni gruppo oppure usare i **tag**.

Message Matching

Prendiamo come esempio due processi che hanno rank `q` ed `r` che si scambiano dei messaggi:

```
1 // rank q
2 MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);
3
4 // rank r
5 MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm,
&status);
```

C C

È importante che i campi `dest` e `src` corrispondano ovvero:

- In `dest` dobbiamo avere il rank di `r` e su `src` il rank di `q`.
- `send_comm`, `recv_comm` devono essere lo stesso comunicatore.
- `send_tag`, `recv_tag` devono essere lo stesso tag.

Oltre a queste condizioni un messaggio può essere ricevuto correttamente se:

- `recv_type = send_type` - Quindi i tipi devono combaciare
- `recv_buf_sz ≥ send_buf_sz` - Il buffer del ricevente deve essere grande abbastanza da contenere tutto il buffer del mittente.

Un processo può ricevere un messaggio senza conoscere:

- Quanti dati riceve.
- Il mittente (`MPI_ANY_SOURCE`)
- Il tag del messaggio (`MPI_ANY_TAG`)

Il comando `MPI_Send` può comportarsi in modo diverso a seconda di:

- Dimensione del messaggio
- Disponibilità del ricevente
- Risorse del sistema

Per un `MPI_Send` ci sono due diversi protocolli di implementazione:

- **Eager Protocol:** Viene usato per messaggi piccoli, MPI copia i dati in un buffer interno e li spedisce immediatamente anche se il ricevente non ha ancora effettuato una receive. In questo caso restituisce subito un valore e quindi non è bloccante, finché il destinatario non riceve il messaggio questo vivrà nel buffer interno.
- **Rendezvous Protocol:** Si usa per messaggi grandi, il mittente non invia subito i dati ma manda prima una **request to send** e se il destinatario risponde in modo positivo allora i dati vengono inviati. In questo caso la send è bloccante, fino a quando il ricevente non è pronto.

Ogni implementazione decide la soglia per usare questi protocolli, in OpenMPI abbiamo:

- `< 8 kB` Eager
- `≥ 8 kB` Rendezvous

La receive però a differenza della send blocca sempre il programma in attesa di ricevere qualcosa, a meno che non si utilizzi una versione non bloccante come `MPI_Irecv`.

È importante capire però che anche se oggi un programma che scriviamo funziona perchè la send ritorna subito con messaggi piccoli non dobbiamo scrivere codice che si basa su questo comportamento perchè se lo portiamo su altre macchine con diverse implementazioni potremmo ottene dei comportamenti diversi.

Per scrivere buon codice bisogna pensare alle send come se fossero sempre bloccanti.

Attenzione!

1. Se un processo prova a ricevere un messaggio che però non combacia con nessuno di quelli inviati, il processo rimarrà bloccato per sempre.
2. Se una send è bloccante e nessuno la riceve allora anche il processo mittente rimarrà bloccato.
3. Se la send è bufferizzata come ad esempio nel protocollo eager e nessuno riceve i dati, allora i dati nel buffer andranno persi. O perchè il programma termina o perchè in generale non verranno mai prelevati dalla memoria.
4. Se il rank del processo destinatario è uguale a quello del mittente il processo potrebbe bloccarsi, o peggio, abbinarsi ad un messaggio sbagliato. (Questo comportamento può essere usato per inviare messaggi a se stessi ma va usato con attenzione perchè potrebbe portare a dei deadlock.)

2.2. Point-to-Point Communication Modes

`MPI_Send` utilizza il metodo di comunicazione **standard**, decide in base alla grandezza del messaggio se bloccare la chiamata fino alla ricezione da parte del destinatario o di ritornare prima che questo accada, questa seconda opzione avviene solo se il messaggio è piccolo e rende la `Send` **localmente bloccante** ovvero si blocca soltanto per copiare i dati nel buffer interno ma non per attendere una risposta da un altro processo.

Esistono però altri 3 metodi di comunicazione:

- **Buffered**: Funziona sempre con un buffer indipendentemente dalla grandezza del messaggio, l'unica differenza è che il buffer è fornito dall'utente.
- **Synchronous**: L'operazione di `Send` ritorna soltante se il destinatario ha iniziato la ricezione dei dati, è un'operazione **globally blocking**, infatti i due processi si sincronizzeranno nello stesso punto per scambiare i dati senza ulteriori comunicazioni.
- **Ready**: Il mittente invia i dati ma il destinatario deve essere già pronto, ovvero aver già chiamato una `Receive`, in caso contrario la `Send` ritorna un errore. Serve a ridurre l'overhead di handshake tra i due processi. Il lato negativo è che se il destinatario non è pronto alla ricezione l'invio fallisce, i due processi vanno quindi sincronizzati nel modo corretto.

Per utilizzarle hanno questa firma:

```
1 int [ MPI_Bsend | MPI_Ssend | MPI_Rsend ] (void *buf, int count,  
1   MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```



2.3. Non-Blocking Communication

In generale le `Send` non bloccanti rovinano le performance perché il mittente deve fermarsi e aspettare che venga conclusa la copia sul buffer. Le non bloccanti invece permettono di ritornare subito un valore e permettono **comunicazione e computazione** nello stesso momento. Esistono delle varianti non bloccanti, **immediate**, sia per `Send` che per `Receive`.

Un lato negativo è che il processo mittente per poter riutilizzare il buffer in altre comunicazioni deve comunque assicurarsi che i dati inviati siano stati ricevuti correttamente dal destinatario. Per controllare il buffer si utilizzano le operazioni `MPI_Wait` o `MPI_Test`. Anche il destinatario prima di poter utilizzare i dati deve assicurarsi che siano arrivati tutti correttamente, si utilizzano le stesse funzioni.

Vediamo prima la firma delle due operazioni `Isend` e `Irecv`:

```
1 int MPI_Isend (void *buf, // Address of data buffer
2                 int count, // Number of data items
3                 MPI_Datatype datatype, // Tipo di dato
4                 int source, // Rank del destinatario
5                 int tag, // Identificatore del tipo di messaggio
6                 MPI_Comm comm, // Comunicatore
7                 MPI_Request *req // Si usa per controllare lo stato
8                 della richiesta
9 )
```

```
1 int MPI_Irecv (void *buf, // Address of receive buffer
2                  int count, // Capacità del buffer in items
3                  MPI_Datatype datatype, // Tipo di dato
4                  int source, // Rank del mittente
5                  int tag, // Identificatore del tipo di messaggio
6                  MPI_Comm comm, // Comunicatore
7                  MPI_Request *req // Si usa per controllare lo stato
8                  della richiesta
9 )
```

Per controllare lo stato della ricezione o dell'invio utilizziamo quindi:

- **Metodo bloccante**, distrugge l'handle:

```
1 int MPI_Wait(MPI_Request *req, // Indirizzo dell'handle con lo stato della
2               richiesta la chiamata lo invalida impostandolo a MPI_REQUEST_NULL
3               MPI_Status *st // Indirizzo della struttura che conterrà le
4               informazioni del comunicatore
5 )
```

- **Metodo non bloccante**, distrugge l'handle solo se la comunicazione è avvenuta con successo:

```
1 int MPI_Test(MPI_Request *req, // Indirizzo dell'handle con lo stato della
2               richiesta la chiamata lo invalida impostandolo a MPI_REQUEST_NULL
3               int *flag, // Impostato a 1 se andata a buon fine
4 )
```

```
3     MPI_Status *st    // Indirizzo della struttura che conterrà le
        informazioni del comunicatore
4     )
```

Esistono comunque altre funzioni per testare il successo dell'operazione:

- Waitall
- Testall
- Waitany
- Testany
- ...

3. Parallel Program Design

3.1. Foster's Methodology

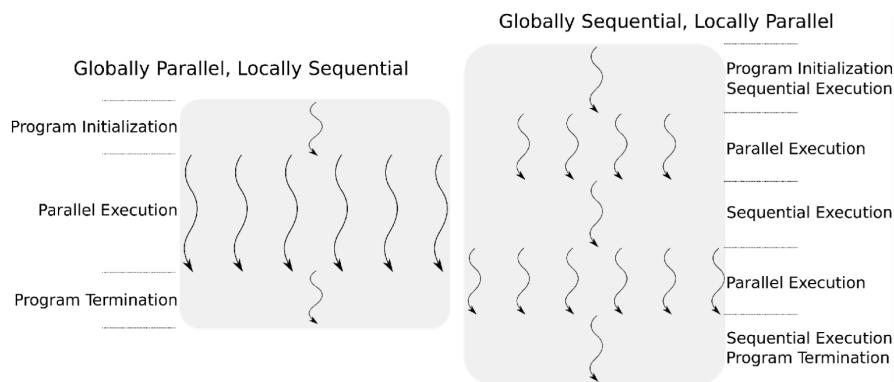
Questo metodo si divide in:

1. **Partitioning:** Dividere la computazione da svolgere in task più piccole, è importante quindi saper individuare le task eseguibili in parallelo.
2. **Communication:** Determinare il tipo di comunicazione necessarie fra le tasks individuate precedentemente. Ci si deve concentrare sul permettere una comunicazione veloce ed efficiente con meno spreco di lavoro possibile e senza errori di sincronizzazione.
3. **Agglomeration or Aggregation:** Combinare le tasks e le comunicazioni precedenti in tasks più grandi. Ad esempio se una task A va eseguita prima di una task B allora ha senso unirle in un'unica task, in questo modo riduciamo l'overhead delle comunicazioni fra queste due.
4. **Mapping:** Assegnare le tasks individuate a dei processi o threads. Andrebbe fatto in modo che ci sia il minor numero di comunicazioni possibile e i processi svolgano tutti la stessa quantità di lavoro.

4. Parallel Design Patterns

Possiamo dividere i pattern per programmi paralleli in due grandi categorie: **Globally Parallel, Locally Sequential (GPLS)** e **Globally Sequential, Locally Parallel (GSLP)**.

- **GPLS:** L'applicazione è in grado di svolgere più operazioni nello stesso momento, ciascuna operazione viene eseguita in modo sequenziale, ricadono in questa categoria:
 - Single-Program, Multiple Data
 - Multiple-Program, Multiple Data
 - Master-Worker
 - Map-reduce
- **GSLP:** L'applicazione viene eseguita come un programma sequenziale ma ci sono più parti eseguite in parallelo, ricadono in questo pattern:
 - Fork/join
 - Loop Parallelism



4.1. Single Program Multiple Data

Mantiene tutta la logica dell'applicazione in un unico programma, tipicamente hanno una struttura del tipo:

- **Program Initialization**
- **Obtaining a unique identifier:** Questi sono numerati a partire da 0 fino al numero di processi o threads, alcuni sistemi utilizzano dei vettori (CUDA).
- **Running the program:** Tutti i processi eseguono la stessa computazione ma su set di dati diversi.
- **Shutting down the program:** Pulizia, salvataggio dei risultati ecc...

4.2. Multiple Program Multiple Data

Il SPMD inizia a non funzionare bene quando ci sono tanti dati e quindi la memoria di un singolo processo non riesce a mantenerli oppure quando i processi non sono simili in termini di potenza (macchine eterogenee). Se abbiamo delle macchine con questa architettura allora possiamo valutare il MPMD che ha una struttura simile al precedente ma ci permette di eseguire compiti diversi su processori diversi, in questo modo possiamo affidare computazioni più pesanti ai core più potenti o implementare altre specifiche più adatte alla macchina.

4.3. Master-Worker

In questo approccio esistono due componenti, il Master e il Worker. Possono esserci più di un master ma in generale si occupa di:

- Mandare il lavoro da svolgere ai workers
- Raccogliere i risultati delle operazioni
- Svolgere operazioni di I/O tra i workers ma anche con l'utente.

Il master potrebbe però fare da bottleneck in alcuni contesti, se accade sarebbe meglio utilizzare una gerarchia di masters.

4.4. Map-Reduce

E' una variante del Master-Worker, il master coordina tutto e i workers possono svolgere due tipi di azione:

- Map: Applicare una funzione su dei dati, fornisce un set di risultati parziali.
- Reduce: Collezione i risultati parziali e ottiene quelli completi

4.5. Fork/Join

C'è un singolo thread padre che è il principale e vengono creati dinamicamente dei thread figli a runtime, è possibile utilizzare anche delle thread pool già pronte risparmiando tempo e risorse per la creazione e distruzione di thread. I thread figli devono finire le loro operazioni per permettere al padre di andare avanti.

4.6. Loop Parallelism

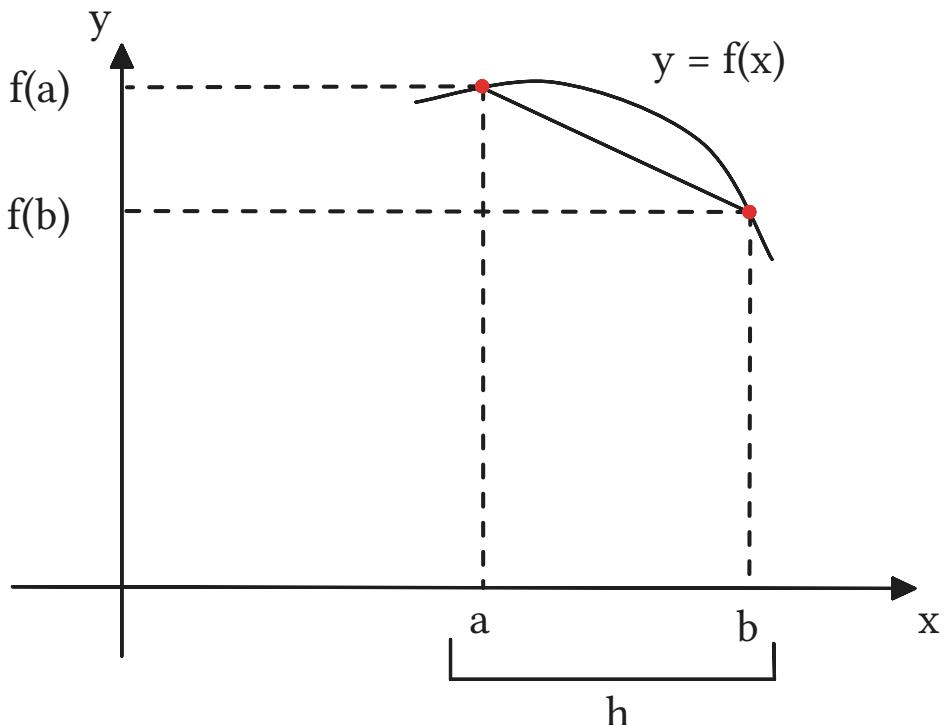
Viene utilizzato per trasformare programmi sequenziali in programmi paralleli, in generale modifica le variabili dei loop (devono avere una forma specifica) e li rende paralleli. Ha una flessibilità molto limitata ma è anche vero che dobbiamo impegnarci molto poco per implementarlo.

Input

Quasi tutte le implementazioni di MPI permettono soltanto al processo con rank 0 in **MPI_COMM_WORLD** di accedere a **stdin**, sarà quindi lui a dover prendere in input i dati dall'utente con *scanf* se necessari.

Esempio: Calcolo area sottostante ad una funzione

Data una funzione vogliamo calcolare l'area sottostante nel grafico in un intervallo a, b



L'area del trapezio è data da:

$$\frac{h}{2} [f(x_i) + f(x_{i+1})]$$

Ovvero somma delle basi per altezza diviso 2.

Per calcolare l'area sottostante alla funzione dividiamo l'area in tanti trapezi per ottenere un'approssimazione della misura, abbiamo quindi che $h = \frac{b-a}{n}$ e che $x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_{n-1} = a + (n-1)h, x_n = b$. Le x sono i punti di inizio e fine dei trapezi e h è la loro distanza (e anche la loro altezza).

Il calcolo della somma delle aree dei trapezi è:

$$h \cdot \left[f\left(\frac{x_0}{2}\right) + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + \frac{f(x_n)}{2} \right]$$

Possimo quindi scrivere uno pseudocodice seriale:

```

1 h = (b - a) / n;
2 approx = (f(a) + f(b)) / 2.0;
3 for (i = 1; i ≤ n - 1; i++) {
4     x_i = a + i * h;
5     approx += f(x_i);
6 }
7 approx = h * approx;

```

C

Possiamo renderlo parallelo andando a far calcolare a più processi diversi trapezi e poi unire i risultati.

Codice parallelo:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <mpi.h>
5 #include <math.h>
6
7 // Si utilizza la funzione coseno
8
9 double Trap(double left_endpt, double right_endpt, int trap_count, double
10 base_len) {
11     double estimate, x;
12     int i;
13
14     estimate = (cos(left_endpt) + cos(right_endpt)) / 2.0;
15     for (i = 1; i ≤ trap_count-1; i++) {
16         x = left_endpt + i * base_len;
17         estimate += cos(x);
18     }
19     estimate = estimate * base_len;
20     return estimate;
21 }
22
23 int main() {
24     int comm_sz, my_rank, n = 1024, local_n;
25     double a = 0.0, b = 1.0, h, local_a, local_b;
26     double local_int, total_int;
27     int source;
28
29     MPI_Init(NULL, NULL);
30     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
31     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
32
33     h = (b - a) / n;           // Altezza dei trapezi (distanza asse x)
34     local_n = n / comm_sz;    // Quanti trapezi deve calcolare ogni processo
35
36     local_a = a + my_rank * local_n * h;
37     local_b = local_a + local_n * h;
38     local_int = Trap(local_a, local_b, local_n, h);
39
40     if (my_rank ≠ 0) {
41         MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
42     } else {
43         total_int = local_int;
44         for (source = 1; source < comm_sz; source++) {
45             MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD,
46             MPI_STATUS_IGNORE);
47             total_int += local_int;
48         }
49     }
50 }
```



```

46         }
47     }
48
49     if (my_rank == 0) {
50         printf("Con %d trapezi, la stima da %f a %f = %f\n", n, a, b,
51             total_int);
52     }
53
54     MPI_Finalize();
55     return 0;
56 }
```

Però c'è un problema con questa implementazione infatti $p - 1$ processi inviano la loro stima al processo 0 che calcolerà la somma di tutti, ma questo non è bilanciato, infatti abbiamo un tempo di $(p - 1) * (T_{\text{sum}} + T_{\text{recv}})$ per il processo 0 mentre per tutti gli altri un tempo di T_{send} .

Un'alternativa potrebbe essere, per tutti i processi un tempo di $\log_2(p) * (T_{\text{sum}} + T_{\text{recv}})$

Il modo ottimale per effettuare una somma dipende dal numero di processi, la grandezza dei dati e dal sistema, avere un modo nativo per esprimere la somma migliora le performance e semplifica la programmazione. Utilizziamo **reduce**.

4.7. Collective Communication

Vediamo la firma per l'operazione di **Reduce** :

```

1 int MPI_Reduce(
2     void        *input_data_p, // .Dati da inviare
3     void        *output_data_p, // Buffer di ricezione
4     int          count,      // numero di elementi
5     MPI_Datatype datatype, // tipo di dato
6     MPI_Op        operator,    // operazione, sono definite da MPI
7     int          dest_process, // chi riceverà il dato
8     MPI_Comm      comm,       // comunicatore
9 );
```



Con questa operazione i dati di tutti i processi partecipanti vengono combinati tramite un'operazione e il risultato finale inviato ad un solo processo.

Possiamo ad esempio utilizzarla nella nostra implementazione del trapezio invece della send:

```

1 MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
2 MPI_COMM_WORLD);
```



Un'altra funzione molto utile è **MPI_Bcast** che serve ad inviare i dati di un processo a tutti gli altri processi nello stesso comunicatore:

```

1 int MPI_Bcast(
2     void        *data_p,      // Input o output
3     int          count,
4     MPI_Datatype datatype,
```



```
5     int          source_proc, // Rank di chi invia i dati
6     MPI_Comm      comm
7 );
```

Tutti i processi che la chiamano devono farlo con gli stessi paramentri, l'operazion è sincrona quindi termina soltanto quando tutti i processi hanno ricevuto i dati.

Infine c'è `MPI_Allreduce` che concettualmente corrisponde a `reduce + broadcast`, consente a tutti i processi di partecipare ad una riduzione, ottenere il risultato finale e poi inviarlo a tutti i processi:

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

C C

5. Performance Evaluation

Per effettuare una valutazione dei tempi di computazione utilizziamo la funzione `double MPI_Wtime(void);`, va chiamata al tempo di inizio della misurazione e alla fine, per misurare il tempo basta calcolare la differenza fra i due tempi:

```
1 double start, finish;
2 start = MPI_Wtime();
3 ...
4 finish = MPI_Wtime();
5 printf("Process %d, elapsed time: %e", my_rank, finish - start);
```

C

Ovviamente i processi, in molti casi, finiranno in tempi diversi ad esempio quando il processo padre (rank 0) esegue più lavoro rispetto agli altri. Dobbiamo quindi stampare il tempo massimo fra tutti i processi. Possiamo farlo con `reduce`:

```
1 double local_start, local_finish, local_elapsed, elapsed;
2
3 local_start = MPI_Wtime();
4 // Codice da valutare
5 ...
6 local_finish = MPI_Wtime();
7 local_elapsed = local_finish - local_start;
8
9 MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
10
11 if (my_rank == 0) {
12     printf("Elapsed time: %e\n", elapsed);
13 }
```

C

Un altro problema però è che non tutti i processi partiranno nello stesso momento e se questo accade allora il tempo che vedremo sarà elevato non perché abbiamo scritto male il programma ma perché semplicemente alcuni processi sono partiti più tardi di altri. Come ci assicuriamo quindi che i processi iniziano nello stesso momento?

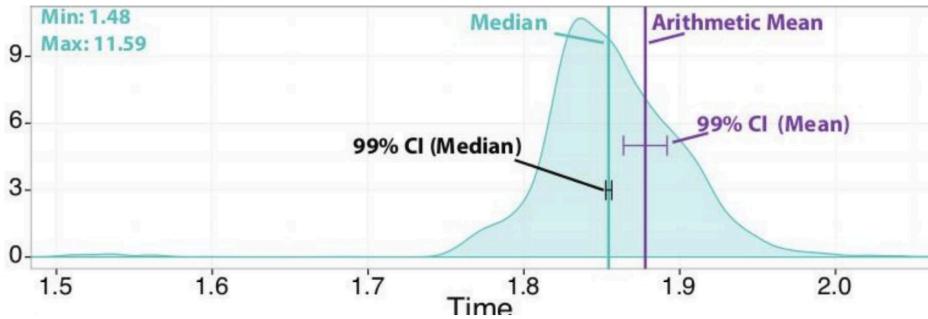
Utilizziamo `MPI_Barrier`, in alcune implementazioni potrebbe non garantire quello che serve a noi ma per lo scopo del corso va bene.

- **Basta una sola misurazione per valutare le performance?**

Ovviamente no, se eseguiamo il codice 100 volte vedremo molti risultati diversi, soprattutto se durante le esecuzioni cambiano le condizioni della macchina come ad esempio memoria utilizzata, utilizzo del processore e altro.

Per eseguire una valutazione quindi eseguiamo il codice molte volte e facciamo un report della distribuzione totale dei tempi.

Ad esempio



Quindi, per raccogliere dati sui tempi d'esecuzione dobbiamo:

1. Impostare un **Barrier** all'inizio in modo che tutti i processi inizino nello stesso momento
2. Salvare il tempo massimo d'esecuzione fra tutti i rank
3. Eseguire l'applicazione più volte e salvare la distribuzione dei tempi

Confrontiamo adesso alcuni tempi d'esecuzione e diamo delle definizioni.

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

- Notiamo che nel caso di un'esecuzione sequenziale, quindi con un solo thread, i tempi crescono con il crescere della grandezza del problema. (Basta guardare la prima riga)
- Se scegliamo una grandezza (una colonna) e scendiamo verso il basso, quindi aumentiamo i processi, notiamo che i tempi diminuiscono

Osserviamo però che con una matrice grande 1024 i tempi d'esecuzione con 8 e 16 thread sono uguali. Perché?

Ci aspettiamo che se eseguiamo un programma con p processi allora sarà p volte più veloce di quando lo eseguiamo con 1, non è sempre così. Definiamo

- $T_{\text{serial}}(n)$ come il tempo con esecuzione sequenziale e n la dimensione del problema.
- $T_{\text{parallel}}(n, p)$ come il tempo con esecuzione parallela con p processi.
- $S(n, p)$ come la **speedup** (incremento) dell'esecuzione parallela:

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Idealmente, vorremmo avere $S(n, p) = p$, se questo accade diciamo che abbiamo **linear speedup**.

Osserviamo i tempi di speedup con lo stesso problema di prima:

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
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

Parallelizzazione con un solo processo

E' importante notare che $T_{\text{serial}}(n) \neq T_{\text{parallel}}(n, 1)$.

In generale si ha che $T_{\text{parallel}}(n, 1) \geq T_{\text{serial}}(n)$

Questo perché il codice parallelo richiede delle operazioni di preparazione anche per un solo processo e quindi sarà più lento rispetto ad un codice seriale.

Oltre allo **speedup** possiamo calcolare la **scalability**:

$$S(n, p) = \frac{T_{\text{parallel}}(n, 1)}{T_{\text{parallel}}(n, p)}$$

E anche l'efficienza:

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \cdot T_{\text{parallel}}(n, p)}$$

Vorremmo avere un'efficienza = 1 ma in pratica avremo sempre dei valori ≤ 1 e andrà sempre peggio con problemi più piccoli.

Strong vs Weak Scaling

- **Strong Scaling:** Fissiamo la grandezza del problema e incrementiamo il numero di processi, se manteniamo un'efficienza alta allora il nostro programma è **strong scalable**.
- **Weak Scaling:** Incrementiamo la grandezza del problema insieme al numero dei processi, se manteniamo un'alta efficienza allora il programma è **weak scalable**.

Vediamo dei dati esempio con lo stesso problema di prima ed i dati sull'efficienza:

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Notiamo che il programma non è strong scalable dato che l'efficienza non rimane alta.

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Notiamo che il programma è weak scalable dato che manteniamo un'alta efficienza.

Possiamo capire in qualche modo come si comporterà il codice?

Amdahl's Law

Ogni programma ha una parte che non può essere parallela (**serial fraction** $1 - \alpha$), ad esempio leggere dei dati o in generale calcoli dove abbiamo bisogno di un valore precedente alla computazione corrente.

La **Amdahl's Law** ci dice che la speedup è limitata da questo $1 - \alpha$ non parallelo:

$$T_{\text{parallel}}(p) = (1 - \alpha)T_{\text{serial}} + \alpha \frac{T_{\text{serial}}}{p}$$

I casi estremi della formula sono:

- Se $\alpha = 0$ allora il codice non può essere parallelizzato e quindi $T_{\text{parallel}}(p) = T_{\text{serial}}$
- Se $\alpha = 1$ allora tutto il codice può essere parallelizzato e $T_{\text{parallel}}(p) = \frac{T_{\text{serial}}}{p}$ ovvero lo speedup ideale.

Per calcolare lo speedup:

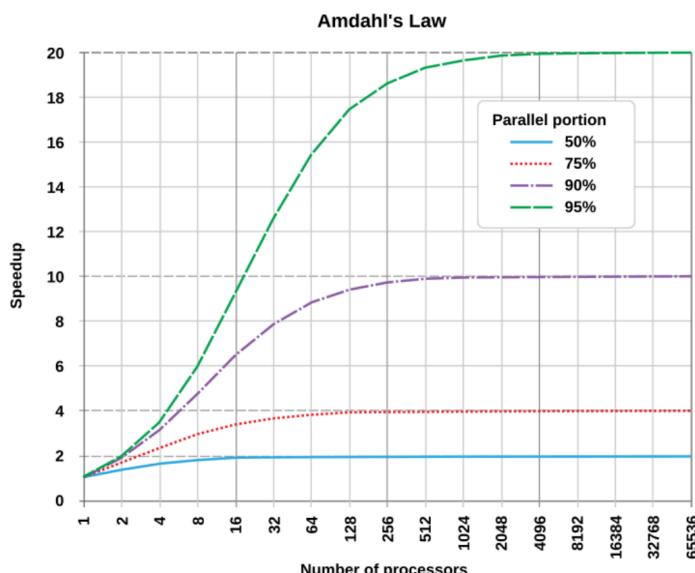
$$S(p) = \frac{T_{\text{serial}}}{(1 - \alpha)T_{\text{serial}} + \alpha \frac{T_{\text{serial}}}{p}}$$

E a questo punto vedere anche lo speedup massimo:

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

Ad esempio se abbiamo il 60% parallelizzabile ovvero $\alpha = 0.6$ allora avremo uno speedup massimo di 2.5

La formula è anche invertibile, quindi se vogliamo capire di quanti processi abbiamo bisogno per raggiungere un certo speedup possiamo farlo. Ad esempio se volessimo usare 100000 processi dovremmo avere $\alpha \geq 0.99999$



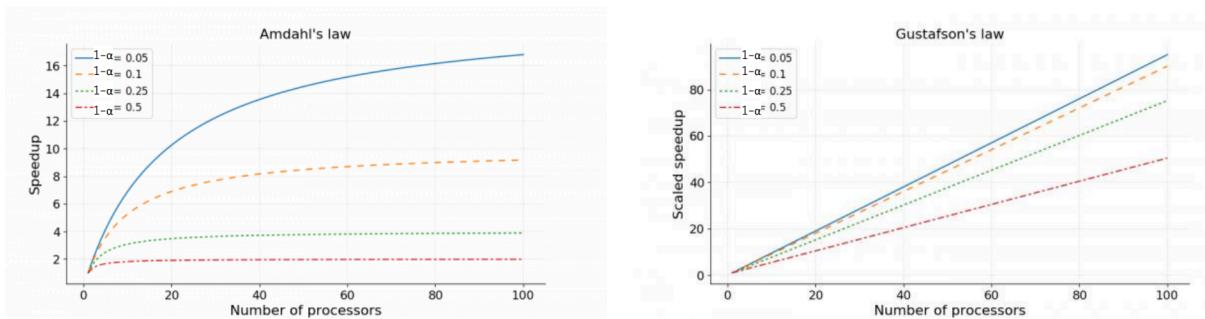
Gustafson's Law

Se consideriamo il weak scaling, la parallel fraction aumenta insieme alla grandezza del problema, quindi il tempo seriale rimane costante ma quello parallelo aumenta. **Scaled Speedup**:

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

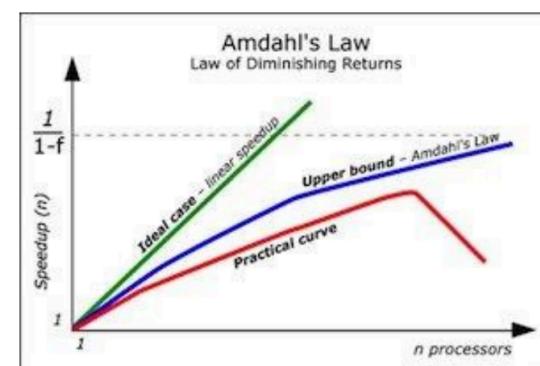
Il carico su ogni thread rimane costante.

Vediamo i due grafici a confronto:



Amdahl's Law Limitations

La serial fraction potrebbe aumentare con l'aumentare del numero di processi, quindi nella realtà non sempre il codice seguirà le stime, infatti potrebbero verificarsi situazioni come:



Vediamo altre funzioni utili di MPI:

- **MPI_Scatter**: Si utilizza per spezzare un dato in più parti ed inviare ciascuna parte ad un processo, ad esempio il processo 0 legge un array, lo divide in parti uguali e invia ciascuna di queste agli altri processi.

```
1 int MPI_Scatter(
2     void*           send_buf_p,
3     int             send_count,
4     MPI_Datatype   send_type,
```



```

5   void*      recv_buf_p,
6   int       recv_count,
7   MPI_Datatype  recv_type,
8   int       src_proc,
9   MPI_Comm    comm
10 );

```

Il `send_count` è il numero di elementi da inviare a ciascun processo non il totale. Se i dati non possono essere divisi in maniera omogenea, ad esempio abbiamo un array da 5 elementi e vogliamo inviare a ciascun processo 2 elementi possiamo utilizzare `MPI_Scatterv`, altrimenti la Scatter normale va a prendere elementi fuori dal buffer.

Risparmiare Spazio

Anche il processo 0 che invia i dati riceverà una parte di questi, se non vogliamo fargli allocare un nuovo buffer possiamo usare `MPI_IN_PLACE`:

```

1 if (rank == 0) {
2   MPI_Scatter(buff, 3, MPI_INT, MPI_IN_PLACE, 3, MPI_INT, 0, MPI_COMM_WORLD);
3 } else {
4   MPI_Scatter(buff, 3, MPI_INT, dest, 3, MPI_INT, 0, MPI_COMM_WORLD);
5 }

```

La sua «inversa» è `Gather`, permette di collezionare le parti di un dato in un unico processo:

```

1 int MPI_Gather(
2   void*      send_buf_p,
3   int       send_count,
4   MPI_Datatype  send_type,
5   void*      recv_buf_p,
6   int       recv_count,
7   MPI_Datatype  recv_type,
8   int       dest_proc,
9   MPI_Comm    comm
10 );

```

Anche qui il `send_count` è il numero di parti che ciascun processo manda e non il totale.

5.1. Matrici

Le matrici possiamo rappresentarle in C o come tanti array allocati dinamicamente oppure come un lungo array contiguo.

Per allocare dinamicamente una matrice come tanti array dobbiamo:

```

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 }

```

Se vogliamo però effettuare delle riduzioni, come ad esempio `Reduce` dobbiamo fare attenzione, a primo impatto vorremo scrivere:

```
1 MPI_Reduce(a, recvbuf, num_rows * num_cols, MPI_INT, MPI_SUM, 0,
2 MPI_COMM_WORLD);
3 // Oppure
4 MPI_Reduce(a[0], recvbuf, num_rows * num_cols, MPI_INT, MPI_SUM, 0,
5 MPI_COMM_WORLD);
```

C C

Ma sono entrambi degli approcci sbagliati dato che gli array allocati dinamicamente nel modo visto sopra non è detto che siano contigui in memoria. Il metodo corretto è fare una reduce per ogni array della matrice:

```
1 for (int i = 0; i < num_rows; i++) {
2     MPI_Reduce(a[i], recvbuf[i], num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
3 }
```

C C

Se invece allochiamo la matrice come un unico array:

```
1 int *a;
2 a = (int*) malloc(sizeof(int) * num_rows * num_cols);
3 ...
4 ...
5 //Per accedere agli elementi
6 a[i * num_cols + j] = ...
```

C C

E in questo caso il metodo visto sopra per le `Reduce` è giusto, quindi:

```
1 MPI_Reduce(a, recvbuf, num_rows * num_cols, MPI_INT, MPI_SUM, 0,
2 MPI_COMM_WORLD);
```

C C

In alcuni casi vengono usate, una dopo l'altra, le operazioni `Gather` e `Broadcast`, in MPI se due operazioni vengono spesso usate una dopo l'altra allora molto probabilmente esiste un'operazione che le combina, vediamo infatti `Allgather`:

```
1 int MPI_Allgather(
2     void*           send_buf_p,
3     int             send_count,
4     MPI_Datatype   send_type,
5     void*           recv_buf_p,
6     int             recv_count,
7     MPI_Datatype   recv_type,
8     MPI_Comm       comm
9 );
```

C C

Altre funzioni utili sono:

- `Reduce-Scatter`: Esegue un'operazione di riduzione, una volta ottenuto il vettore risultato lo distribuisce in parti uguali a tutti i processi.
- `MPI_Alltoall`: Tutti i processi inviano i dati a tutti gli altri processi (inclusi se stessi) e tutti ricevono dati da tutti.

6. Derived Datatypes

Supponiamo di dover inviare le coordinate di punti sul piano ed il loro colore:

Per ogni punto effettueremo 3 send

```
1 MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
2 MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
3 MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
```

C C

Stessa cosa per le `Receive`. Si può rendere più efficiente? Si, utilizzando i **Derived Datatypes**. Grazie a questi possiamo salvare in memoria una collezione di dati memorizzando sia il tipo di dato di ciascuno di essi sia la loro posizione. Grazie a questi tipi di dato possiamo, prima di inviare i dati, raccoglierli tutti e fare una sola send. Ragionamento analogo per le receive.

Per costruire le struct utilizziamo:

```
1 int MPI_Type_create_struct (
2     int             count,
3     int             array_of_blocklengths[],
4     MPI_Aint        array_of_displacements[],
5     MPI_Datatype    array_of_types[],
6     MPI_Datatype*   new_type_p
7 );
```

C C

Ad esempio se vogliamo ricreare la struct:

```
1 struct t {
2     double a;
3     double b;
4     int n;
5 }
```

C C

Avremo come parametri:

- `count = 3`
- `blocklengths = 1,1,1`
- `displacements = 0, 16, 24`
- `types: MPI_DOUBLE, MPI_DOUBLE, MPI_INT`

Non sempre siamo a conoscenza dei `displacements`, utilizziamo quindi:

```
1 int MPI_Get_address(
2     void*      location_p,
3     MPI_Aint*  address_p
4 );
```

C C

Serve a ottenere l'indirizzo di memoria referenziato dal puntatore `location_p`, il tipo `MPI_Aint` è un tipo speciale di MPI in grado di contenere qualsiasi indirizzo di sistema.

Quindi usando la stessa struct di prima avremo:

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

C C

Per calcolare questi indirizzi ovviamente va prima creata un'istanza della struct in modo da calcolare gli indirizzi

Infine, per poter utilizzare il tipo appena creato nelle altre funzioni dobbiamo chiamare:

```
1 int MPI_Type_commit(MPI_Datatype* new_mpi_t_p);
```

C C

A questo punto possiamo utilizzarlo anche nelle altre funzioni, ricordiamoci però che una volta finito il suo utilizzo va eseguito un `free`:

```
1 int MPI_Type_free(MPI_Datatype* old_mpi_t_p);
```

C C

Possiamo creare in generale tipi di dati contigui e non per forza solo `struct`, con le varie funzioni:

- `MPI_Type_contiguous`
- `MPI_Type_vector`
- `MPI_Type_create_subarray`
- `MPI_pack / MPI_unpack`

7. Shared Memory Programming with OpenMP

In OpenMP il sistema è visto come una collezione di cores o CPU, tutti questi hanno accesso alla stessa memoria. L'obiettivo principale è quello di scomporre un programma sequenziali in più componenti che possono essere eseguite in parallelo. Per fare questo si utilizzano le **direttive del compilatore**.

OpenMP segue quindi il paradigma **Globally Sequential, Locally Parallel** e il **fork-join**, infatti i programmi iniziano e sono sequenziali ma durante la loro esecuzioni vengono eseguite delle parti in parallelo.

Pragmas

Sono delle istruzioni speciali lette durante il preprocessing del programma, non fanno parte della specifica di base di C. I compilatori che non le supportano, quindi, le ignorano. Si indicano con `#pragma`

Vediamo alcune direttive

- `# pragma omp parallel` : È la più basica, possiamo definire il numero di thread che eseguiranno il blocco a run-time.

Per terminare il blocco di codice parallelo usiamo: `# pragma omp end parallel`, si può omettere se il codice parallelo dura fino alla fine della funzione.

Ad esempio:

```
1 int main(int argc, char *argv[]) {  
2     int thread_count = strtol(argv[1], NULL, 10);  
3  
4     # pragma omp parallel num_threads(thread_count)  
5         Hello();  
6     return 0;  
7 }  
8  
9 void Hello(void) {  
10    int my_rank = omp_get_thread_num();  
11    int thread_count = omp_get_num_threads();  
12    printf("Hello from thread %d of %d\n", my_rank, thread_count);  
13 }
```

C C

Per compilarlo ed eseguirlo:

```
1 // Compilazione  
2 gcc -g -Wall -fopenmp -o omp_hello omp_hello.c  
3 // Esecuzione  
4 ./omp_hello 4
```

Dove 4 indica il numero di thread con cui vogliamo eseguire il programma.

Thread Team Size Control

Possiamo modificare il numero di thread con cui eseguire i programmi in diversi modi:

- **Dalla shell** utilizzando la variabile globale `OMP_NUM_THREADS` :
 - `echo ${OMP_NUM_THREADS}` per sapere quanto vale la variabile in questo momento
 - `export OMP_NUM_THREADS=4` per impostarla
- **Program Level** attraverso la funzione `omp_set_num_threads` fuori da un costrutto OpenMP
- **Pragma Level** attraverso la direttiva `num_threads`

Se chiamiamo `omp_get_num_threads` riceviamo il numero di threads in una parte di codice parallela, in una sequenziale riceviamo 1. Una chiamata a `omp_get_thread_num` invece ci restituisce l'id del thread che l'ha chiamata.

Clause

È un testo che serve a modificare una direttiva, prima abbiamo usato `num_threads` per specificare il numero di threads che devono eseguire il blocco parallelo.

È importante osservare che il sistema potrebbe imporre delle limitazioni sul numero di threads che un programma può utilizzare, infatti OpenMP standard non garantisce che il programma crei esattamente `thread_count` threads.

Terminologia

In OpenMP i threads che eseguono lo stesso blocco parallelo sono chiamati **team**:

- **master**: Il thread originale d'esecuzione
- **parent**: Un thread che ha incontrato un direttiva per del codice parallelo e ha avviato un team di threads. In alcuni casi il thread parent è anche il master
- **child**: Ogni thread avviato da un parent è considerato un thread child.

Mancato supporto di OpenMP

Nel caso in cui il compilatore non supporti OpenMP possiamo utilizzare:

```
1 # include <omp.h>
2 // Oppure
3 # ifdef _OPENMP
4 # include <omp.h>
5 # endif
6
7 # ifdef _OPENMP
8     int my_rank = omp_get_thread_num();
9     int thread_count = omp_get_num_threads();
10 # else
11     int my_rank = 0;
12     int thread_count = 1;
13 # endif
```

Adesso prendiamo come esempio il codice visto con la regola del trapezio, ad un certo punto i thread dovranno sommare i valori ottenuti ed eseguire quindi:

```
1 global_result += my_result;
```

Ma siccome in OpenMP utilizzano una memoria condivisa, abbiamo il problema della **mutua esclusione**, se due processi lavorano sulla stessa memoria nello stesso momento avremo risultati inaspettati.

Si utilizza `# pragma omp critical` per marcare una sezione critica, queste potranno essere eseguite soltanto da un thread alla volta.

Un'altra direttiva con lo stesso scopo ma, in alcuni casi più efficiente, è `# pragma omp atomic`, questa protegge in una sezione critica soltanto l'incremento della variabile globale, se infatti avessimo un codice tipo `global_result += my_function()` allora la funzione `my_function` verrebbe ancora eseguita in parallelo fra tutti i thread.

Scope

In un programma seriale lo scope di una variabile è quella parte di programma dove la variabile viene utilizzata. In OpenMP lo scope si riferisce a quell'insieme di threads che possono accedere alla variabile in un blocco parallelo.

- **Shared Scope:** Una variabile a cui possono accedere tutti i threads nel team.
- **Private Scope:** Una variabile a cui può accedere un solo thread.

Lo scope predefinito per le variabili dichiarate prima di un blocco parallelo è **shared**.

7.1. Reduction Clause

Prima abbiamo utilizzato una variabile globale per memorizzare il risultato accumulato dai vari threads. Possiamo farlo anche con una funzione che ritorna il valore dell'area? Ad esempio:

```
1 double Local_Trap(double a, double b, int n);
```

C C

Si, ma se scriviamo:

```
1 global_result = 0.0;
2 # pragma omp parallel num_threads(thread_count)
3 {
4 # pragma omp critical
5   global_result += Local_trap(double a, double b, int n);
6 }
```

C C

Abbiamo il problema della sezione critica visto prima, per non avere errori i thread andrebbero eseguiti in modo sequenziale.

Possiamo però risolvere il problema definendo una variabile privata nel blocco parallelo e muovere la sezione critica dopo la chiamata a funzione:

```
1 global_result = 0.0;
2 # pragma omp parallel num_threads(thread_count)
3 {
4   double my_result = 0.0;
5   my_result += Local_trap(double a, double b, int n);
6 # pragma omp critical
7   global_result += my_result;
8 }
```

C C

Quello che abbiamo appena fatto è molto simile a una **reduce** in OpenMPI, anche in OpenMP abbiamo delle operazioni simili.

Reduction Operators

Un **reduction operator** è un operatore binario. Una **riduzione** è un'operazione che ripete la stessa operazione (riduzione) ad una serie di operandi fino ad ottenere un risultato singolo. Tutti i valori intermedi della riduzione dovrebbero essere salvati nella stessa variabile, la **reduction variable**.

Per utilizzarle possiamo aggiungere direttamente le **reduction clause** alle direttive di parallelizzazione:

```
1 reduction(<operator>: <variable list>)
2
3 //Esempio
4 global_result = 0.0;
5 # pragma omp parallel num_threads(thread_count) reduction(+: global_result)
6
7 global_result += Local_trap(double a, double b, int n);
```

C C

Info sulla Riduzione

La variabile privata inizializzata per la riduzione viene inizializzata al valore identità per l'operatore di riduzione, quindi ad esempio se usiamo una moltiplicazione abbiamo 1, con una somma 0.

Inoltre la riduzione accumula i valori sia all'interno che all'esterno del blocco parallelo.

Esempio

```
1 int acc = 6; C C
2 # pragma omp parallel num_threads(5) reduction(*: acc)
3 {
4     acc += omp_get_thread_num();
5     printf("thread %d: private acc is %d\n", omp_get_thread_num(), acc);
6 }
7 printf("after: acc is %d\n", acc)
```

Otteniamo come output:

```
1 tid = 0 sum = 1
2 tid = 3 sum = 4
3 tid = 4 sum = 5
4 tid = 2 sum = 3
5 tid = 1 sum = 2
6 Final sum = 720
```

Otteniamo 720 perché appunto ogni thread ottiene come risultato locale $1 + \text{tid}$ dato che `acc` è inizializzata ad 1 per via della moltiplicazione e infine tutti i valori, anche il 6 esterno, vengono ridotti con la moltiplicazione e otteniamo quindi $1 * 2 * 3 * 4 * 5 * 6 = 720$.

Info sugli scope

Si può sovrascrivere lo scope di default con:

```
1 default(none) C C
```

Facendo così il compilatore richiederà di specificare lo scope di ogni variabile che usiamo in un blocco parallelo e che è stata dichiarata al suo esterno.

Esistono dei modificatori che vanno inseriti nelle direttive per scegliere gli scope delle variabili:

- **shared**: È il comportamento di default delle variabili dichiarate fuori dai blocchi paralleli, va utilizzato soltanto se abbiamo specificato anche `default(none)`.
- **reduction**: Un'operazione di riduzione viene effettuata tra le copie private e gli oggetti esterni dal blocco, il valore finale è salvato nell'oggetto esterno.
- **private**: Crea una copia della variabile per ogni thread del team, le variabili private **non sono inizializzate** quindi non si ha il valore dichiarato all'esterno del blocco parallelo.

Esempio

```
1 int x = 5;
2 # pragma omp parallel private(x)
3 {
4     x = x+1 // Pericoloso dato che x non è inizializzata
5     printf("thread %d: private x is %d\n", omp_get_thread_num(), x);
6 }
7 printf("after: x is %d\n", x); // Pericoloso dato che stampa la x dichiarata
nella prima riga
```

C C

Infatti otteniamo come output:

```
1 thread 0: private x is 1
2 thread 1: private x is 1
3 thread 3: private x is 1
4 thread 2: private x is 1
5 after: x is 5
```

Altri modificatori sono:

- **firstprivate**: Si comporta come il private ma la copia della variabile privata è inizializzata al valore di quella esterna.
- **lastprivate**: Si comporta come il private ma l'ultimo thread, **quello che esegue l'ultima iterazione del blocco sequenziale**, copia il valore della variabile in quella esterna.

Quando dichiariamo una variabile **privata** questa non esisterà più una volta usciti dal suo scope. Come possiamo dichiarare una variabile privata che esiste in più sezioni parallele?

- **threadprivate**: Crea uno storage dedicato ai thread per i dati globali, questi dati devono essere delle variabili globali o statiche in C o membri statici di una classe in C++.
- **copyin**: Utilizzata insieme a **threadprivate** per inizializzare le copie threadprivate di un team di threads dalle variabili del thread master.

Esempio

```
1 int tp;
2
3 int main(int argc, char **argv) {
4     # pragma omp threadprivate(tp)
5
6     # pragma omp parallel num_threads(7)
7     tp = omp_get_thread_num();
8
9     # pragma omp parallel num_threads(9)
10    printf("Thread %d has %d\n", omp_get_thread_num(), tp);
11 }
```

C C

Otteniamo come output:

```
1 Thread 3 has 3
```

```

2 Thread 2 has 2
3 Thread 8 has 0
4 Thread 7 has 0
5 Thread 5 has 5
6 Thread 4 has 4
7 Thread 6 has 6
8 Thread 1 has 1
9 Thread 0 has 0

```

Questo perché i thread del secondo blocco non hanno associato il loro valore alla variabile.

Esempi con threadprivate

Se vogliamo che il dato privato sia uguale per tutti i threads:

```

1 # pragma omp threadprivate(private_var)
2 ...
3 private_var = 1;
4
5 # pragma omp parallel copyin(private_var)
6 private_var += omp_get_thread_num()

```

C

Se invece vogliamo fare in modo che un thread imposti il suo valore privato anche sugli altri thread:

```

1 # pragma omp parallel
2 {
3     # pragma omp single copyprivate(private_var)
4     private_var = 4
5 }

```

C

7.2. Parallel For

Con la direttiva che vedremo potremo rendere parallela l'esecuzione di un for loop. Ad esempio il codice per l'esercizio del trapezio:

```

1 h = (b - a) / n;
2 approx = (f(a) + f(b)) / 2.0;
3 for (i = 1; i <= n - 1; i++) {
4     approx += f(a + i * h);
5 }
6 approx = h * approx;

```

C

Possiamo parallelizzarlo con:

```

1 h = (b - a) / n;
2 approx = (f(a) + f(b)) / 2.0;
3 # pragma omp parallel for num_threads(thread_count) reduction(+ : approx)
4 for (i = 1; i <= n - 1; i++) {
5     approx += f(a + i * h);

```

C

```

6 }
7 approx = h * approx;

```

È importante ricordare però che non tutti i tipi di *for* sono parallelizzabili con questa direttiva. I *for* concessi devono rispettare:

- La variabile index usata per indicare le iterazioni deve essere un intero o un puntatore.
- Le espressioni `start`, `end`, `incr` devono essere di un tipo compatibile ad index.
- Le espressioni `start`, `end`, `incr` non devono cambiare durante le iterazioni.
- Durante le iterazioni la variabile index può essere modificata soltanto dall'espressione di incremento indicata nel *for*.

Ottimizzazione con i threads

Se in più punti del codice utilizziamo dei *for* paralleli, ad esempio:

```

1 ...
2 # pragma omp parallel for num_threads(thread_count) default(none) shared(a,
3   n) private(i, tmp)
4 for (i = 1; i < n; i += 2) {
5 ...
6 ...
7
8 # pragma omp parallel for num_threads(thread_count) default(none) shared(a,
9   n) private(i, tmp)
10 for (i = 1; i < n - 1; i += 2) {
11 ...
12 ...

```

Ogni volta verranno creati nuovi threads e non è molto efficiente, infatti potremmo creare direttamente dei thread ad inizio programma e riutilizzare sempre gli stessi:

```

1 # pragma omp parallel for num_threads(thread_count) default(none)
2   shared(a, n) private(i, tmp, phase)
3 ...
4 # pragma omp for
5 for (i = 1; i < n; i += 2) {
6 ...
7 ...
8 # pragma omp for
9 for (i = 1; i < n - 1; i += 2) {
10 ...
11 }

```

Questo si applica in generale a tutti gli approcci multithreads e non soltanto ai *for*.

In OpenMP è possibile dare dei nomi alle sezioni critiche con:

```
1 # pragma omp critical(name)
```

C C

In questo modo, due blocchi protetti perché sezione critica possono comunque essere eseguiti nello stesso momento se hanno un nome diverso. Questo dobbiamo comunque farlo a tempo di compilazione ma se non sappiamo a tempo di compilazione quante sezioni critiche ci sono? Utilizziamo i locks in OpenMP:

```
1  omp_lock_t writelock;
2  omp_init_lock(&writelock);
3
4  # pragma omp parallel for
5  for (i = 0; i < x; i++) {
6      ...
7      omp_set_lock(&writelock);
8      // Codice da far eseguire da un solo thread alla volta
9      omp_unset_lock(&writelock);
10     ...
11 }
12 omp_destroy_lock(&writelock);
```

C C

Cosa sceglieremo quindi? **Critical, Atomic o i Locks?** Atomic è quella che di solito è la più veloce per farci ottenere mutua esclusione. I lock andrebbero utilizzati quando vogliamo mutua esclusione per delle strutture dati invece che un blocco di codice.

Inoltre è importante non mischiare i due tipi di mutua esclusione per una singola sezione critica perché non è garantita l'equità, è quindi pericoloso innestare due costrutti di mutua esclusione:

```
1 int main() {
2     # pragma omp critical
3     y = f(x);
4     ...
5 }
6
7 double f(double x) {
8     # pragma omp critical
9     z = g(x); // z is shared
10    ...
11 }
12 return z;
```

C C

Il codice sopra andrà in **deadlock**, in questo caso possiamo risolverlo rinominando le sezioni:

```
1 int main() {
2     # pragma omp critical (one)
3     y = f(x);
4     ...
5 }
```

C C

```

7  double f(double x) {
8      # pragma omp critical (two)
9      z = g(x); // z is shared
10     ...
11 }
12 return z;

```

7.3. Nested Loops

Nella maggior parte dei casi quando si hanno dei cicli for annidati possiamo parallelizzare soltanto quello più esterno. Inoltre, spesso, il ciclo più esterno è quello con meno iterazioni e non tutti i thread creati vengono utilizzati ma questo non significa che è meglio parallelizzare quello interno infatti potrebbe accadere che alcuni thread hanno più lavoro di altri.

La soluzione corretta è quella di collassarli tutti in unico loop, possiamo farlo o noi manualmente oppure farlo fare ad OpenMP:

```

1 a();
2 # pragma omp parallel for collapse(2)
3 for (int i = 0; i < 3; ++i) {
4     for (int j = 0; j < 6; ++j) {
5         c(i, j);
6     }
7 }
8 z();

```

Attenzione!

Di default la parallelizzazione innestata è disabilitata, ovvero non possiamo fare questo:

```

1 # pragma omp parallel for
2 for (int i = 0; i < 3; ++i) {
3     # pragma omp parallel for
4     for (int j = 0; j < 6; ++j) {
5         c(i, j);
6     }
7 }

```

Questo perché, se fosse abilitata, andrebbe a creare ad esempio su una macchina con 4 threads: $3 * 4 = 12$ threads

8. GPU Programming

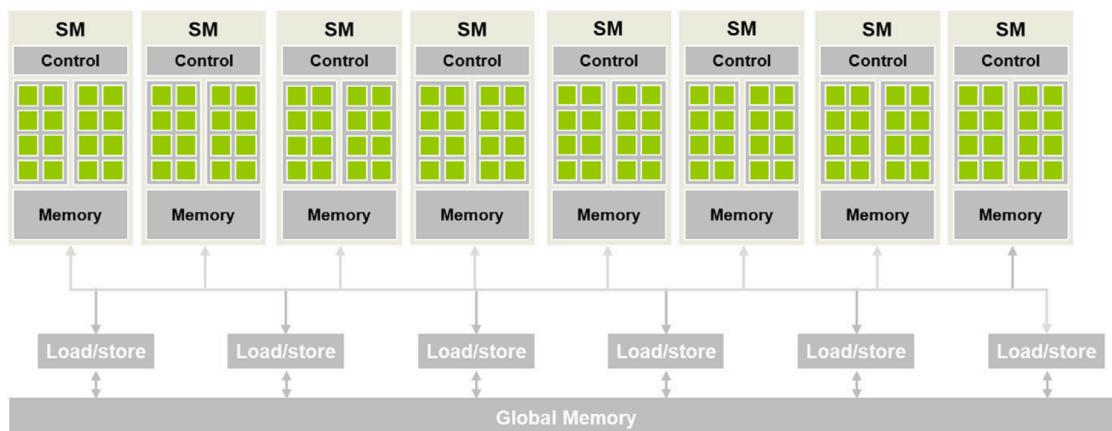
Per ora abbiamo visto la computazione avvenire nella CPU, più precisamente nei core della ALU, questi comportano:

- Alte frequenze di clock
- Grandi cache che riducono i tempi di accesso in memoria
- Predizione delle operazioni da svolgere che in caso di successo riducono i tempi di esecuzione

Adesso iniziamo a fare calcoli con la GPU:

- Ci sono molti più core ma ad una frequenza di clock moderata
- Cache più piccole
- Niente branch control quindi niente predizioni di operazioni
- Ha delle interfacce a banda ultralarga per le memorie.

Architettura di una GPU CUDA-Capable



Sono composte da:

- **SM (Streaming Multiprocessor)**: Sono un insieme di quadrati verdi, uno singolo è uno streaming processor (o anche CUDA Core). I processori di un multiprocessor condividono la stessa control logic e instruction cache.
- Due o più SM formano un **building block**
- Sono tutti collegati da una memoria a banda larga.

Application Benefits from CPU and GPU

- Utilizziamo la CPU per le parti di programma sequenziali dove è importante la latenza.
- La GPU la utilizziamo per codice parallelo dove è più importante il throughput.

La programmazione con le GPU ha un ostacolo principale ovvero che la memoria della GPU e dell'host non sono unite e quindi sono necessarie delle operazioni esplicite per trasferire i dati fra le due memorie. Un altro problema è che non sempre la GPU ha la stessa rappresentazione della CPU per i numeri floating-point.

Piattaforme per lo sviluppo con GPU

- **CUDA:** Compute Unified Device Architecture, fornisce due set di API, una a basso ed un ad alto livello. Supporta Windows, MacOS X e Linux ma funziona soltanto su hardware Nvidia anche se adesso ci sono dei tool per eseguire codice CUDA su GPU AMD.
- **HIP:** È l'equivalente di CUDA ma per AMD, esistono appunto dei tool che permettono di convertire codice CUDA in codice HIP.
- **OpenCL:** Open Computing Language è uno standard Open Source per scrivere programmi su piattaforme diverse come GPU, CPU o altri tipi di processori, è supportato sia da Nvidia che AMD ed è la piattaforma di sviluppo principale per AMD.
- **OpenACC:** Permette di utilizzare le direttive del compilatore per mappare automaticamente i calcoli da eseguire sulla GPU.

8.1. Cuda

Prima di CUDA, che permette di programmare con le schede Nvidia, le GPU trasformavano gli algoritmi in una sequenza di primitive per la manipolazione di immagini. CUDA inoltre:

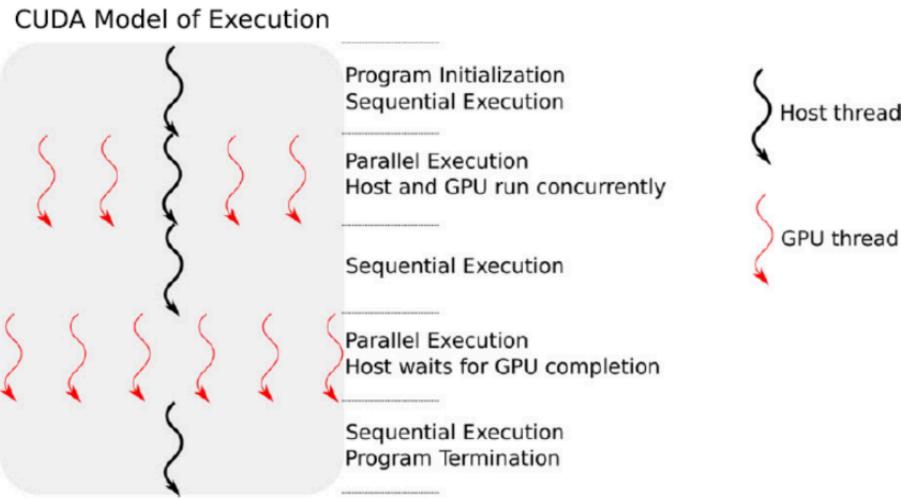
- Abilita la gestione esplicita della memoria.
- La GPU è vista come un dispositivo di computazione che:
 - È un co-processore della CPU o dell'host
 - Ha la sua DRAM
 - Esegue più threads in parallelo, la creazione o il cambio di questi costa comunque qualche ciclo di clock

È importante ricordare che CUDA **è una piattaforma o modello di programmazione e non è un linguaggio di programmazione.**

Struttura di un programma CUDA

1. Alloca la memoria della GPU
2. Trasferisci i dati dalla memoria dell'host alla memoria della GPU
3. Esegui il Kernel CUDA
4. Copia i risultati dalla memoria della GPU alla memoria host

Come viene eseguito, invece, del codice CUDA?



Nella maggior parte dei casi è l'host che si occupa delle operazioni di I/O, del passaggio e della raccolta dei dati dalla memoria della GPU.

I thread in CUDA sono organizzati in una struttura 6-D ma sono possibili anche con dimensioni più piccole. La struttura è divisa in:

- **Griglia**: Qui vengono posizionati i blocchi e possono avere quindi fino a 3 dimensioni per essere individuati
- **Blocco**: All'interno dei blocchi ci sono i thread, anche qui possiamo avere fino a 3 dimensioni per individuare ciascun thread.

Grazie a questo coordinate ogni thread riesce a capire a quale subset dei dati è stato assegnato.

La dimensione dei blocchi e delle griglie è determinata dalla **capability** che determina, per ogni generazione di GPU, quanto può elaborare. La **compute capability** di un dispositivo è rappresentata dal **version number** chiamato anche **SM version**. Questo numero identifica le features supportate dalla GPU e viene anche utilizzate a runtime dalle applicazioni per capire se delle istruzioni sono disponibili o meno.

8.1.1. Scrivere Programmi Cuda

Bisogna scrivere una funzione che verrà eseguita da tutti i threads, questa funzione è chiamata **kernel**, dobbiamo specificare come i threads sono organizzati nelle griglie / blocchi:

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

CUDA

- `dim3` è un vettore di interi
- Ogni componente non specificato è impostato a 1 e ogni componente è accessibile con i campi `x,y,z`

Vediamo come dichiarare le dimensioni di griglie e blocchi:

```
1 dim3 b(3, 3, 3);
2 dim3 g(20, 100);
```

CUDA

```

3 foo<<<g, b>>>(); // Run a 20x100 grid made of 3x3x3 blocks
4 foo<<<10, b>>>(); // Run a 10-block grid, each block made by 3x3x3 threads
5 foo<<<g, 256>>>(); // Run a 20x100 grid made of 256 threads
6 foo<<<g, 2048>>>(); // Invalid, maximum block size is 1024 threads even for
7 compute capability 5.x
8 foo<<<5, g>>>(); Invalid, that specifies a block size of 20x100=2000 threads

```

Hello World in CUDA:

```

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 }

```

CUDA

- La funzione `hello()` può essere chiamata dall'host
- La funzione `hello()` è il kernel e va sempre dichiarata void, i risultati vanno copiati in modo esplicito dalla GPU alla memoria dell'host.
- `cudaDeviceSynchronize` serve a bloccare finchè il CUDA Kernel non è terminato

Per compilare ed eseguire, lanciamo il comando:

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

Function Decorations

- `__global__` : Può essere chiamata dall'host o dalla GPU ed eseguita sul device o GPU.
- `__device__` : Una funzione che viene eseguita sulla GPU e può essere soltanto chiamata dal kernel, ovvero dalla GPU
- `__host__` : Una funzione che può essere eseguita soltanto sull'host, di solito viene omesso a meno che non è usato in combinazione a `__device__` per indicare che la funzione può essere eseguita sia su host che device. Questo significa che ci saranno due versioni del codice compilato.

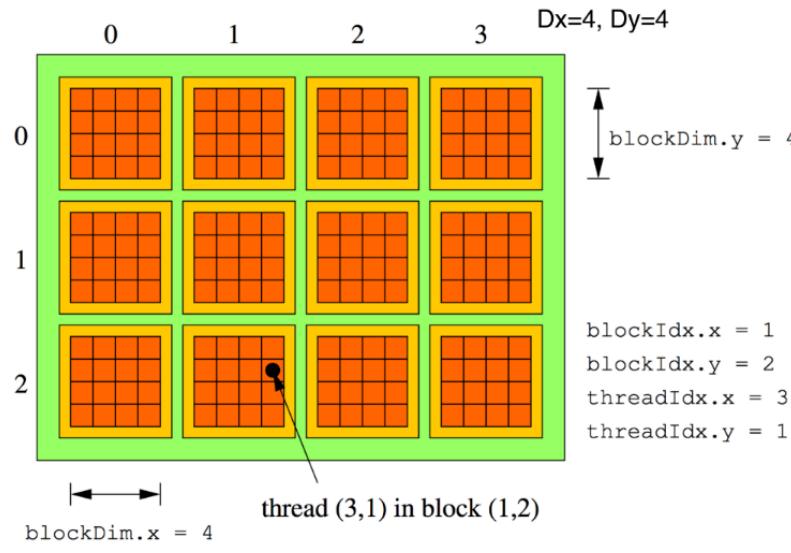
8.1.2. Ottenerere la posizione dei thread

I Thread sono organizzati in uno spazio 6D:

- `blockDim` : Contiene la dimensione di un blocco, (B_x, B_y, B_z)
- `gridDim` : Contiene la dimensione della griglia in blocchi, (G_x, G_y, G_z)
- `threadIdx` : Le coordinate (x, y, z) del thread all'interno del blocco con $x \in [0, B_x - 1], y \in [0, B_y - 1], z \in [0, B_z - 1]$

- `blockIdx` : Le coordinate (b_x, b_y, b_z) del blocco di cui fa parte il thread all'interno della griglia con $b_x \in [0, G_x - 1]$, $b_y \in [0, G_y - 1]$, $b_z \in [0, G_z - 1]$

Esempio



`threadIdx`:

$$\begin{aligned} x &= \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} = 1 \times 4 + 3 = 7 \\ y &= \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} = 2 \times 4 + 1 = 9 \end{aligned}$$

`threadId` = $9 \times 16 + 8$

Come prendiamo un unico thread ID?

- Threads diversi potrebbero avere lo stesso ID ma allora si trovano in blocchi differenti
- Per avere un ID unico vanno combinati `threadIdx` e `blockIdx` :

```

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

```

Quando i thread si trovano in spazi più piccoli di 6 dimensioni allora queste valgono 1 nella moltiplicazione e 0 nelle coordinate per la somma.

8.1.3. Thread Scheduling

- Ogni thread viene eseguito su uno Streaming Processor (CUDA Core)
- Un gruppo di cores nello stesso SM condividono la stessa control unit, questo significa che devono coordinarsi per eseguire la stessa istruzione.
- Diversi SM possono eseguire diversi kernel.
- Ogni blocco viene eseguito su un SM, quando uno di questi viene eseguito completamente allora l'SM esegue il successivo
- Non tutti i threads di un blocco vengono eseguiti in modo concorrente.

8.1.4. Warps

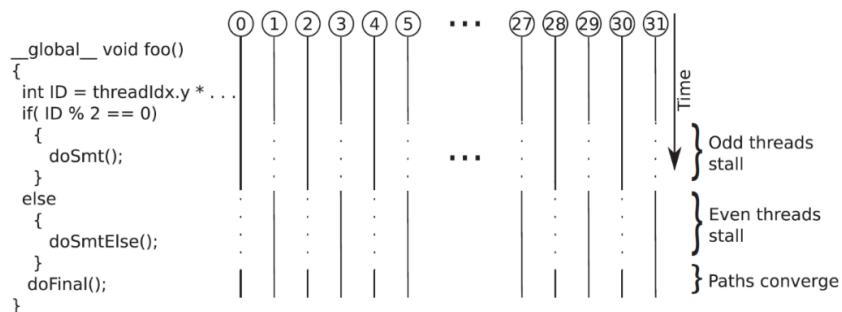
I thread di uno stesso blocco sono eseguiti in gruppi, chiamati **warps**, nelle GPU attuali la dimensione di un warp è di 32 ma potrebbe cambiare in futuro. Di solito vengono ordinate nei warps in base al loro intra-block ID.

Tutti i threads in un warp vengono eseguiti secondo il modello Single Instruction Multiple Data (SIMD) ovvero in qualsiasi istante viene eseguito il fetch di un'istruzione ed eseguita per tutti i threads nel warp.

Su ogni SM possono esserci diversi warps schedulers, questo significa che sullo stesso SM possono venir eseguiti più warps e ciascuno di questi con un execution path differente.

Warp Divergence

Dato che in un warp si segue il modello SIMD, cosa succede se il risultato di un'operazione condizionale porta a diversi percorsi? Vengono valutati tutti questi percorsi fino a quando non si riuniscono, i thread che non devono essere eseguiti vengono messi in stall.



Context Switching

Di solito un SM ha più blocchi / warps di quanti ne sia in grado di eseguire, questo perché un SM può switchare fra i warps infatti ogni thread il suo execution context mantenuto sul chip.

Quando un'istruzione che deve essere eseguita su un warp ha bisogno di aspettare i risultati di un'operazione precedente con tanta latenza allora il warp non viene selezionato per l'esecuzione ma ne viene selezionato uno che può partire subito.

Questo meccanismo di selezionare chi non deve aspettare nulla si chiama **latency tolerance** o **latency hiding**.

Con un numero sufficiente di warps il sistema sarà sempre in grado di trovare un warp eseguibile utilizzando al massimo l'hardware. Questo meccanismo è il motivo principale per cui le GPU non devono dedicare così tanto spazio a zone come la cache o meccanismi di branch predictions come fanno le CPU.

Esempio 1

- Un dispositivo CUDA permette fino a 8 blocchi con 1024 threads per SM e 512 threads per blocco

- Quanto facciamo grandi i thread blocks? 8x8, 16x16 o 32x32?
- 8x8:
 - Avremo 64 threads per blocco
 - Per riempire i 1024 threads di ciascun SM ci servono $1024 / 64 = 16$ blocchi
 - Ma possiamo avere al massimo 8 blocchi per SM, ma in questo caso avremo $64 \times 8 = 512$ threads per SM
 - Non stiamo utilizzando al massimo le risorse a disposizione, questo significa che in qualche momento lo scheduler potrebbe non trovare qualcosa da eseguire.
- 16x16:
 - 256 threads per blocco
 - Per riempire i 1024 threads di ogni SM ci servono $1024 / 256 = 4$ blocchi
 - Abbiamo 1024 threads per ogni SM, perfetto
- 32x32:
 - Abbiamo 1024 threads per blocco che è molto più grande di 512 ovvero il massimo che possiamo avere.

Esempio 2